Architecting a Network Query Engine for Producing Partial Results

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

Jeffrey Naughton[1] David Maier[3]

jai@cs.wisc.edu, tufte@cse.ogi.edu, dewitt@cs.wisc.edu, naughton@cs.wisc.edu, maier@cse.ogi.edu

[1]Department of Computer Sciences
University of Wisconsin
Madison, WI 53706

[2]IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120

[3]Department of Computer Science
Oregon Graduate Institute
Portland, OR 97291

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, there are several changes that need to be made to existing query engines to make them applicable for the task of querying the Internet. One of the challenges is providing partial results of query computation, based on the initial part of the input, because it may be undesirable to wait for all of the input. This 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 blocking operators, such as max, average, negation and nest. While previous work on producing partial results has looked at a limited set of blocking 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 blocking operators. Re-architecting a query engine to produce partial results requires modifications to the runtime operators. We explore implementation alternatives and quantitatively compare their performance using our prototype system.

Keywords

Partial results, blocking operators, query processing, XML, Internet.

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 key word searches. An important and challenging 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/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]).

The main challenge in producing partial results lies in dealing with blocking operators, such as average, sum, nest and negation since these operators need to see all of their input before they produce the correct output. Previous solutions to the problem of producing partial results present solutions for specific aggregate operators [3][7] and thus do not extend to new blocking operators such as nest and negation that are becoming increasingly important for network query engines. Further, the previous solutions do not allow blocking operators to appear deep in a user query. Thus, for example, 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, cannot be handled by previous techniques (nest is blocking and appears deep in the query tree). Neither can they handle a query that constantly monitors the average price of BMW cars posted in the Internet except those that appear on salvage lists (average and except are blocking).

The Niagara Internet Query System [6] contains a general framework for producing partial results for queries involving blocking operators. The framework allows blocking and non-blocking operators to be arbitrarily intermixed in the query tree, i.e., non-blocking operators can operate on the results of blocking operators and vice-versa. However, the framework imposes certain key requirements on the implementations of both blocking and non-blocking operators. In this paper, we identify alternative algorithms and implementations satisfying the key requirements and evaluate their performance using the Niagara system. This paper complements the architectural overview in [6] and justifies the implementation decision made therein.

The rest of the paper is organized as follows. In Section 2, we formally define partial results and list the properties operator implementations need to satisfy in order to produce partial results. In Section 3, we identify alternative operator implementation techniques and discuss the issue of accuracy of partial results. The performance results are contained in Section 4 and Section 5 concludes the paper.

2. DESIRED OPERATOR PROPERTIES FOR PRODUCING PARTIAL RESULTS

In the previous section, we illustrated the need for partial results involving arbitrary blocking operators. We now identify some key properties of operator implementations, not supported by traditional query engine architectures, which are crucial for producing partial results. We start by defining the term "partial result."

Definition: Let Q be a query with n inputs and let $Q(I_1, \ldots, I_n)$ represent the result of query Q on inputs $I_1, \ldots, I_n$. A *partial result* of the query Q on inputs $I_1,\ldots,I_n$ is $Q(PI_1, \ldots, PI_n)$, where for $1 \le j \le n$, $PI_j \subseteq I_j$.

Intuitively, a partial result of a query on a set of inputs is the result of the query on a (possibly) different set of inputs such that each input in the new set is a sub-set of the corresponding input in the old set.

We now turn to the notions of "non-blocking" and "blocking" operators. Intuitively, a "non-blocking" operator is one that produces the same output for a given input, regardless of whether there are further inputs; i.e., it does not block waiting to see all of its inputs. Thus, select, project, join, intersect and distinct (duplicate elimination) are non-blocking operators. Operators that are not non-blocking are "blocking" i.e., the output for a given input depends on further inputs. Thus sort, nest, average and outer-join operators are blocking. Some operators such as "except" are blocking on a sub-set of their inputs. Consider the example A.a except B. The "except" operator will block until all of B is received, at which point it becomes a non-blocking operator.

The properties listed below summarize the key requirements that blocking and non-blocking operators must satisfy to produce partial results. Traditional operator implementations are not suitable for partial result production because they do not have all of these characteristics. (See [6] for more details.)

1) **Flexible Input Property:** Operators should not stall waiting for input from a particular input stream if there is some input available on another input stream. This is necessary in order to be able to provide partial results without stalling on a slow input stream.

