NiagaraCQ: A Scalable Continuous Query System for Internet Databases

Jianjun Chen David J. DeWitt Feng Tian Yuan Wang
Computer Sciences Department
University of Wisconsin-Madison

{jchen, dewitt, ftian, yuanwang}@cs.wisc.edu

ABSTRACT

Continuous queries are persistent queries that allow users to receive new results when they become available. While continuous query systems can transform a passive web into an active environment, they need to be able to support millions of queries due to the scale of the Internet. No existing systems have achieved this level of scalability. NiagaraCQ addresses this problem by grouping continuous queries based on the observation that many web queries share similar structures. Grouped queries can share the common computation, tend to fit in memory and can reduce the I/O cost significantly. Furthermore, grouping on selection predicates can eliminate a large number of unnecessary query invocations. Our grouping technique is distinguished from previous group optimization approaches in the following ways. First, we use an incremental group optimization strategy with dynamic re-grouping. New queries are added to existing query groups, without having to regroup already installed queries. Second, we use a query-split scheme that requires minimal changes to a general-purpose query engine. Third, NiagaraCQ groups both change-based and timer-based queries in a uniform way. To insure that NiagaraCQ is scalable, we have also employed other techniques including incremental evaluation of continuous queries, use of both pull and push models for detecting heterogeneous data source changes, and memory caching. This paper presents the design of NiagaraCQ system and gives some experimental results on the system's performance and scalability.

1. INTRODUCTION

Continuous queries [TGNO92][LPT99][LPBZ96] allow users to obtain new results from a database without having to issue the same query repeatedly. Continuous queries are especially useful in an environment like the Internet comprised of large amounts of frequently changing information. For example, users might want to issue continuous queries of the form:

> *Notify me whenever the price of Dell or Micron stock drops by more than 5% and the price of Intel stock remains unchanged over next three month.*

In order to handle a large number of users with diverse interests, a continuous query system must be capable of supporting a large number of triggers expressed as complex queries against web-resident data sets.

The goal of the Niagara project is to develop a distributed database system for querying distributed XML data sets using a query language like XML-QL [DFF+98]. As part of this effort, our goal is to allow a very large number of users to be able to register continuous queries in a high-level query language such as XML-QL. We hypothesize that many queries will tend to be similar to one another and hope to be able to handle millions of continuous queries by grouping similar queries together. Group optimization has the following benefits. First, grouped queries can share computation. Second, the common execution plans of grouped queries can reside in memory, significantly saving on I/O costs compared to executing each query separately. Third, grouping makes it possible to test the "firing" conditions of many continuous queries together, avoiding unnecessary invocations.

Previous group optimization efforts [CM86] [RC88] [Sel86] have focused on finding an optimal plan for a small number of similar queries. This approach is not applicable to a continuous query system for the following reasons. First, it is computationally too expensive to handle a large number of queries. Second, it was not designed for an environment like the web, in which continuous queries are dynamically added and removed. Our approach uses a novel incremental group optimization approach in which queries are grouped according to their signatures. When a new query arrives, the existing groups are considered as possible optimization choices instead of re-grouping all the queries in the system. The new query is merged into existing groups whose signatures match that of the query.

Our incremental group optimization scheme employs a query-split scheme. After the signature of a new query is matched, the sub-plan corresponding to the signature is replaced with a scan of the output file produced by the matching group. This optimization process then continues with the remainder of the query tree in a bottom-up fashion until the entire query has been analyzed. In the case that no group "matches" a signature of the new query, a new query group for this signature is created in the

Permission to make digital or hard copies of part or all of this work or personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
MOD 2000, Dallas, TX USA
© ACM 2000 1-58113-218-2/00/05 . . .$5.00

system. Thus, each continuous query is split into several smaller queries such that inputs of each of these queries are monitored using the same techniques that are used for the inputs of user-defined continuous queries. The main advantage of this approach is that it can be implemented using a general query engine with only minor modifications. Another advantage is that the approach is easy to implement and, as we will demonstrate in Section 4, very scalable.

Since queries are continuously being added and removed from groups, over time the quality of the group can deteriorate, leading to a reduction in the overall performance of the system. In this case, one or more groups may require "dynamic re-grouping" to re-establish their effectiveness.

Continuous queries can be classified into two categories depending on the criteria used to trigger their execution. *Change-based* continuous queries are fired as soon as new relevant data becomes available. *Timer-based* continuous queries are executed only at time intervals specified by the submitting user. In our previous example, day traders would probably want to know the desired price information immediately, while longer-term investors may be satisfied being notified every hour. Although change-based continuous queries obviously provide better response time, they waste system resources when instantaneous answers are not really required. Since timer-based continuous queries can be supported more efficiently, query systems that support timer-based continuous queries should be much more scalable. However, since users can specify various overlapping time intervals for their continuous queries, grouping timer-based queries is much more difficult than grouping purely change-based queries. Our approach handles both types of queries uniformly.

NiagaraCQ is the continuous query sub-system of the Niagara project, which is a net data management system being developed at University of Wisconsin and Oregon Graduate Institute. NiagaraCQ supports scalable continuous query processing over multiple, distributed XML files by deploying the incremental group optimization ideas introduced above. A number of other techniques are used to make NiagaraCQ scalable and efficient. 1) NiagaraCQ supports the incremental evaluation of continuous queries by considering only the changed portion of each updated XML file and not the entire file. Since frequently only a small portion of each file gets updated, this strategy can save significant amounts of computation. Another advantage of incremental evaluation is that repetitive evaluation is avoided and only new results are returned to users. 2) NiagaraCQ can monitor and detect data source changes using both push and poll models on heterogeneous sources. 3) Due to the scale of the system, all the information of the continuous queries and temporary results cannot be held in memory. A caching mechanism is used to obtain good performance with limited amounts of memory.

The rest of the paper is organized as follows. In Section 2 the NiagaraCQ command language is briefly described. Our new group optimization approach is presented in Section 3 and its implementation is described in Section 4. Section 5 examines the performance of the incremental continuous query optimization scheme. Related work is described in Section 6. We conclude our paper in Section 7.

2. NIAGARACQ COMMAND LANGUAGE

NiagaraCQ defines a simple command language for creating and dropping continuous queries. The command to create a continuous query has the following form:

CREATE *CQ_name*
XML-QL query
DO *action*
{**START** *start_time*} {**EVERY** *time_interval*} {**EXPIRE** *expiration_time*}

To delete a continuous query, the following command is used:

Delete *CQ_name*

Users can write continuous queries in NiagaraCQ by combining an ordinary XML-QL query with additional time information. The query will become effective at the *start_time*. The *Time_interval* indicates how often the query is to be executed. A query is timer-based if its *time_interval* is not zero; otherwise, it is change-based. Continuous queries will be deleted from the system automatically after their *expiration_time*. If not provided, default values for the time are used. (These values can be set by the database administrator.) *Action* is performed upon the XML-QL query results. For example, it could be ``MailTo dewitt@cs.wisc.edu" or a complex stored procedure to further processing the results of the query. Users can delete installed queries explicitly using the delete command.

