Architecting a Network Query Engine for Producing Partial Results

Jayavel Shanmugasundaram[1,3], Kristin Tufte[1,2], David DeWitt[1],
David Maier[2], Jeffrey Naughton[1]

[1] Department of Computer Sciences
University of Wisconsin-Madison, Madison, WI 53706, USA
{jai, dewitt, naughton}@cs.wisc.edu
[2] Department of Computer Science
Oregon Graduate Institute, Portland, OR 97291, USA
{tufte, maier}@cse.ogi.edu
[3] IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120, USA

Abstract. The growth of the Internet has made it possible to query data in all corners of the globe. This trend is being abetted by the emergence of standards for data representation, such as XML. In face of this exciting opportunity, however, existing query engines need to be changed in order to use them to effectively query the Internet. One of the challenges is providing partial results of query computation, based on the initial portion of the input, because it may be undesirable to wait for all of the input. This situation is due to (a) limited data transfer bandwidth (b) temporary unavailability of sites and (c) intrinsically long-running queries (e.g., continual queries or triggers). A major issue in providing partial results is dealing with non-monotonic operators, such as sort, average, negation and nest, because these operators need to see all of their input before they can produce the correct output. While previous work on producing partial results has looked at a limited set of non-monotonic operators, emerging hierarchical standards such as XML, which are heavily nested, and sophisticated queries require more general solutions to the problem. In this paper, we define the semantics of partial results and outline mechanisms for ensuring these semantics for queries with arbitrary non-monotonic operators. Re-architecting a query engine to produce partial results requires modifications to the implementations of operators. We explore implementation alternatives and quantitatively compare their effectiveness using the Niagara prototype system.

1 Introduction

With the rapid and continued growth of the Internet and the emergence of standards for data representation such as XML [1], exciting opportunities for querying data on the Internet arise. For example, one might issue queries through web browsers rather than relying on semantically impoverished keyword searches. An important and chal-

lenging research issue is to architect query engines to perform this task. Some of the main issues in designing such query engines are to effectively address (a) the low network bandwidth that causes delays in accessing the widely distributed data, (b) the temporary unavailability of sites and (c) long running triggers or continual queries that monitor the World Wide Web. An elegant solution to these problems is to provide partial results to users. Thus, users can see incomplete results of queries as they are executed over slow, unreliable sites or when the queries are long running (or never terminate! [2]).

A major challenge in producing partial results is dealing with non-monotonic operators, such as sort, average, sum, nest and negation. Since the output of these operators on a subset of the input is not, in general, a subset of the output on the whole input, these operators need to see all of their input before they produce the correct output. Previous solutions to the problem of producing partial results proposed by Hellerstein et al. and Tan et al. present solutions for specific non-monotonic aggregate operators, such as average [3,7], and thus do not extend to non-monotonic operators such as nest and negation that are becoming increasingly important for network query engines. Further, the previous solutions do not allow non-monotonic operators to appear deep in a query plan. Thus, for example, a query that asks for all BMW cars that do not appear on salvage lists and that cost less than 10% of the average price of cars in its class is a query that cannot be handled by previous techniques. Neither could a query that requests an XML document where books are nested under author, and authors are nested under state, and states are further nested under country. (The non-monotonic operators in the first query are “not in” and “average” while the non-monotonic operators in the second query is “nest”).

A main contribution of this paper is the development of a general framework for producing partial results for queries involving any non-monotonic operator. A key feature of this framework is that it provides a mechanism to ensure consistent partial results with unambiguous semantics. The framework is also general enough to allow monotonic and non-monotonic operators to be arbitrarily intermixed in the query tree (as in the examples above), i.e., monotonic operators can operate on the results of a non-monotonic operator and vice-versa. It is important to note that the framework by itself does not stipulate any particular implementation of non-monotonic operators but merely identifies some abstract properties that operator implementations need to satisfy (indeed, much of the generality of the framework is precisely due to this). Interestingly, these properties affect both monotonic and non-monotonic operator implementations. Another contribution of this paper is the identification of implementations for operators that satisfy the desired properties and a performance evaluation of the various alternatives using our prototype system.

1.1 Relationship to Other Work

As mentioned above, most of the previous research by Hellerstein et al. and Tan et al. on partial results has been in the context of particular aggregate functions such as sum and average [3,7]. Further, they deal with at most one level of nesting of non-

monotonic operators [7]. The main focus of this paper is to provide a general framework whereby queries with arbitrary non-monotonic operators appearing possibly deep in the query tree can produce partial results. Thus, techniques developed for specific cases, such as aggregates fit in easily in our framework and can exploit the general system architecture. For instance, it would be possible to integrate the methods for relaying accuracy [3] into our system. The added flexibility is that the operators can appear anywhere in the query tree, mixed with other monotonic and non-monotonic operators (see Section 3.3).

There has been some work on non-blocking implementations (i.e., implementations that produce some output as soon as they see some input) of monotonic operators so that results can be sent to the user as soon as they are produced. Urhan and Franklin have proposed a non-blocking implementation of join [8]. Ives et al. describe an adaptive data integration system that uses non-blocking operators to address issues including unpredictable data arrival [5] and have also proposed an operator, x-scan, for incrementally scanning and parsing XML documents [4]. There has also been work on modifying the query plans so that network delays can be (partially) hidden from the user [9]. These approaches, while partially addressing the problem of low network bandwidth and unavailable sites, do not address the general problem because a query may require non-monotonic operators, such as nest and average. In these cases, unless we provide partial results, the query execution has to block until all the data is fetched.

1.2 Roadmap

The rest of the paper is organized as follows. Section 2 formally defines the semantics of partial results and identifies some key properties that query engine operator implementations need to satisfy in order to produce complete and meaningful partial results. Section 3 proposes a system architecture that produces consistent partial results and Section 4 identifies alternative operator implementation techniques. Section 5 provides a performance evaluation of the various operator implementation strategies and Section 6 concludes the paper and outlines our ideas for future work.