2) **Maximal Output Property:** Operators should produce results as soon as possible. That is, the operator should output as much of the result as it can without potentially giving a wrong answer. Note that this property is desirable even for blocking operators. For example, the outer-join operator can produce (the joining) results before the end of its inputs.

3) **Non-Monotonic Input-Output Property:** Each operator has to deal with input streams (and produce output streams) that are not monotonically increasing. This is a direct result of requiring blocking operators such as nest, except and average, to produce partial results.

4) **Anytime Property:** At any time, blocking operators should be able to output the "current" result, based on the data seen so far on its input stream(s). This enables the system to provide a partial result whenever the user requests one. Note that the Maximal Output property implies the Anytime property for non-blocking operators.

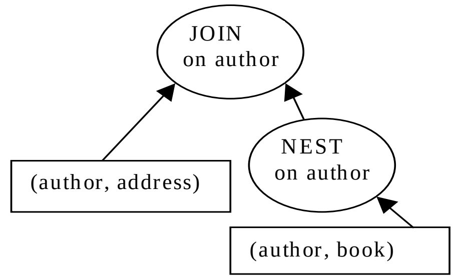


Figure 1: Operator Tree Example

The design of operator implementations satisfying the properties above is crucial in designing a flexible system capable of producing partial results. We turn to this issue next.

3. OPERATOR IMPLEMENTATION ALTERNATIVES

We now explore two alternatives (Re-evaluation and Differential) for modifying existing operator implementations so that they satisfy the desired properties for producing partial results. The Re-evaluation approach retains the structure of existing operator implementations but requires the re-execution of all parts of the query above the blocking operators. The Differential alternative processes changes as part of the operator implementation, similar to the technique used in the CQ project [5], and avoids re-execution. There is 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 tuple structure changes, but are likely to be more efficient.

The Re-evaluation and Differential approaches are similar in that they both use non-blocking, flexible input, maximal output implementations for operators wherever possible. For example, joins are implemented using symmetric hash join [9] and symmetric nested loops join algorithms (or their variants [4][8]). The algorithms in this section extend such flexible input, maximal output, non-blocking operator implementations to satisfy the non-monotonic input/output property and further, identify blocking operator implementations satisfying all four desirable properties.

3.1 Re-evaluation Algorithm

As mentioned before, we must determine what form partial results produced by blocking operators take and how updates to those results are communicated. The Re-evaluation Algorithm handles this in a straightforward manner by having blocking operators simply 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 operators above the blocking operator must re-evaluate the query each time a partial result request is issued; hence the name Re-evaluation Algorithm.

Consider the operator tree in Figure 1 which shows a nest operator reading (author, book) pairs from an XML file on disk (or any non-blocking operator), nesting the pairs on author and sending its output to a join operator. The nest is blocking; the join is non-blocking. Upon receipt of a partial result request, the nest operator transmits all (author, <set of books>) groups it has created so far to the join. At this point, the join 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. Below we describe the re-evaluation implementations of join and nest. Descriptions of other operator implementations are omitted in the interest of space.

Re-evaluation Join: Re-evaluation Join functions identically to a symmetric hash join except that when Re-evaluation Join is notified that a new partial result set is beginning on a particular input stream, it clears the hash table associated with that input. 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. This can occur if the input comes from a union operator, which unions the output of a blocking and non-blocking operator.

Re-evaluation Nest: Similar to a traditional hash-based nest, Re-evaluation Nest creates a hash table entry for each distinct value of the grouping attribute (author in our example). When a start partial result notification is received, Re-evaluation Nest acts lazily and does not delete the hash table. Instead, Re-evaluation Nest 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.

3.2 Differential Algorithm

The Re-evaluation algorithm 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 in [5] in the context of continual queries. Our system, however, handles changes as the query is being executed as opposed to [5], which proposes a model for periodic re-execution of queries. This gives rise to new techniques for handling changes as the operator is in progress.

In Figure 1, in order for the join to process differences between sets of partial results, the nest operator must produce the "difference" and the join must be able to process that "difference." We accomplish this by having all operators produce and consume tuples that consist of the old tuple value and the new tuple value, as in [5]. Since the partial results produced by blocking operators consist of differences from previously propagated results, each tuple produced by a blocking operator is an insert, delete or update. In the interest of space, we describe only the differential join and nest algorithms below.

Differential Join: Differential Join is based on symmetric hash join. A Differential Join 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 in the tree. 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: Differential Nest is similar to hash-based nest. Inserts are treated just as tuples are in a traditional nest operator. For deletes, Differential Nest 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 and 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.