3. OUR INCREMENTAL GROUP OPTIMIZATION APPROACH

In Section 3.1, we present a novel incremental group optimization strategy that scales to a large number of queries. This strategy can be applied to a wide range of group optimization methods. A specific group optimization method based on this approach is described in Section 3.2. Section 3.3 introduces our query-split scheme that requires minimal changes to a general-purpose query engine. Section 3.4 and 3.5 apply our group optimization method to selection and join operators. We discuss how our system supports timer-based queries in Section 3.6. Section 3.7 contains a brief discussion of the caching mechanisms in NiagaraCQ to make the system more scalable.

3.1 General Strategy of Incremental Group Optimization

Previous group optimization strategies [CM86] [RC88] [Sel86] focused on finding an optimal global plan for a small number of queries. These techniques are useful in a query environment where a small number of similar queries either enter the system within a short time interval or are given in advance. A naive approach for grouping continuous queries would be to apply these methods directly by reoptimizing all queries whenever a new query is added. We contend that such an approach is not acceptable for large dynamic environments because of the associated performance overhead.

We propose an incremental group optimization strategy for continuous queries in this paper. Groups are created for existing queries according to their signatures, which represent similar structures among the queries. Groups allow the common parts of

two or more queries to be shared. Each individual query in a query group shares the results from the execution of the group plan. When a new query is submitted, the group optimizer considers existing groups as potential optimization choices. The new query is merged into those existing groups that match its signatures. Existing queries are not, however, re-grouped in our approach. While this strategy is likely to result in sub-optimal groups, it reduces the cost of group optimization significantly. More importantly it is very scalable in a dynamic environment. Since continuous queries are frequently added and removed, it is possible that current groups may become inefficient. "Dynamic re-grouping" would be helpful to re-group part or all of the queries either periodically or when the system performance degrades below some threshold. This is left as future work.

3.2 Incremental Group Optimization using Expression Signature

Based on our incremental grouping strategy, we designed a scalable group optimization method using expression signatures. Expression signatures [HCH+99] represent the same syntax structure, but possibly different constant values, in different queries. It is a specific implementation of the signature concept.

3.2.1 Expression Signature

For purposes of illustration, we use XML-QL queries on a database of stock quotes.

```
Where <Quotes> <Quote>
            <Symbol>INTC</>
        </> </> element_as $g
in "http://www.cs.wisc.edu/db/quotes.xml"
construct $g

Where <Quotes> <Quote>
            <Symbol>MSFT</>
        </> </> element_as $g
in "http://www.cs.wisc.edu/db/quotes.xml"
construct $g
```

Figure 3.1 XML-QL query examples

The two XML-QL queries in Figure 3.1 retrieve stock information on either Intel (symbol INTC) or Microsoft (symbol MSFT). Many users are likely to submit similar queries for different stock symbols. An expression signature is created for the selection predicates by replacing the constants appearing in the predicates with a placeholder. The expression signature for the two queries in Figure 3.1 is shown in Figure 3.2.

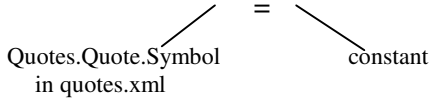


Figure 3.2 Expression signature of queries in Figure 3.1

A query plan is generated by Niagara query parser. Figure 3.3 shows the query plans of the queries in Figure 3.1. The lower part in each query plan corresponds to the expression signature of the queries. A new operator TriggerAction is added on the top of the XML-QL query plan after the query is parsed. Expression signatures allow queries with the same syntactic structure to be grouped together to share computation [HCH+99]. Expression signatures for different queries will be discussed later. Note, in NiagaraCQ, users can specify an XML-QL query without specifying the destination data sources by using a "*" in the file name position and giving a DTD name. This allows users to specify continuous queries without naming the data sources. Our group query optimizer is easily extended to support this capability by using a mapping mechanism offered by the Niagara Search Engine. Without losing generality for our incremental grouping algorithm, we assume continuous queries are defined on a specific data source in this paper.

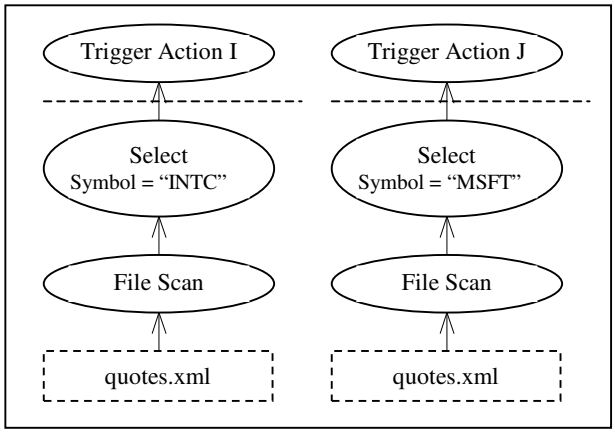


Figure 3.3 Query plans of queries in Figure 3.1

3.2.2 Group

Groups are created for queries based on their expression signatures. For example, a group is generated for the queries in Figure 3.1 because they have same expression signature. We use this group in following discussion. A group consists of three parts.

1. Group signature

The *group signature* is the common expression signature of all queries in the group. For the example above, the expression signature is given in Figure 3.2.

Constant_value	Destination_buffer
….	….
INTC	Dest. i
MSFT	Dest. j
….	….

Figure 3.4 an example of group constant table

2. Group constant table

The *group constant table* contains the signature constants of all queries in the group. The constant table is stored as an XML file. For the example above, "INTC" and "MFST" are stored in this table (Figure 3.4). Since the tuples produced by the shared computation need to be directed to the correct individual query for further processing, the destination information is also stored with the constant.

3. Group plan

The *group plan* is the query plan shared by all queries in the group. It is derived from the common part of all single query plans in the group. Figure 3.5 shows the group plan for the queries in Figure 3.1.