2 Partial Results and Implications for Operator Implementations

In the previous section, we illustrated the need for producing partial results for queries having arbitrary non-monotonic operators appearing deep in the query plan. Having such a general notion of partial results does not come without associated challenges. The following questions immediately come to mind: What are the semantics of partial results? Can we use traditional query engine architectures and associated operator implementations to produce partial results? If not, then what are the modifications that need to be made? This section is devoted to the above questions. We begin by briefly outlining the structure of traditional query engines. We then formally define the semantics of partial results and identify key properties of operator

implementations, not supported by traditional query engine architectures, which are crucial for partial result production. These properties lay the foundations for designing operator implementations capable of producing correct, maximal partial results.

2.1 The Traditional Query Engine Architecture

A common way of executing a query is to structure it as a collection of operators, each of which transforms one or more input streams terminated by an End of Stream (EOS) element and produces one or more output streams, also terminated by an EOS element. Thus an operator defines the transformation that input streams undergo in order to produce the output stream. Typical operators include Select, Project, Join and Group-by. The query execution can be represented as a directed graph, where each operator is a node and each stream is represented as a directed edge from the operator writing into the stream to the operator reading from the stream.

As an example, consider a query that asks for the details of all cars priced less than 10% of the average price of cars of the same model. Figure 1 shows a graphical representation of an operator tree for this query. The replicate operator produces two identical output streams containing the car information. The replication captures the fact that the car information is a common sub-expression used in two places. The first output stream of the replicate operator feeds to the average operator, which computes the average selling price for each model of cars. The second output stream of the replicate operator feeds to the join operator that relates a car's price to the average price of cars of that model.

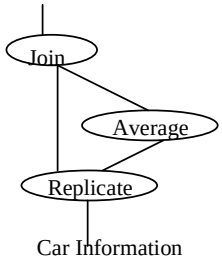

Figure 1

Each operator in a query graph potentially has many implementations. Each implementation defines a particular mechanism to achieve the transformation specified by the operator. For example, a join operator can have nested-loops and hash-based implementations, while an average operator can have both sort-based and hash-based implementations. Each operator implementation again operates on input streams and produces output streams. The output of an operator implementation is well defined even on input streams not terminated by an EOS element because it represents the output of the operator implementation on seeing the input streams "so far", at the current point in time. On the other hand, there is no notion of "time" for operators, and so their output is well defined only on input streams terminated by an EOS element.

The relationship between operators and their implementations is formalized below. For ease of exposition, the formal discussion is restricted to unary operators. It is easy to generalize to n-ary operators based on later descriptions of operator implementations. We begin by defining stream and sub-stream.

Definition D1: A ***stream*** is a sequence (ordered collection) of data elements. A ***sub-stream*** is a contiguous sub-sequence of the original stream sequence.

Definition D2: An operator implementation, O, is an ***implementation*** *of* an operator, Oper, if for all input streams I not having an EOS element, Oper(I.EOS) = O(I.EOS).

The previous definition just says that an operator implementation (O) should produce the same output as the operator it implements (Oper) at the point in time when all the inputs have been seen (i.e., the point in time when the input stream has been terminated by an EOS element). However, this definition requires nothing about what the implementation emits before the EOS. Traditionally the input stream(s) and output stream of each operator implementation are monotonically increasing, i.e., data is only added to the streams, never updated or removed. Thus, operator implementations and streams are structured to consume and produce only additions. This restriction limits traditional query engine architectures for producing partial results. To see why this is the case, we need to first formally define the semantics of partial results and understand the behavior of operators and their implementations. We will then be in a position to identify the properties, not satisfied by traditional query engines, that nevertheless need to be satisfied by operator implementations in order to produce maximal and correct partial results.

2.2 Preliminaries: Partial Results, Operators, Operator Implementations

We begin by defining the notion of a partial result of a query.

Definition D3: Let Q be a query with input I and let Q(I) represent the result of the query Q on input I. A ***partial result*** of a query Q, given an input I, is Q(PI), where PI is a sub-stream of I.

Intuitively, a partial result of a query on an input stream is the result of the query on a (possibly) different input stream such that the new input stream is a sub-stream of the original input stream. We proceed to formally define monotonic and non-monotonic operators and blocking and non-blocking implementations and provide a theorem connecting the concepts of monotonicity and blocking.

Definition D3: An operator Oper is a ***monotonic operator*** if for all input sub-streams I, J not having an EOS element, if I is a sub-stream of J, then Oper(I.EOS) is a sub-stream of Oper(J.EOS). An operator is a ***non-monotonic operator*** if it is not a monotonic operator.

Intuitively, a monotonic operator is one that given additional input produces additional output without needing to modify previously produced output. As an illustration, consider the average operator in the example in Figure 1. This operator takes an input stream having (car-model, car-price) pairs and computes the average car-price for each car-model. The output of this operator on the input I1 = (“Toyota”, 10000).EOS is O1 = (“Toyota”, 10000).EOS. Its output on the input I2 = (“Toyota”, 10000).(Toyota, 20000).EOS is O2 = (“Toyota”, 15000).EOS. Since I1 is a sub-stream of I2 but O1 is not a sub-stream of O2, the average operator is not monotonic.

Select and join operators are examples of monotonic operators. For example, consider a select operator that selects all (car-price) tuples that have a car-price less than

10000. Its output on the input I1 = (5000).(10000).EOS is the output O1 = (5000).EOS. Its output on the input I2 = (5000).(10000).(7000).EOS is the output O2 = (5000).(7000).EOS. Here I1 is a sub-stream of I2 and O1 is a sub-stream of O2. In general, operators such as the select and join operators always add more data to the output when they see more data on their inputs. They are thus monotonic.

An interesting case is the nest or group-by operator. Formally, a monotonic function, *f*, on inputs x, y, is a function such that $x \leq y \Rightarrow f(x) \leq f(y)$. Traditionally, a database operator, Oper, is considered monotonic if $I \subseteq J \Rightarrow Oper(I) \subseteq Oper(J)$ where I and J are sets. In this case, "less than (or equal to)" is interpreted as subset. The extension to nested structures is, however, not straightforward. Consider a query that nests (CarMake, CarModel) pairs on CarMake to produce (CarMake, {Set of Car Model}) pairs. The table below shows possible initial and subsequent inputs and results for the query.