3.3 Accuracy of Partial Results

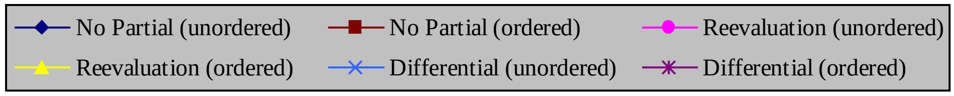
In the above sections, we have concentrated on operator implementations that produce partial results. An important concern is the accuracy of these results. We believe that our framework is general enough to accommodate various techniques for computing the accuracy of partial results, such as those proposed 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, unlike [3][7], our framework allows blocking operators (such as aggregates) to appear anywhere in the query tree. It is also important to address accuracy of partial results for non-numeric blocking operators such as nest and except. This is more difficult because notions such as "average" and "confidence intervals" are not well defined in these domains. It is, however, possible to provide the user with statistics such as the percentage of XML files (received and) processed and/or the geographical locations of the processed files. The user may well be able to use this information to understand the partial result.

4. PERFORMANCE EVALUATION

In the previous section, we outlined the Re-evaluation and Differential implementation alternatives for operators. In this section, we quantitatively compare the performance of the two approaches. We begin by describing the experimental set up in Section 4.1. Section 4.2 describes the performance results.

4.1 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 XML data being processed is resident in main memory. Though we expect this to be acceptable for many cases given current large main memory



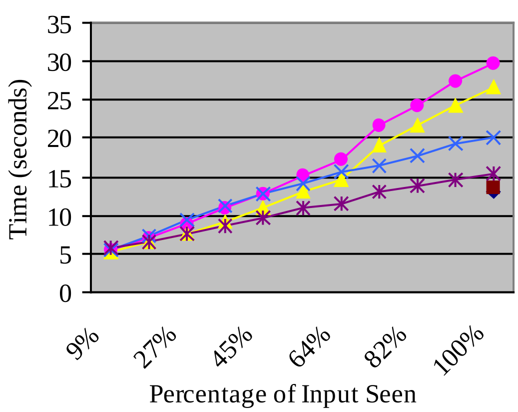


Figure 2: Breakdown of Execution Time for Q1

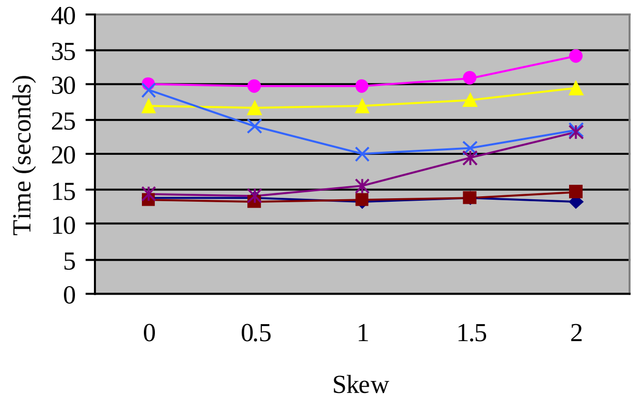


Figure 3: Effects of Skew on Q1

sizes, we also plan to explore more flexible implementations that handle spillovers to disk as part of future work.

To evaluate the performance of the Re-evaluation and Differential approaches, we used two queries that allowed us to study the overhead of partial result production and identify the strengths and weaknesses of each approach. The first query (Q1) contains a join over two blocking operators. The input is two XML documents, one containing flat (author, book) pairs and the other containing 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. Data was generated so that the number of books (articles) of an author follows a Zipfian distribution.

The second query (Q2) is similar to Q1 except that the inputs are (author, book-price) and (author, article-price) pairs and the blocking (aggregate) operators are average, in contrast to nest in Q1. Q2 produces the average prices of books and articles written by an author. A significant difference between Q1 and Q2 is that the aggregate in Q2 (average) returns a small, constant size result compared to the potentially large variable size result of the aggregate (nest) in Q1. The number of book (article) prices per author follows a Zipfian distribution.

Default Parameters
Skew of Zipfian Distribution: 1
Mean of Zipfian Distribution: 10
Number of partial result requests: 10
Number of Tuples: 10000

The parameters varied in the experiments along with their default values are shown above. Note that “Number of Tuples” refers to the number of ((author, book) or (author, article) pairs) for Q1, ((author, book-price) or (author, article-price) pairs) for Q2, in the base XML data files. In addition, we explore the case where the input is ordered on the author attribute because it corresponds to some real world scenarios where, for example, each XML file contains information about an author and also because it illustrates the working of the differential algorithm.