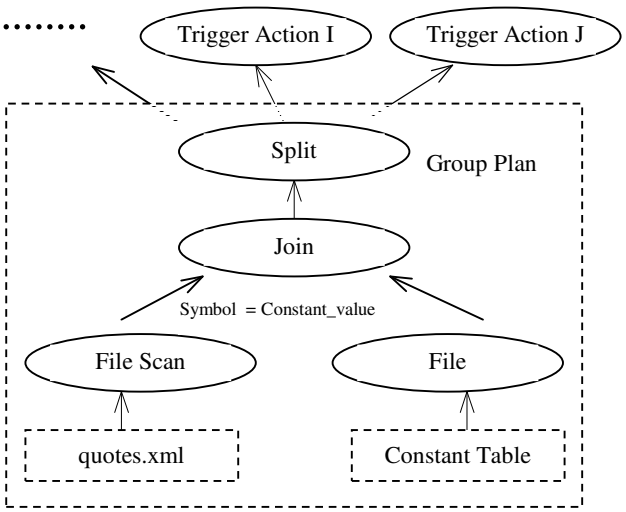


Figure 3.5 Group plan for queries in Figure 3.1

An expression signature allows queries in a group to have different constants. Since the result of the shared computation contains results for all the queries in the group, the results must be filtered and sent to the correct destination operator for further processing. NiagaraCQ performs filtering by combining a special ***Split*** operator with a Join operator based on the constant values stored in the constant table. Tuples from the data source (e.g. Quotes.xml) are joined with the constant table. The *Split* operator distributes each result tuple of the Join operator to its correct destination based on the destination buffer name in the tuple (obtained from the Constant Table). The *Split* operator removes the name of the destination buffer from the tuple before it is put into the output stream, so that subsequent operators in the query do not need to be modified. In addition, queries with the same constant value also share the same output stream. This feature can significantly reduce the number of output buffers.

Since generally the number of active groups is likely to be on the order of thousands or ten of thousands, group plans can be stored in a memory-resident hash table (termed a *group table*) with the group signature as the hash key. Group constant tables are likely to be large and are stored on disk.

3.2.3 Incremental Grouping Algorithm

In this section we briefly describe how the NiagaraCQ group optimizer performs incremental group optimization.

When a new query (Figure 3.6) is submitted, the group optimizer traverses its query plan bottom up and tries to match its expression signature with the signatures of existing groups. The expression signature of the new query, which is the same as the signature in Figure 3.2, matches the signature of the group in Figure 3.5. The group optimizer breaks the query plan (Figure 3.7) into two parts. The lower part of the query is removed. The upper part of the query is added onto the group plan. If the constant table does not have an entry "AOL", it will be added and a new destination buffer allocated.

```
Where  <Quotes>
<Quote>
    <Symbol>AOL</>
     </></>
   element_as $g
in
"http://www.cs.wisc.edu/
db/quotes.xml"
construct $g
```

Figure 3.6 XML-QL query examples

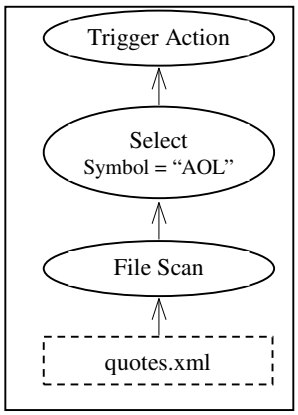


Figure 3.7 Query plan for query in Figure 3.6

In the case that the signature of the query does not match any group signature, a new group will be generated for this signature and added to the group table.

In general, a query may have several signatures and may be merged into several groups in the system. This matching process will continue on the remainder of the query plan until the top of the plan is reached. Our incremental grouping is very efficient because it only requires one traversal of the query plan.

In the following sections, we first discuss our *query-split* scheme and then describe how incremental group optimization is performed on selection and join operators.

3.3 Query Split with Materialized Intermediate Files

The destination buffer for the split operator can be implemented either in a pipelined scheme or as an intermediate file. Our initial design of the split operator used a pipeline scheme in which tuples are pipelined from the output of one operator into the input of the next operator. However, such a pipeline scheme does not work for grouping timer-based continuous queries. Since timer-based queries will only be fired at specified time, output tuples must be retained until the next firing time. It is difficult for a split operator to determine which tuples should be stored and how long they should be stored for.

In addition, in the pipelined approach, the ungrouped parts of all query plans in a group are combined with the group plan, resulting in a single execution plan for all queries in the group. This single plan has several disadvantages. First, its structure is a directed graph, and not a tree. Thus, the plan may be too complicated for a general-purpose XML-QL query engine to execute. Second, the combined plan may be very large and require resources beyond the limits of some systems. Finally, a large portion of the query plan may not need to be executed at each query invocation. For example, in Figure 3.5, suppose only the price of Intel stock changes. Although the destination buffer for Microsoft is empty, the upper part of the Microsoft query (Trigger Action J) is also executed. This problem can be avoided only if the execution engine has the ability to selectively

load part of a query plan in a bottom-up manner. Such a capability would require a special implementation of the XML-QL query engine.

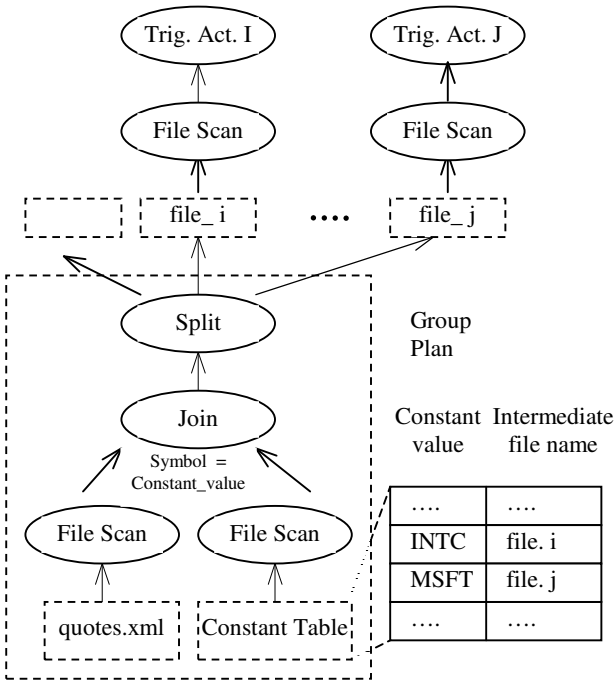


Figure 3.8 query-split scheme using intermediate files

Since a split operator has one input stream and multiple (possibly tens of thousands) output streams, split operators may become a bottleneck when the ungrouped parts of queries consume output tuples from the split stream at widely varying rates. For example, suppose 100 queries are grouped together, 99 of which are very simple selection queries, and one is a very expensive query involving multiple joins. Since this expensive query may process the input from the split operator very slowly, it may block all the other simple queries.

The pipeline scheme can be used in systems that support only a small number of change-based continuous queries. Since our goal is to support millions of both change-based and timer-based continuous queries, we adopt an approach that is more scalable and easier to implement. We also try to use a general query engine to the maximal extent possible.