Initial Input	**Initial (Partial) Result**
(Toyota, Camry)	(Toyota, {Camry})
(Honda, Accord)	(Honda, {Accord, Prelude})
(Honda, Prelude)	
Subsequent Input	**Subsequent Result**
(Toyota, Corolla)	(Toyota, {Camry, Corolla})
	(Honda, {Accord, Prelude})

At issue is whether Initial Result is "less than" Subsequent Result. Depending on your viewpoint, it may or may not be. The real question is how to extend "less than" to nested structures such as XML documents (or sequences of such structures). There are two obvious possibilities:

1. "Less than" is interpreted as "subset" – $Oper(A) \leq Oper(B)$ means that all the elements (pairs) in Oper(A) are in Oper(B).
2. "Less than" is interpreted as "substructure" – $Oper(A) \leq Oper(B)$ means that all the elements (pairs) in Oper(A) are sub-structures of some element (pair) in Oper(B). Here, sub-structure is a "deep" (recursive) subset relationship.

Under the first interpretation listed above, Initial Result is not "less than" Subsequent Result and nest would not be considered monotonic. Under the second interpretation, Initial Result is "less than" Subsequent result and nest would be considered monotonic. Both interpretations of "less than" are valid; the interpretation chosen should be determined by the query processing framework and how that framework interprets nested structures. We use the first interpretation since our system does not support nested updates.

Definition D4: An operator implementation O is a ***non-blocking operator implementation*** if for all input sub-streams I not having an EOS element, O(I.EOS) = O(I).EOS. An operator implementation is a ***blocking operator implementation*** if it is not non-blocking.

Intuitively, a non-blocking operator implementation is one that does not "block" waiting for the EOS notification to produce results. The EOS notification on its input stream can cause it only to send an EOS notification on its output stream. As an illus-

tration, consider a hash-based implementation of the average operator in Figure 1. This implementation, on seeing an input (car-model, car-price) pair, hashes on the car-model to retrieve the node in the hash table that stores the number of tuples and the sum of the car-prices in these tuples, for the given car-model. It increments the number of tuples by one, and adds the new car-price to the running sum, and then proceeds to process the next input tuple. On seeing an EOS element, it divides the sum of the car-prices for each car-model by the number of cars seen for that car-model and writes the (car-model, avg-car-price) pairs to the output stream. This operator implementation is blocking because it does not produce any output until it sees an EOS element.

On the other hand, consider an implementation of a select operator that selects all (car-price) tuples that have a car-price less than 10000. This implementation looks at each tuple and adds it to the output stream if the car-price value is less than 10000. On seeing the EOS element in the input, it just adds the EOS element to the output. This is thus a non-blocking operator implementation of the select operator.

The following theorem relates monotonic operators and non-blocking operator implementations. We use the notation O(I/J) to denote the output of an operator implementation O on the input I when it has already seen (and output the results corresponding to) the input sub-stream J.

Theorem T1: An operator Oper is monotonic if and only if Oper has a non-blocking operator implementation O.

Proof: (If Part) Consider a non-blocking operator implementation O of Oper. We must prove that for all input streams I, J not having an EOS element, if I is a sub-stream of J then Oper(I.EOS) is a sub-stream of Oper(J.EOS). Consider any two input sub-streams I, J not having an EOS element such that I is a sub-stream of J. There exists a input sub-stream K such that I.K = J. Now:

Oper(I.EOS) = O(I.EOS)	(by definition D1)
= O(I).EOS	(because O is non-blocking)
is a sub-stream of O(I).O(K/I).EOS	(by definition of sub-streams)
is a sub-stream of O(I.K).EOS	(by definition of O(K/I))
is a sub-stream of O(I.K.EOS)	(because O is non-blocking)
is a sub-stream of O(J.EOS)	(because I.K = J)
is a sub-stream of Oper(J.EOS)	(by definition D1)

(Only If Part) Consider a monotonic operator, Oper. Now consider the operator implementation O that works as follows. Let PrevI denote the input stream seen so far. On a new input element e that is not an EOS element in the input stream, the output of O, i.e. O(e/PrevI), is the sub-stream J such that PrevOpt.J = CurrOpt. Here PrevOpt is the output stream Oper(PrevI.EOS) without the EOS element, and CurrOpr is the output stream Oper(PrevI.e.EOS) without the EOS element. Such a J always exists because Oper is monotonic. When O sees an EOS, it simply puts an EOS element to the output stream. It is easy to see that O is an operator implementation of Oper. Also, by the definition of O, O(I.EOS) = O(I).EOS for all input streams I. Thus, O is a non-blocking operator implementation of Oper. (*End of Proof*)

We are now in a position to study the properties that operator implementations need to satisfy in order to produce partial results.

2.3 Desirable Properties of Operator Implementations

Theorem T1 has important implications for the production of partial results. These are best brought out by means of an example. Consider the query in Figure 1 that asks for all cars that cost less than 10% of the average price of cars of the same model. The average operator is non-monotonic because the average cost for a given model potentially changes as more inputs are seen. The join operator, on the other hand, is monotonic because no future input can invalidate the join between two previous inputs.

Consider a scenario where the query plan has been running for a few seconds and car information for a few cars has been sent as input to the query processor, but the query processor is still waiting for more inputs over an unreliable network channel. If the user now desires to see the partial results for the query, then the average operator implementation must output the average "so far" for each car model seen so that the join operator implementation can join the average price for each model with the cars seen "so far" for that model. The only way the user will automatically see the partial results is if all the operators in the query graph have been implemented with non-blocking implementations. However, from Theorem T1, we know that average does not have a non-blocking implementation. In order to produce partial results, the blocking implementation for average has to be able to produce the result "so far" on request. In general, in order to produce partial results, all blocking operator implementations for non-monotonic operators need to be structured in such a way that they can produce the result "so far" at any time. We refer to this as the ***Anytime*** property for blocking operator implementations.