4.2 Performance Results

Figure 2 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 above graph and from results obtained by varying the total number of partial results (not shown), is that if a user issues only a limited number of partial result requests, the Re-evaluation algorithm may be adequate. This is because the extra overhead of the differential algorithm more than offsets the reduction in retransmission.

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. This may be due to processor cache effects.

Figure 3 shows the effect of skew on the different algorithms for Q1. Skew has the effect of changing the size of the

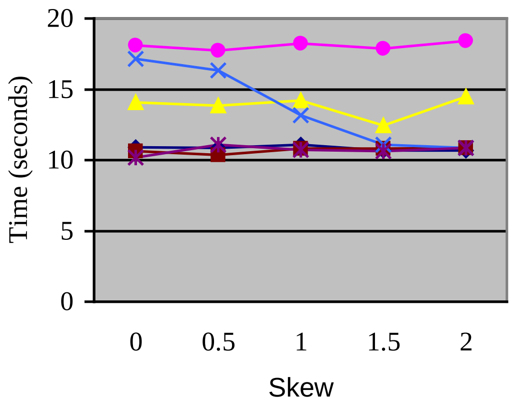


Figure 4: Effects of Skew on Q2

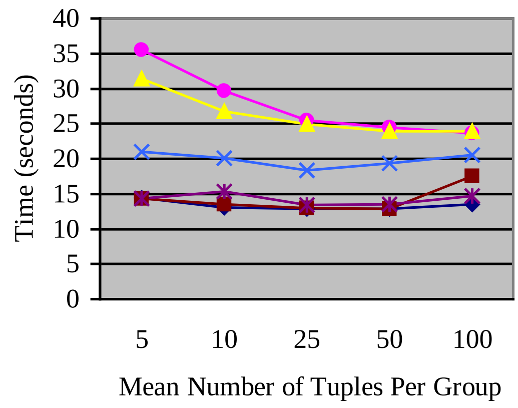


Figure 5: Q1 – Changing Mean No. of Tuples

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.

In contrast to Q1, increasing skew for query Q2 (Figure 4) does not adversely affect the performance of the Differential algorithm. This is because 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 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 5 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 on varying the number of tuples were not very surprising – the performance of both algorithms scales linearly with the number of tuples. We also ran experiments with simulated network delays. In these experiments, we inserted an exponential delay after every 100 input tuples during query execution. The results of these experiments (not shown) showed that with increasing delay, the overhead of partial results production reduces. This is because the partial result computation time is overlapped with the time spent waiting for data over the slow network.

5. CONCLUSION AND FUTURE WORK

Querying the web is creating new challenges in the design and implementation of query engines. A core 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 mediums and long running queries. In this paper, we have identified extensions to the traditional query engine architecture to make this possible and explored the effectiveness of alternative implementation strategies (Re-evaluation and Differential). Our quantitative evaluation shows that the Differential algorithm 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 significant bearing on the performance of the Differential Algorithm – Differential performs better for "small" aggregate results because the cost of retransmission is less.

There are many possible directions for 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 this in the context of heavily nested XML structures would be very useful for efficiently monitoring data over the Internet. Another interesting challenge lies in providing accuracy bounds for general blocking operators.

6. ACKNOWLEDGEMENTS

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.

7. 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, et. al., “NIAGARACQ: Continuous Queries,” Proceedings of the 2000 ACM SIGMOD Conference, Dallas, TX, May 2000 (to appear).

[3] J. M. Hellerstein, P. J. Haas, H. Wang, “Online Aggregation”, Proceedings of the 1997 ACM SIGMOD Conference, Tuscon, AZ, May 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 1999 ACM-SIGMOD Conference, Philadelphia, PA, June 1999.

[5] L. Liu, C. Pu, R. Barga, T. Zhou, “Differential Evaluation of Continual Queries”, Proceedings of the International Conference on Distributed Computing Systems, 1996.

[6] J. Naughton, et. al., “The Niagara Internet Query System”, submitted for publication.

[7] K. Tan, C. H. Goh, B. C. Ooi, “Online Feedback for Nested Aggregate Queries with Multi-Threading”, Proceedings of the 1999 VLDB Conference, Edinburgh, Scotland, September 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] 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.