In our new design (Figure 3.8), the split operator writes each output stream into an intermediate file. A query plan is cut into two parts at the split operator and a file scan operator is added to the upper part of plan to read the intermediate file. NiagaraCQ treats the two new queries like normal user queries**.** In particular, changes to the intermediate files are monitored in the same way as those to ordinary data sources! Since a new continuous query may overlap with multiple query groups, one query may be split into several queries. However, the total number of queries in the system will not exceed the number of groups plus the number of original user queries. Since we assume that no more than thousands of groups will be generated for millions of user queries, the overall number of queries in the system will increase only slightly. Intermediate file names are stored in the constant table and grouped continuous queries with the same constant share the same intermediate file.

The advantages of this new design include:

1. Each query is scheduled independently, thus only the necessary queries are executed. For example, in Figure 3.8, if only the price of Intel stock changes, queries on intermediate files other than "file_i" will not be scheduled. Since usually only a small amount of data is changed, only a few of the installed continuous queries will be fired. Thus, computation time and system resource usage is significantly reduced.

2. Queries after a split operator will be in a standard, tree-structured query format and thus can be scheduled and executed by a general query engine.

3. Each query in the system is about the size of a common user query, so that it can be executed without consuming an unusual amount of system resources.

4. This approach handles intermediate files and original data source files uniformly. Changes to materialized intermediate files will be processed and monitored just like changes to the original data files.

5. The potential bottleneck problem of the pipelined approach is avoided.

There are some potential disadvantages. First, the split operator becomes a blocking operator since the execution of the upper part of the query must wait for the intermediate files to be completely materialized. Since continuous queries run over data changes that are usually not very large, we do not believe that the impact of this blocking will be significant. Second, reading and writing the intermediate files incurs extra disk I/Os. Since most data changes will be relatively small, we anticipate that they will be buffered in memory before the upper part queries consume them. There will be disk I/Os in the case of timer-based queries that have long time intervals because data changes may be accumulated. In this situation, data changes need to be written to disk no matter what strategy is used. As discussed in Section 3.7, NiagaraCQ uses special caching mechanisms to reduce this cost.

3.4 Incremental Grouping of General Selection Predicates

Our primary focus is on predicates that are in the format of "*Attribute op Constant*." *Attribute* is a path expression without wildcards in it. Op includes "=", "<", ">". Such formats dominate in selection queries. Other predicate formats could also be handled in our approach, but we do not discuss them further in this paper.

Figure 3.9 shows an example of a range selection query that returns every stock whose price has risen more than 5%. Figure 3.9 also gives its expression signature. The group plan for queries with this signature is the same in Figure 3.5, except the join condition is *Change_Ratio > constant*.

A general range-query has both lower_bound and upper_bound values. Two columns are needed to represent both bounds in the constant table. Thus each entry of the constant table will be [lower_bound, upper_bound, intermediate_file_name]. The join condition is *Change_Ratio < upper_bound and Change_Ratio > lower_bound*. A special index would be helpful to evaluate this predicate. For example, an interval skip list [HJ94] could be used for this purpose when all the intervals fit in memory. We

are considering developing a new index method that handles this case more efficiently.

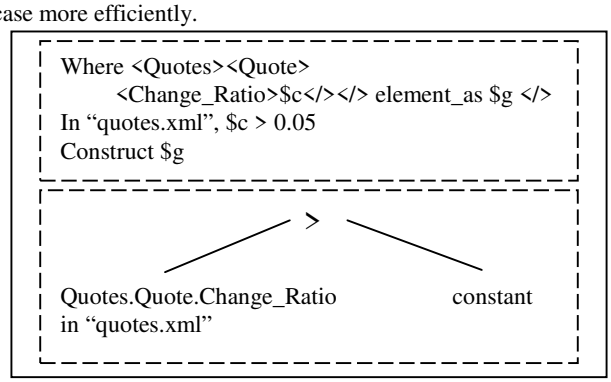


Figure 3.9 Range selection query example and its expression signature

One potential problem for range-query groups is that the intermediate files may contain a large number of duplicate tuples because range predicates of the different queries might overlap. "Virtual intermediate files" are used to handle this case. Each virtual intermediate file stores a value range instead of actual result tuples. All outputs from the split operator are stored in one real intermediate file, which has a clustered index on the range attribute. Modification on virtual intermediate files can trigger upper-level queries in the same way as ordinary intermediate files. The value range of a virtual intermediate file is used to retrieve data from the real intermediate file. Our query-split scheme need not be changed to handle virtual intermediate files.

In general, a query may have multiple selection predicates, i.e. multiple expression signatures. Predicates on the same data source can be represented in conjunctive normal form. The group optimizer chooses the most selective conjunct, which does not contain "or", to do incremental grouping. Other predicates are evaluated in the upper levels of the continuous query after the split operator.

```
Where <Quotes><Quote><Symbol>"INTC"</>
        <Current_Price>$p</></>  element_as $g </>
in "quotes.xml", $p < 100
Construct $g
```

Figure 3.10 an example query with two selection predicates

Figure 3.10 shows a query with two selection predicates, which retrieves Intel stock whenever its price falls below $100. This query has two expression signatures, one is an equal selection predicate on *Symbol* and the other is a range selection predicate on *Current_price*. The expression signature on the equal selection predicate (i.e. on *Symbol*) is used for grouping because it is more selective. In addition, a new select operator with the second selection predicate (i.e. the range select on *Current_price*) will be added above the file scan operator.

3.5 Incremental Grouping of Join Operators

Since join operators are usually expensive, sharing common join operations can significantly reduce the amount of computation. Figure 3.11 shows a query with a join operator that, for each company, retrieves the price of its stock and the company's profile. The signature for the join operation is shown on the right side of the figure. A join signature in our approach contains the names of the two data sources and the predicate for the join. The group optimizer groups join queries with the same join signatures. A constant table is not needed in this case because there is only one output intermediate file, whose name is stored in the split operator. This file is used to hold the results of the shared join operation.