The fact that blocking operator implementations for non-monotonic operators need to produce results "so far" at any time has other implications. In our example, the average price transmitted per model is potentially wrong because there can be more inputs for a given model, which can change the model's average price. When more inputs are seen and the average price per model changes, this change must be transmitted to the join operation above (which needs to transmit it to the output). In general, the fact that an operator is non-monotonic implies that the result "so far" transmitted to higher operators can be wrong. Therefore, there needs to be some mechanism to "undo" the wrong partial outputs (change the average price for a given model, in our example). In other words, operator implementations need to be capable of producing non-monotonic output streams (as in the case of the average operator implementation) and processing non-monotonic input streams (as in the case of the join operator implementation). Note that both blocking and non-blocking operators need to handle non-monotonic input and output streams as they can be arbitrarily placed in the query graph. We refer to this as the ***Non-Monotonic Input/Output*** property for operator implementations. We now turn our attention to another property of operator implementations that is useful for producing partial results.

Intuitively, the ***Maximal Output*** property requires that operator implementations produce results as soon as possible. That is, the operator implementation puts out as much of the result as it can without potentially giving a wrong answer. This property is useful for producing partial results because the user can see all the correct results

that can possibly be produced, given the inputs seen so far. For example, consider the non-monotonic operator "outer join." Operator implementations for this operator can output the joining results as soon as they are produced, without having to wait for the end of its inputs. The ***Maximal Output*** property is formally defined below.

Definition D5: Let Oper be an operator and O an implementation of Oper. Let I be a stream of elements in the domain of Oper. O satisfies the ***Maximal Output*** propert***y*** if O(I).EOS is the maximal stream such that it is a sub-stream of Oper(I.K.EOS), for every K, where K is a stream of elements from domain of Oper.

It is easy to see that all non-blocking operator implementations automatically satisfy the Maximal Output property. It turns out that, in fact, there is a stronger relationship between these two properties, as exemplified by the following theorem.

Theorem T2: An operator implementation O of an operator Oper is non-blocking if and only if Oper is monotonic and O satisfies the Maximal Output property.
Proof: (If Part) Assume Oper is monotonic and O satisfies the Maximal Output property. We must prove that for all input sub-streams, I, not having EOS, O(I).EOS = O(I.EOS). Since Oper is monotonic, by definition D3, Oper(I.EOS) is a sub-stream of Oper(I.J.EOS) for all J. Since O satisfies the Maximal Output property, O(I).EOS is the maximal sub-stream of Oper(I.K.EOS), for all K. We can thus infer that O(I).EOS = Oper(I.EOS) which implies that O is non-blocking.
(Only If Part) Assume O is a non-blocking operator implementation. Since non-monotonic operators cannot have non-blocking operator implementations (Theorem T1), O must be an implementation of a monotonic operator, Oper. It remains to be shown that O satisfies the Maximal Output property. For every K, a stream of elements from the domain of Oper, we know that Oper(I.EOS) is a sub-stream of Oper(I.K.EOS) because Oper is monotonic. By the definition of a non-blocking operator implementation, we have O(I).EOS = Oper(I.EOS). This implies that O(I).EOS is the maximal sub-stream of Oper(I.K.EOS), for all streams K in the domain of Oper. Hence O satisfies the maximal output property. *(End of Proof)*

The theorem above essentially states that for operator implementations of monotonic operators, the maximal output property and the non-blocking property are the same thing. Thus ensuring that all operator implementations satisfy the maximal output property automatically ensures that all monotonic operators will have non-blocking operator implementations.

A final operator implementation property that is useful for producing partial results is what we call the ***flexible input*** property. This property essentially states that operator implementations should not stall on a particular input stream if there is some input available on some other input stream. The motivation behind this property is that in a network environment, traffic delays may be arbitrary and data in some input streams may arrive earlier than data in other input streams, and it may be impossible to determine this information a priori. Thus, in order to provide up-to-date partial results at any time, operators need to be able to process information from any input stream, without stalling on any particular input stream. Many traditional operator implementations do not satisfy this property. Consider, for example, typical imple-

mentations of the join operator. The nested loops join operator implementation requires all the tuples of the inner relation to be present before it can process any tuple in the outer relation. Similarly, the hash join operator implementation requires the whole inner relation (to build the hash table) before it can process the outer relation. Symmetric hash join [10] and its variants [8,4] are the only join operator implementations that satisfy this property. In order to provide partial results effectively, traditional implementations will have to give way to the "flexible input" variants.

To summarize, in this section we formally defined the semantics of partial results and developed the notions of monotonic and non-monotonic operators and blocking and non-blocking operator implementations. We then identified certain key properties, namely the Anytime, Non-monotonic Input/Output, Maximal Output and Flexible Input properties, that operator implementations need to satisfy in order to provide partial results.

Before we turn our attention to the design of operator implementations satisfying the above properties, let us pause for a moment to ask whether these properties in isolation are sufficient to ensure the semantics of partial results as defined above. It turns out that while the above properties are sufficient to produce partial results upon user-request, they are not sufficient to ensure that the partial results are consistent. The next section is devoted to studying this problem and proposing a solution. We tackle the issue of designing operator implementations in Section 4.

3 Consistency of Partial Results and its Implications for a Query Engine Architecture

As mentioned in Section 2.1, the query execution graph can be represented as a graph with the nodes representing operators and the edges representing streams. In general, this graph is not a tree but a Directed Acyclic Graph (DAG), as shown in Figure 1. This form is due to the presence of common sub-expressions in a query (in our example, the car information is the common sub-expression, which is replicated along two separate paths). The fact that the operator graph can be a DAG has important implications for the architecture of a system designed to produce partial results. The definition of partial results requires that the partial output be the result of executing the query on a subset of the inputs. This requirement implies that any data item that is replicated must contribute to the partial result along *all* possible paths to the output or not contribute to the output at all (along any path). This condition is necessary to avoid anomalies such as selecting cars below the average price, without the car's price being used to compute the average (see Figure 1). Note that this issue does not arise when constructing only final results because each operator then produces results based on all of the inputs it sees. This potential inconsistency is because of our desire to interrupt input streams in order to produce partial results.

A related issue also arises when an operator logically produces more than one output data item corresponding to a single input data item. This situation might arise, for instance, in a join operator when a single tuple from one input stream joins with more

than one tuple from the other input stream and produces many output tuples. Another example where this can arise is while projecting many set sub-elements (say employees) from a single element (say department) in XML documents. In these cases again, we need to ensure that the partial query result includes the effects of all or none of the output data items that correspond to a single input data item.

3.1 Synchronization Packets

We now show how the notion of *synchronization packets* can be used to ensure the consistency of partial results. Conceptually, these packets are inserted in the input streams of the query plan whenever a partial result is desired as is shown in Figure 2. These synchronization packets are replicated whenever a stream is replicated and their main function is to "synchronize" input streams at well-defined points so that operator implementations see consistent partial inputs. More precisely, each operator uses all the data "before" the synchronization packets in the production of partial results. In the example above, the join operator implementation uses all the data before the synchronization packets in the production of partial results. This ensures that every data item reaching the join directly from replicate is also reached through average (and vice versa). The problem now is to determine how the synchronization information is to be propagated up the operator graph. We propagate synchronization packets up the operator graph by following the rules below:

1. If there is a synchronization packet received through an input stream of an operator implementation, then no further inputs are taken from that input stream until synchronization packets are received through all input streams
2. Once synchronization packets are received from all input streams of an operator implementation, the operator implementation puts its partial results (in the case of a blocking operator implementation) and synchronization packets into its output stream(s). It is important to note that the Anytime property of blocking operators is used here for partial result production and synchronization packet propagation.

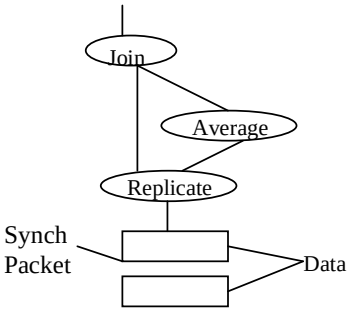

Figure 2

The following theorem shows that these rules are sufficient to guarantee consistent partial results. We assume that all common sub-expressions have a replicate operator that produces many output streams, as in Figure 1.

Theorem T3: If the synchronization packets are inserted into the input streams when partial results are desired, the synchronization rules guarantee that the partial results produced are consistent.

Proof Sketch: Since the operator graph is a DAG, there exists a topological ordering of the operators in the graph such that each operator in the graph appears before all

the operators reachable from it. The proof uses induction on the position of operators in the topological ordering to prove that for the implementation of each operator, its partial output is based on all and only the data of the input streams occurring before the synchronization packets (and is hence consistent). Thus implying that the top-level operator implementation's output (the query's output) is consistent.

3.2 Partial Request Propagation and Generation

In the previous section, we assumed that synchronization packets are inserted into input streams when partial results are required. Typically, however, the user or application has access only to the output stream of the top-level operator because the operators of the query plan can be distributed at various sites in the network. Thus, user and application requests must be propagated down the operator graph. This can be achieved by propagating control messages from the user or application to the base of the operator graph. Once the partial result request control messages reach the base of the operator graph, they must be intercepted and synchronization control messages must be inserted into the input streams. "Partial" operators provide this functionality.

Partial operators are added to the base of the operator graph and perform two simple functions: (a) propagate the data from the input stream unchanged to the output stream and (b) on receiving a partial result request from an output stream, they send a synchronization packet to their output streams. Thus partial operators provide an automatic way of handling synchronization packets.

Using partial operators to handle synchronization packets allows us to explore algebraic equivalences between partial operators and other operators. These equivalences can be used to move partial operators up in the operator graph (and even merged) under certain conditions. This "transformation" of the operator graph is likely to lead to better response times because synchronization packets travel less far down the operator graph and because operators below the partial operators do not have to be synchronized. We plan to study these equivalences in more detail as part of future work.

To summarize this section, we outlined consistency anomalies that can arise while producing partial results and proposed solutions using the notions of synchronization packets and partial operators. Together with the operator implementation extensions discussed in Section 2, they extend the traditional query engine architecture to support the generation of consistent partial results. A key part of the puzzle, however, remains to be solved – the design of operators satisfying the properties outlined above. We turn to this issue next.

4 Operator Implementation Alternatives

We explore two alternatives, Re-evaluation and Differential, for modifying existing operator implementations so that they satisfy the desired properties for producing partial results outlined in Section 2.3. The Re-evaluation approach retains the struc-

ture of existing operator implementations but requires the re-execution of all parts of the query plan above the blocking operators. Alternatively, the Differential approach processes changes as part of the operator implementation, similar to the technique used in the CQ project [6], and avoids re-execution. There is thus a trade-off between the complexity of the operators and their efficiency: Re-evaluation implementations are easier to add to existing query engines, while Differential implementations are more complex and require changes to the tuple structure, but are likely to be more efficient.

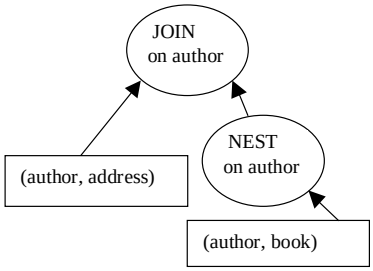

Figure 3

The Re-evaluation and Differential approaches are similar in that, for monotonic operators, they use existing non-blocking and flexible input operator implementations where possible. For example, joins are implemented using symmetric hash join [10] and symmetric nested loops join algorithms (or their variants [4,8]). The algorithms in this section extend such non-blocking and flexible input operator implementations to satisfy the non-monotonic input/output property and further, identify blocking operator implementations satisfying all four desirable properties.

4.1 Re-evaluation Approach

In order to satisfy the Non-monotonic Input/Output property, we must determine what form partial results produced by blocking operators take and how updates to those results are communicated. The Re-evaluation approach handles this decision straightforwardly by having blocking operator implementations transmit their current result set when a partial result request is received. If there are multiple partial result requests, the same results will be transmitted multiple times. Note that all operator implementations above a blocking operator implementation must re-evaluate the query each time a partial result request is issued; hence the name Re-evaluation approach.