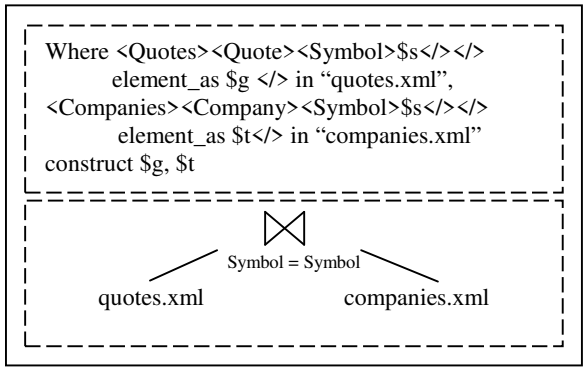


Figure 3.11 an example query with join operator and its signature

There are two ways to group queries that contain both join operators and selection operators. Figure 3.12 shows such an example, which retrieves all stocks in the computer service industry and the related company profiles. The group optimizer can place the selection either below or above the join, so that two different grouping sequences can be used during incremental group optimization process. The group optimizer chooses the better one based on a cost model. We discuss these alternatives below using the query example in Figure 3.12.

```
Where <Quotes><Quote><Symbol>$s</>
   <Industry>"Computer Service"</></>
        element_as $g </> in "quotes.xml",
<Companies><Company><Symbol>$s</></>
        element_as $t</> in "companies.xml"
construct $g, $t
```

Figure 3.12 an example query with both join and selection operators

If the selection operator (e.g., on *Industry*) is pulled above the join operator, the group optimizer first groups the query by the join signature. The selection signature, which contains the intermediate file, is grouped next. The advantage of this method is that it allows the same join operator to be shared by queries with different selection operators. The disadvantage is that the join, which will be performed before the selection, may be very expensive and may generate a large intermediate file. If there are only a small number of queries in the join group and each of them has a highly selective selection predicate, then this grouping method may be even more expensive than evaluating the queries individually.

Alternatively, the group optimizer can push down the selection operator (e.g., on *Industry)* to avoid computing an expensive join. First, the signature for the selection operator is matched with an existing group. Then a file scan operator on the

intermediate file produced by the selection group is added and the join operator is rewritten to use the intermediate file as one of its inputs. Finally, the group optimizer incrementally groups the join operation using its signature. Compared to the first approach, this approach may create many join groups with significant overlap between them. Note, however, that this same overlap exists in the non-grouping approach. Thus, in general, this method always outperforms than non-grouping approach.

The group optimizer will select one of these two strategies based on a cost model. To date we have implemented the second approach in NiagaraCQ. In the future we plan on implementing the first strategy and compare the performance of the two approaches.

3.6 Grouping Timer-based Continuous Queries

Since timer-based queries are only periodically executed their use can significantly reduce computation time and make the system more scalable. Timer-based queries are grouped in the same way as change-based queries except that the time information needs to be recorded at installation time. Grouping large number of timer-based queries poses two significant challenges. First, it is hard to monitor the timer events of those queries. Second, sharing the common computation becomes difficult due to the various time intervals. For example, two users may both request the query in Figure 3.1 with different time intervals, e.g. weekly and monthly. The query with the monthly interval should not repeat the weekly query's work. In general, queries with various time intervals should be able to share the results that have already been produced.

3.6.1 Event Detection

Two types of events in NiagaraCQ can trigger continuous queries. They are data-source change events and timer events. Data sources can be classified into push-based and pull-based. Push-based data sources will inform NiagaraCQ whenever interesting data is changed. On the other hand, changes on pull-based data sources must be checked periodically by NiagaraCQ.

Timer-based continuous queries are fired only at specified times. However, queries will not be executed if the corresponding input files have not been modified. Timer events are stored in an event list, which is sorted in time order. Each entry in the list corresponds to a time instant where there exists a continuous query to be scheduled. Each query in NiagaraCQ has a unique id. Those query ids are also stored in the entry. Whenever a timer event occurs, all related files will be checked. Each query in the entry will be fired if its data source has been modified since its last firing time. The next firing times for all queries in the entry are calculated and the queries are added into the corresponding entries on the list.

3.6.2 Incremental Evaluation

Incremental evaluation allows queries to be invoked only on the changed data. It reduces the amount of computation significantly because typically the amount of changed data is smaller than the original data file. For each file, on which continuous queries are defined, NiagaraCQ keeps a "delta file" that contains recent changes. Queries are run over the delta files whenever possible instead of their original files. However, in some cases the complete data files must be used, e.g., incremental evaluation of join operators. NiagaraCQ uses different techniques for handling delta files of ordinary data sources and those of intermediate files used to store the output of the split operator. NiagaraCQ calculates the changes to a source XML file and merges the changes into its delta file. For intermediate files, outputs from the split operators are directly appended to the delta file.

In order to support timer-based queries, a time stamp is added to each tuple in the delta file. Since timer-based queries with different firing times can be defined on one file, the delta file must keep data for the longest time interval among those queries that use the file as an input. At query execution time, NiagaraCQ fetches only tuples that were added to the delta file since the query's last firing time.

Whenever a grouped plan is invoked, the results of its execution are stored in an intermediate file regardless of whether or not queries defined on these intermediate files should be fired immediately. Subsequent invocations of this group query do not need to repeat previous computation. Upper level queries defined on intermediate files will still be fired at their scheduled execution time. Thus, the shared computation is totally transparent to these subsequent operators.

3.7 Memory Caching

Due to the desired scale of the system, we do not assume that all the information required by the continuous queries and intermediate results will fit in memory. Caching is used to obtain good performance with a limited amount of memory. NiagaraCQ caches query plans, system data structures, and data files for better performance.

1. Grouped query plans tend to be memory resident since we assume that the number of query groups is relatively small. Non-grouped change-based queries may be cached using an LRU policy that favors frequently fired queries. Timer-based queries with shorter firing intervals will have priority over those with longer intervals.
2. NiagaraCQ caches recently accessed files. Small delta files generated by split operators tend to be consumed and discarded. A caching policy that favors these small files saves lots of disk I/Os.
3. The event list for monitoring the timer-based events can be large if there are millions of timer-based continuous queries. To avoid maintaining the whole list in memory, we keep only a "time window" of this list. The window contains the front part of the list that should be kept in memory, e.g. within 24 hours.

4. IMPLEMENTATION

NiagaraCQ is being developed as a sub-system of Niagara project. The initial version of the system was implemented in Java (JDK1.2). A validating XML parser (IBM XML4J) from IBM is used to parse XML documents. We describe the system architecture of NiagaraCQ in Section 4.1 and how continuous queries are processed in Section 4.2.