Consider the query execution graph in Figure 3 which shows a nest operator reading (author, book) pairs from an XML file on disk (or any non-blocking operator implementation), nesting the pairs on author and sending its output to a join operator implementation. The nest implementation is blocking; the join implementation is non-blocking. Upon receipt of a partial result request, the nest operator implementa-

tion transmits all (author, <set of books>) groups it has created so far to the join. At this point, the join implementation must ignore all input it has previously received from nest, and process the new partial result as if it had never received any input from nest before. We describe the re-evaluation implementations of join and nest below. Descriptions of other operator implementations are omitted in the interest of space.

Re-evaluation Join: The Re-evaluation join implementation functions similar to a symmetric hash join implementation except that when the Re-evaluation join implementation is notified that a new partial result set is beginning on a particular input stream, it clears the hash table associated with that input stream. In addition, special techniques are used to deal with the case when an input contains a mixture of tuples that are "final" – produced by a non-blocking operator and will never be repeated and tuples that are "partial" – produced by a blocking operator (as part of a partial result set) and will be retransmitted at the start of the next partial result. The intermixing of partial and final tuples can occur if the input comes from a union operator implementation, which unions the output of a blocking and non-blocking operator implementation or from an operator such as outer join that produces final tuples before EOS and partial tuples upon request for a partial result.

Re-evaluation Nest: Similar to a traditional hash-based nest implementation, the Re-evaluation nest implementation creates a hash table entry for each distinct value of the grouping attribute (author in our example). When a partial result notification is received, the Re-evaluation nest implementation acts lazily and does not delete the hash table. Instead, the Re-evaluation nest implementation simply increments a partial result counter. Upon insert into the hash table, each book tuple is labeled with the current counter value. When an entry is retrieved during nest processing, all books having counter value less than the counter value of the operator are ignored and deleted. We utilize this lazy implementation because when the input consists of a mixture of partial and final tuples, they will be combined in the <set of book> entries in the hash table. Deleting all obsolete book tuples in an eager fashion would require retrieving and updating most of the hash table entries, which is too expensive.

4.2 Differential Approach

The Re-evaluation approach is relatively easy to implement, but may have high overhead as it causes upstream operators to reprocess results many times. The Differential approach addresses this problem by having operators process the changes between the sets of partial results, instead of reprocessing all results. Differential versions of traditional select, project and join are illustrated and formalized by Lin et al. [6] in the context of continual queries. Our system, however, handles changes as the query is being executed as opposed to that approach, which proposes a model for periodic re-execution of queries. This difference gives rise to new techniques for handling changes as the operator is in progress.

In Figure 3, in order for the join to process differences between sets of partial results, the nest operator implementation must produce the "difference" and the join

operator implementation must be able to process that "difference." We accomplish this "differential" processing by having all operators produce and consume tuples that consist of the old tuple value and the new tuple value, as in Lin et al. [6]. Since the partial results produced by blocking operator implementations consist of differences from previously propagated results, each tuple produced by a blocking operator implementation is an insert, delete or update. In the interest of space, we describe only the differential join and nest operator implementations below.

Differential Join: The Differential join implementation is again based on the symmetric hash join implementation. A Differential join implementation with inputs A and B works as follows. Upon receipt of an insert of a tuple τ into relation B, τ is joined with all tuples in A's hash table and the joined tuples are propagated as inserts to the next operator implementation in the query execution graph. Finally τ is inserted into B's hash table for joining with all tuples of A received in the future. Upon receipt of a delete of a tuple τ from relation B, τ is joined with all tuples in A's hash table and the joined tuples are propagated as deletes to the next operator in the tree. Updates are processed as deletes followed by inserts.

Differential Nest: The Differential nest implementation is similar to the traditional hash-based nest implementation. Inserts are treated just as tuples are in a traditional nest operator implementation. For deletes, the Differential nest operator implementation probes the hash table to find the affected entry and removes the deleted tuple from that entry. For updates, if the grouping value is unchanged, the appropriate entry is pulled from the hash table and updated, otherwise, the update is processed as a delete followed by an insert. Changes are propagated upon receipt of a partial result request. Only the groups that have changed since the last partial request are propagated on receipt of a new partial request.

4.3 Accuracy of Partial Results

In the previous sections, we have concentrated on operator implementations that produce partial results. An important concern is the accuracy of the results produced. We believe that our framework is general enough to accommodate various techniques for computing the accuracy of partial results, such as those proposed for certain numerical aggregate operators [3,7]. These techniques can be incorporated into our framework if the desired statistics are passed along with each tuple produced by an operator. In addition, our framework allows non-monotonic operators (such as aggregates) to appear anywhere in the query tree. It is also important to address accuracy of partial results for non-numeric non-monotonic operators such as nest and except. Providing information about the accuracy of these operators is more difficult because we do not have notions such as "average" and "confidence intervals" in these domains. It is, however, possible to provide the user with statistics such as the percentage of XML files processed or the geographical locations of the processed files. The user may well be able to use this information to understand the partial result.

5 Performance Evaluation

This section compares the performance of the Re-evaluation and Differential approaches for implementing operators. We begin by describing the experimental set up in Section 5.1 and outline our performance results in Section 5.2.

5.1 Experimental Setup

Our system is written in Java and experiments were run using JDK 1.2 with 225MB of memory on a Sun Sparc with 256MB of memory. Our system assumes that the data being processed is resident in main memory. Though we expect this assumption to be acceptable for many cases given current large main memory sizes, we plan to explore more flexible implementations that handle spillovers to disk in the future.

We used three queries to evaluate the performance of the Re-evaluation and Differential approaches. The first query (Q1) contains a join over two blocking operators. The input is two XML documents, one having flat (author, book) pairs and the other having flat (author, article) pairs. It produces, for each author, a list of articles and a list of books written by that author. Q1 is executed by nesting the (author, book) and (author, article) streams on author and (outer) joining these streams on author to produce the result. Finally, a construct operator is used to add tags. The number of books (articles) for each author follows a Zipfian distribution.

The second query (Q2) is similar to Q1 except that the inputs are (author, bookprice) and (author, article-price) pairs and the blocking operators are average, in contrast to nest in Q1. Q2 produces the average prices of books and articles written by an author. This query was modeled after Q1 to study the effect of the size of the result of blocking operators on performance (average returns a small, constant size result compared to the potentially large, variably sized result of nest). The number of books (articles) per author follows a Zipfian distribution.

Query 3 (Q3) is similar to the query in Figure 1 and has a DAG operator graph. The data consists of tuples with car model, dealer, price and color information. The query returns all cars that meet a selection criterion and which are priced less than the average price of cars of the same model. To execute Q3, the car information is replicated: one leg of this information goes to the average and then to the join; the other leg goes through a selection and then to the join. The number of cars for a given model follows a Zipfian distribution.

The parameters varied in the experiments are (a) the Zipfian skew, (b) the Zipfian mean, (c) the number of partial result requests issued during query execution, (d) the number of tuples ((author, book) or (author, article) pairs for Q1, (author, bookprice) or (author, articleprice) pairs for Q2, number of cars for Q3), in the base XML data files and (e) percentage of cars selected (Q3 only). The default parameters are shown in Figure 4. In addition, we explore the case where the input is ordered on the nesting or averaging attribute (author for Q1, Q2 and model for Q3) because it corresponds to some real world scenarios where, for example, each XML file contains information about an author, and because it illustrates the working of the differential algorithm.

Skew of Zipfian Distribution: 1 Mean of Zipfian Distribution: 10 Number of partial result requests: 10 Number of Tuples: 10000 Selectivity (Q3 only): 10%

Figure 4: Default Parameters

5.2 Performance Results

Figure 5 shows a breakdown of the execution time for Q1 using the default parameters. For reference, the graph shows a point for the query evaluation time in the absence of any partial result calculation (No Partial). There were 10 partial result requests, each returning about 9% of the data, and a final request to get the last 9% of the data. The data points show the cumulative time after the completion of each partial result. The overhead of parsing, optimization, etc. is contained in the time for the first partial result.

For the first 45% of the input, the Differential and Re-evaluation algorithms perform similarly. After that point, the differential algorithm is better. In fact, for the complete query, the Differential algorithm reduces the overhead of partial result calculation by over 50%. An interesting observation from the graph above is that if a user issues only a limited number of partial result requests, the Re-evaluation algorithm may be adequate because the extra overhead of the differential algorithm more than offsets the reduction in retransmission.

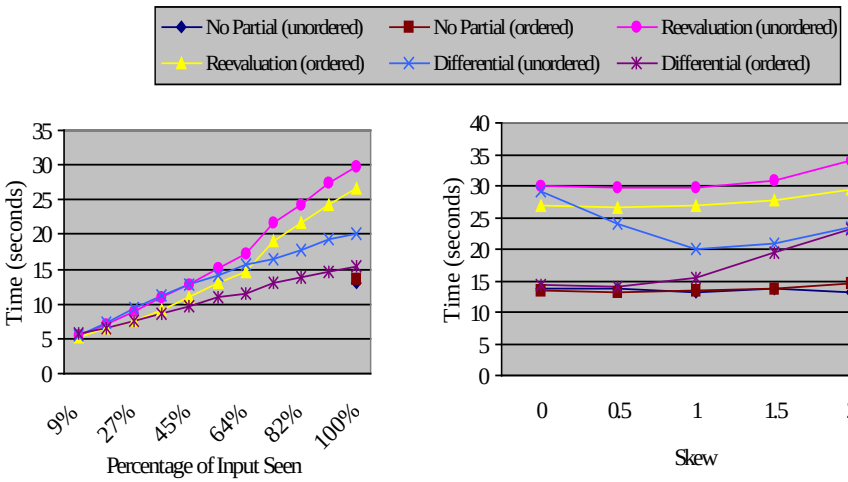

Figure 5: Execution Time for Q1

Figure 6: Effects of Skew on Q1

The difference between Differential (ordered) and No Partial is exactly the overhead of the Differential tuple processing. The 25% difference in total execution time between the ordered and unordered versions of Differential is the overhead caused by tuple retransmission and reprocessing (Differential reduces retransmission, it does not eliminate it.) Finally, though the behavior of Re-evaluation and No Partial is insensitive to order we notice improvement on ordered input, which may be due to processor cache effects.

Figure 6 shows the effect of skew on the different algorithms for Q1. Skew has the effect of changing the size of the groups. The interesting case is the unsorted Differential graph where we see a decrease in execution time followed by an increase. The cost of the Differential algorithm is directly related to the number of tuples that have to be retransmitted. At a skew of 0, there are 1000 groups each with approximately 10 elements. If a group has changed since the last partial result request, the whole group must be retransmitted and reprocessed by the join operator. With a group size of 10 and 10 partial result requests, most groups will change between partial result sets limiting the ability of Differential to reduce retransmission. As skew increases, we see the presence of many very small (2-5 element) groups and a few medium size groups. Very small groups are good for the performance of the Differential algorithm because a group can not be transmitted more times than it has elements. As the skew increases further, the presence of a few very large groups begins to hurt performance. When the skew is 2, there is one group of size approximately 6000. This group changes with almost every partial result request and therefore many elements in this group must be retransmitted many times.