4.1 System Architecture

Figure 4.1 shows the architecture of Niagara system. NiagaraCQ is a sub-system of Niagara that handles continuous queries. NiagaraCQ consists of

1. A continuous query manager, which is the core module of NiagaraCQ system. It provides a continuous query interface to

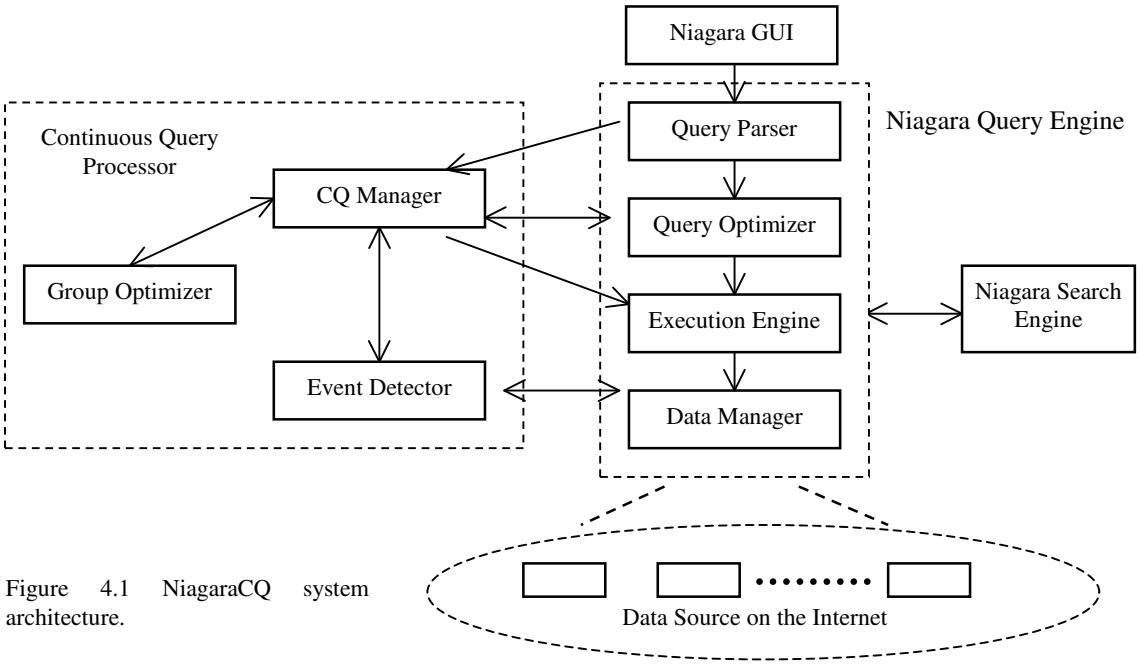


Figure 4.1 NiagaraCQ system architecture.

users and invokes the Niagara query engine to execute fired queries.

2. A group optimizer that performs incremental group optimization.
3. An event detector that detects timer events and changes of data sources.

In addition, the Niagara data manager was enhanced to support the incremental evaluation of continuous queries.

4.2 Processing Continuous Queries

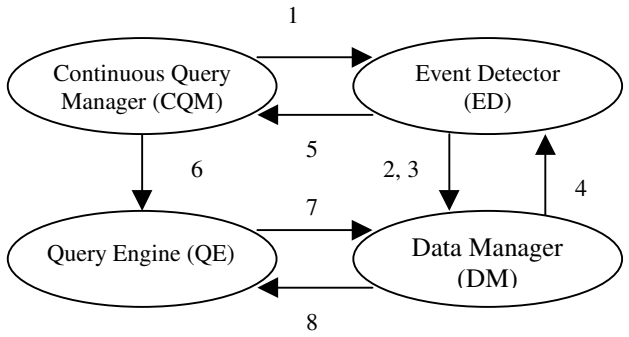
Figure 4.2 shows the interactions among the Continuous Query Manager, the Event Detector and the Data Manager as continuous queries are installed, detected, and executed. Continuous query processing is discussed in following sections.

4.2.1 Continuous Query Installation

When a new continuous query enters the system, the query is parsed and the query plan is fed into the group optimizer for incremental grouping. The group optimizer may split this query into several queries using the query-split scheme described in Section 3. The continuous query manager then invokes the Niagara query optimizer to perform common query optimization for these queries and the optimized plans are stored for future execution. Timer information and data source names of these queries are given to the Event Detector (Step 1 in Figure 4.2). The Event Detector then asks the Data Manager to monitor the related source files and intermediate files (Step 2 in Figure 4.2), which in turn caches a local copy of each source file. This step is necessary in order to detect subsequent changes to the file.

The Event Detector monitors two types of events: *timer events* and *file-modification* events. Whenever such events occur, the Event Detector notifies the Continuous Query Manager about which queries need to be fired and on which data sources.

The Data Manager in Niagara monitors web XML sources and intermediate files on its local disk. It handles the disk I/O for both ordinary queries and continuous queries and supports both push-based and pull-based data sources. For push-based data sources, the Data Manager is informed of a file change and notifies Event Detector actively. Otherwise, the Event Detector periodically asks the Data Manager to check the last modified time.



1. CQM adds continuous queries with file and timer information to enable ED to monitor the events.
2. ED asks DM to monitor changes to files.
3. When a timer event happens, ED asks DM the last modified time of files.
4. DM informs ED of changes to push-based data sources.
5. If file changes and timer events are satisfied, ED provides CQM with a list of firing CQs.
6. CQM invokes QE to execute firing CQs.
7. File scan operator calls DM to retrieve selected documents.
8. DM only returns data changes between last fire time and current fire time.

Figure 4.2 Continuous Query processing in NiagaraCQ

4.2.2 Continuous Query Deletion

A system unique name is generated for every user-defined continuous query. A user can use this name to retrieve the query status or to delete the query. Queries are automatically removed from the system when they expire.

4.2.3 Execution of Continuous Queries

The invocation of a continuous query requires a series of interactions among the Continuous Query Manager, Event Detector and Data Manager.

When a timer event happens, the Event Detector first asks the Data Manager if any of the relevant data sources have been modified (Step 3 in Figure 4.2). The Data Manager returns a list of names of modified source files. The Data Manager also notifies the Event Detector when push-based data sources have been changed (Step 4 in Figure 4.2). If a continuous query needs to be executed, its query id and the names of the modified files are sent to the Continuous Query Manager (Step 5 in Figure 4.2). The Continuous Query Manager invokes the Niagara query engine to execute the triggered queries (Step 6 in Figure 4.2). At execution time, the Query Engine requests data from the Data Manager (Step 7. in Figure 4.2). The Data Manager recognizes that it is a request for a continuous query and returns only the delta file (Step 8 in Figure 4.2). Delta files for source files are computed by performing an XML-specific "diff" operation using the original file and the new version of the file.

5. EXPERIMENTAL RESULTS

We expect that for a continuous query system over the Internet, incremental group optimization will provide substantial improvement to system performance and scalability. In the following experiments, we compare our incremental grouping approach with a non-grouping approach to show benefits from sharing computation and avoiding unnecessary query invocations.