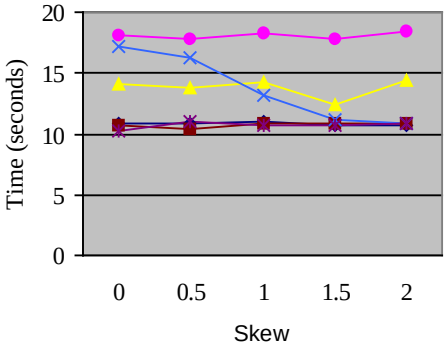

Figure 7: Effects of Skew on Q2

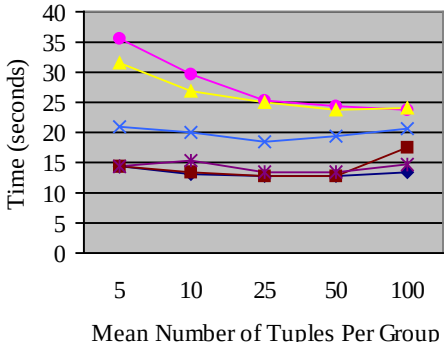

Figure 8: Varying Mean No. of Tuples – Q1

In contrast to Q1, increasing skew for query Q2 (Figure 7) does not adversely affect the performance of the Differential algorithm. At high skews, the partial result for each group is still small for Q2, unlike the large nested values for Q1, and hence has a very low retransmission overhead. This result suggests that finer granularity implementations for large partial results, whereby changes to groups rather than entire groups are retransmitted, can make the Differential algorithm more effective.

Figure 8 shows the affect of changing the mean number of tuples per group (mean of the Zipfian distribution) for Q1. As the mean number of tuples increases, the number of groups decreases since the number of tuples is fixed. The decrease in the number of groups helps the Re-evaluation algorithm because it reduces the size of the join. The Differential algorithm also sees this advantageous affect, but as the mean group size increases, the Differential algorithm suffers because it does more retransmission, as discussed before. Note that when there is only one group, Differential is identical to Re-evaluation and when all groups have size 1, Differential is identical to the case when no partial result requests are issued.

The results from varying the number of tuples (not shown) indicated that the performance of both algorithms scales linearly with the number of tuples. Varying the selectivity for Q3 (not shown) produced predictable results; the Differential approach always performed better than the Re-evaluation approach and the difference increased with decreasing selectivity. We ran experiments with simulated network delays wherein we inserted an exponential delay after every 100 input tuples during query execution. The results (not shown) showed that with increasing delay, the overhead of partial results production decreases. This reduction is due to the overlap between the partial result computation and time spent waiting for data over the slow network.

6 Conclusion and Future Work

Querying the web is creating new challenges in the design and implementation of query engines. A key requirement is the ability to produce partial results that allows users to see results as quickly as possible in spite of low bandwidth, unreliable communication media and long running queries. In this paper, we have identified extensions to the traditional query engine architecture to make this possible. A main extension is the design of operator implementations satisfying the anytime, non-monotonic input/output, maximal output and flexible input properties. Another extension is synchronization packets and partial operators, which are used to ensure the consistency of partial results. Together they form the building blocks for a flexible system that is capable of producing consistent partial results.

Generalizing the operator properties leads to design and implementation challenges. One approach is to stay close to the traditional operator implementation and make as few changes as possible, thus reusing operator code and structure. This choice is embodied in our Re-evaluation approach. Another approach is to design operators to handle the changes intrinsic to the production of partial results. This Differential approach requires more extensive rewrite to the operators, but is more suited to the task of producing partial results. Our quantitative evaluation shows that the Differential approach is successful in reducing partial result production overhead for a wide variety of cases, but also indicates that there are important cases where the Re-evaluation approach works better. In particular, for the cases where the user kills the query after just two or three early partial results, the overhead of the differential approach more than offsets the gain in performance. Another interesting conclusion from the experiments is that the size of the results of blocking operators has a signifi-

cant bearing on the performance of the Differential approach – Differential performs better for "small" aggregate results because the cost of retransmission is less. As expected, the overhead of partial result production reduced with increased communication delays because partial result processing is overlapped with the delays.

There are many threads to follow in the scope of future research. The good performance of the Differential approach suggests that handling changes at granularities finer than tuples is likely to lead to further improvements. Studying fine granularity changes in the context of heavily nested XML structures would be very useful for efficiently monitoring data over the Internet. In terms of providing accuracy and consistency for arbitrary queries, there is the open issue of providing accuracy bounds for general non-monotonic operators. Optimizing the placement of partial operators in the operator graph and generalizing the consistency model to handle weaker and stronger forms of consistency is another area for future investigation.

7 Acknowledgement

Funding for this work was provided by DARPA through NAVY/SPAWAR Contract No. N66001-99-1-8908 and by NSF through NSF award CDA-9623632.

References

1. T. Bray, J. Paoli, C. M. Sperberg-McQueen, "Extensible Markup Language (XML) 1.0", http://www.w3.org/TR/REC-xml.
2. J. Chen, D. DeWitt, F. Tian, Y. Wang, "NiagaraCQ: A Scalable Continuous Query System for Internet Databases," Proceedings of the SIGMOD Conference, Dallas, Texas (2000).
3. J. M. Hellerstein, P. J. Haas, H. Wang, "Online Aggregation", Proceedings of the SIGMOD Conference, Tuscon, Arizona (1997).
4. Z. G. Ives, D. Florescu, M. Friedman, A. Levy, D. S. Weld, "An Adaptive Query Execution System for Data Integration", Proceedings of the SIGMOD Conference, Philadelphia, Pennsylvania (1999).
5. Z. G. Ives, A. Y. Levy, D. S. Weld. Efficient Evaluation of Regular Path Expressions on Streaming XML Data. Technical Report UW-CSE-2000-05-02, University of Washington.
6. L. Liu, C. Pu, R. Barga, T. Zhou, "Differential Evaluation of Continual Queries", Proceedings of the International Conference on Distributed Computing Systems (1996).
7. K. Tan, C. H. Goh, B. C. Ooi, "Online Feedback for Nested Aggregate Queries with Multi-Threading", Proceedings of the VLDB Conference, Edinburgh, Scotland (1999).
8. T. Urhan, M. J. Franklin, "XJoin: Getting Fast Answers from Slow and Bursty Networks", University of Maryland Technical Report, UMIACS-TR-99-13 (1999).
9. T. Urhan, M. J. Franklin, L. Amsaleg, "Cost Based Query Scrambling for Initial Delays", Proceedings of the SIGMOD Conference, Seattle, Washington (1998).
10. A. N. Wilschut, P. M. G. Apers, "Data Flow Query Execution in a Parallel Main Memory Environment", International Conference on Parallel and Distributed Information Systems (1991).