5.1 Experiment Setting

The following experiments were conducted on a Sun Ultra 6000 with 1GB of RAM, running JDK1.2 on Solaris 2.6.

```
<!ELEMENT Quotes ( Quote )*>
<!ELEMENT Quote ( Symbol, Sector, Industry,
Current_Price, Open, PrevCls, Volume, Day's_range,
52_week_range?, Change_Ratio>
<!ELEMENT Day's_range (low, high)>
<!ELEMENT 52_week_change (low, high)>
```

Figure 5.1 DTD of quotes.xml

```
<!ELEMENT Companies ( Company )*>
<!ELEMENT Company ( Symbol, Name, Sector, Industry,
Company_profiles?>
<!ELEMENT Company_profiles (Capital, Employees,
Address, Description)>
<!ELEMENT Address (City, State)>
```

Figure 5.2 DTD of companies.xml

Data Sets

Our experiments were run against a database of stock information consisting of two XML files, "quotes.xml" and "companies.xml". "Quotes.xml" contains stock information on about 5000 NASDAQ companies. The size of "quotes.xml" is about 2 MB. Related company information is stored in "companies.xml", whose size is about 1MB. The DTDs of these two XML files are given in Figure 5.1 and 5.2, respectively. Data changes on "quotes.xml" are generated artificially to simulate the real stock market and continuous queries are triggered by these changes. The "companies.xml" file was not changed during our experiments.

We give a brief description of the assumptions that we made to generate "quotes.xml". Each stock has a unique *Symbol* value. The *Industry* attribute takes a value randomly from a set with about 100 values. The *Change_Ratio* represents the change percentage of the current price to the closing price for the previous session. It follows a normal distribution with a mean value of 0 and standard deviation of 1.0.

Since time spent calculating changes in source files is the same for both the grouped and non-grouped approaches, we run our experiments directly against the data changes. Unless specified, the number of "tuples" modified is 1000, which is about 400K bytes.

Queries

Although users may submit many different queries, we hypothesize that many queries will contain similar expression signatures. In our experiments, we use four types of queries to represent the effect of grouping queries in a stock environment by their expression signatures.

```
Where <Quotes><Quote><Symbol>"INTC"</></>
element_as $g </> in "quotes.xml", construct $g
```

Query Type-1 Example: Notify me when Intel stocks change.

```
Where <Quotes><Quote><Change_Ratio>$c</></>
element_as $g </> in "quotes.xml", $c > 0.05
construct $g
```

Query Type-2 Example: Notify me of all stocks whose prices rise more than 5 percent.

```
Where <Quotes><Quote><Symbol>"INTC"</>
<Current_Price>$p</></> element_as $g </>
in "quotes.xml", $p < 100,   construct $g
```

Query Type-3 Example: Notify me when Intel stock trades below 100 dollars.

```
Where <Quotes><Quote><Symbol>$s</><Industry>
"Computer Service"</></> element_as $g </>
in "quotes.xml",
<Companies><Company><Symbol>$s</></>
element_as $t</> in "companies.xml"
construct $g, $t
```

Query Type-4 Example: Notify me all of changes to stocks in the computer service industry and related company information.

- Type-1 queries have the same expression signature on the equal selection predicate on *Symbol*.
- Type-2 queries have the same expression signature on the range selection predicate on *Change_ratio*.

- Type-3 queries have two common expression signatures, one is on the equal selection predicate on *Symbol*, and the other is on the range selection predicate on *Current_price*. The expression signature of the equal selection predicate is used for grouping Type-3 queries because it is more selective than that of the range predicate.
- Type-4 queries contain expression signatures for both selection and join operators. Selection operators are pushed down under join operators. The incremental group optimizer first groups selection signatures and then join signatures.

Queries of Type-3 are generated following a normal distribution with a mean value of 3 and a standard deviation of 1.0. Queries of the other types are generated using different constants following a uniform distribution on the range of values in the data unless specified.

5.2 Interpretation of Experimental Result

The parameters in our experiments are:

1. **N**, the number of installed queries, is an important measure of system scalability.

2. **F**, the number of fired queries in the grouping case. The number of fired queries may vary depending on triggering conditions in the grouping case. For example, in a Tye-1 query, if Intel stock does not change, queries defined on "INTC" are not scheduled for execution after the common computation of the group. This parameter does not affect non-grouping queries.

3. **C**, the number of tuples modified.

In our grouping approach, a user-defined query consists of grouped part and non-grouped part. $\mathbf{T_g}$ and $\mathbf{T_{ng}}$ represent the execution time of each part. The execution time **T** for evaluating **N** queries is the sum of $\mathbf{T_g}$ and $\mathbf{T_{ng}}$ of each of **F** fired queries, $T = T_g + \sum_{F} T_{ng}$, because the grouped portion is executed only once.

Since the non-grouping strategy needs to scan each XML data source file multiple times, we cache parsed XML files in memory so that both approaches scan and parse XML files only once. This ensures that the comparison between the two approaches is fair. However, in a production system, parsed XML files probably could not be retained in memory for long periods of time. Thus, many non-grouped queries may each have to scan and parse the same XML files multiple times.

5.2.1 Experimental results on single type queries

We studied how effectively incremental group optimization works for each type of query. We measured and compared execution time for queries of each type for both the grouping and non-grouping approaches.

Experiment results on type-1 queries

Experiment 1. (Figure 5.3) C =1000 tuples.

- **Case 1**: **F = N**, i.e. all queries are fired in both approaches.

The execution time of the non-grouping approach grows dramatically as N increases. It cannot be applied to a highly loaded system. On the other hand, the grouping approach consumes significantly less execution time by sharing the computation of the selection operator. It also grows more slowly because in a single Type-1 query $\mathbf{T_{ng}}$ is much smaller than $\mathbf{T_g}$.

- **Case 2**: **F = 100**, i.e., 100 queries are invoked in the grouping approach.

In the grouping approach, the execution time of Case 2 is almost constant when **F** is fixed. The execution time of the grouping approach depends on number of fired queries **F**, not on the total number of installed queries **N**. The reason is that, although $\mathbf{T_g}$ increases as **N** grows, this shared computation is executed only once and is a very small portion of total execution time. The execution time for the upper queries, which is proportional to the number of fired queries F, dominates the total execution time. On the other hand, the execution time for the non-grouping approach is proportional to **N** because all queries are scheduled for execution.

Experiment 2. (Figure 5.4) F = N = 2000 queries

In this experiment we explore the impact of **C**, the number of modified tuples, on the performance of the two approaches. **C** is varied from 100 tuples (about 40K bytes) to 2000 tuples (about 800K bytes). Increasing **C** will increase the query execution time. For the non-grouping approach, the total execution time is proportional to **C** because the selection operator of every installed query needs to be executed. For the grouping approach, the execution time is not sensitive to the change of **C** because the increase of $\mathbf{T_g}$ only counts for a small percentage of the total execution time and the sum of $\mathbf{T_{ng}}$ of all fired queries does not change because of the predicate's selectivity.

Experiment results for Type-2, 3, 4 queries (Figure 5.5, 5.6, 5.7) C =1000 tuples, **F = N**

We discuss the influence of different expression signatures in this set of experiments.

Figure 5.5 and Figure 5.6 show that our group optimization works well for various selection predicates. Type-2 queries are grouped according to their range selection signature. Type-3

queries have two signatures. The group optimizer chooses an equal predicate to group queries since it is more selective.

Figure 5.7 shows the results for Type-4 queries. Type-4 queries have one selection signature and one join signature. The selection operator is pushed below the join operator. Queries are first grouped by their selection signature. There are 100 different industries in our test data set. The output of the selection group is written to 100 intermediate files and one hundred join groups are created. Each join group consumes one of the intermediate files as its input. The difference between the execution time with and without grouping is much larger than in the previous experiments because a join operator is more expensive than a selection operator.

5.2.2 Experiment results on mixed queries of Type-1 and type-3 (Figure 5.8) C =1000 tuples, F = N (N/2 Type-1 queries and N/2 Type-3 queries)

Previous experiments studied each type of query separately for the purpose of showing the effectiveness of different kinds of expression signatures. Our incremental group optimizer is not limited to group only one type of queries. Different types of queries can also be grouped together if they have common

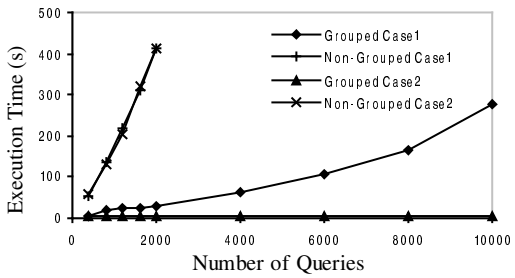


Figure 5.3

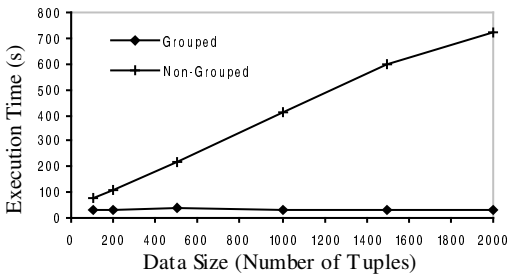


Figure 5.4

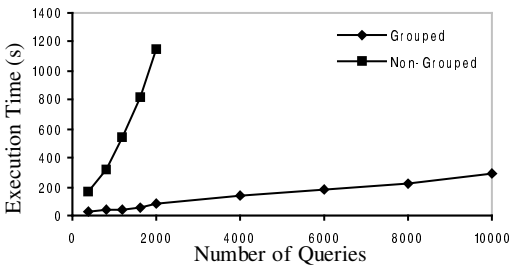


Figure 5.5

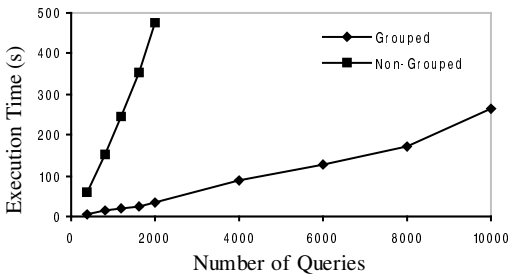


Figure 5.6

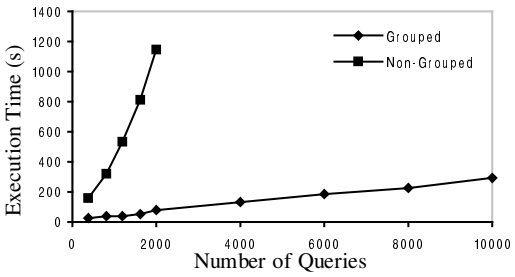


Figure 5.7

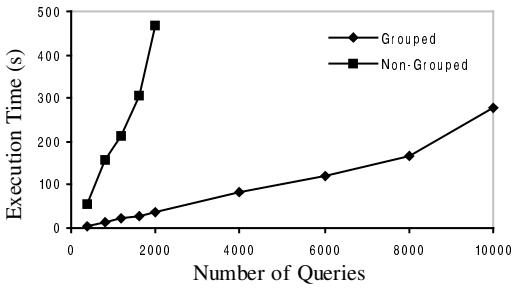


Figure 5.8

signatures. In this experiment, Type-1 queries and Type-3 queries are grouped together because they have the same selection signature. Figure 5.8 shows the performance difference between the grouped and non-grouped cases.

5.3 System Status and Future Work

A prototype version of NaigraCQ has been developed, which includes a Group Optimizer, Continuous Query Manager, Event Detector, and Data Manager. As the core of our incremental group optimization, the Group Optimizer currently can incrementally group selection and join operators. Our incremental group optimizer is still at a preliminary stage. However, incremental group optimization has been shown to be a promising way to achieve good performance and scalability. We intend to extend incremental group optimization to queries containing operators other than selection and join. For example, sharing computation for expensive operators, such as aggregation, may be very effective. “Dynamic regrouping” is another interesting future direction that we intend to explore.

6. RELATED WORK AND DISCUSSION

Terry et al. first proposed the notion of "continuous queries" [TGNO92] as queries that are issued once and run continuously. He used an incremental evaluation approach to avoid repetitive computation and return only new results to users. Their approach was restricted to append-only systems, which is not suitable for our target environment. NiagaraCQ uses an incremental query evaluation method but is not limited to append-only data sources. We also include action and timer events in Niagara continuous queries.

Continuous queries are similar to triggers in traditional database systems. Triggers have been widely studied and implemented [WF89][MD89][SJGP90][SPAM91][SK95]. Most trigger systems use an Event-Condition-Action (ECA) model [MD89]. General issues of implementing triggers can be found in [WF89].

NiagaraCQ is different from traditional trigger systems in the following ways.

1. The main purpose of the NiagaraCQ is to support continuous query processing rather than to maintain data integrity.
2. NiagaraCQ is intended to support millions of continuous queries defined on large number of data sources. In a traditional DBMS, a very limited number of triggers can be installed on each table and a trigger can usually only be defined on a single table.
3. NiagaraCQ needs to monitor autonomous and heterogeneous data sources over the Internet. Traditional trigger systems only handle local tables.
4. Timer-based events are supported in NiagaraCQ.

Open-CQ [LPT99] [LPBZ96] also supports continuous queries on web data sources and has functionality similar to NiagaraCQ. NiagaraCQ differs from Open-CQ in that we explore the similarity among large number of queries and use group optimization to achieve system scalability.

The TriggerMan [HCH+99] project proposes a method for implementing a scalable trigger system based on the assumption that many triggers may have common structure. It uses a special selection predicate index and an in-memory trigger cache to achieve scalability. We share the same assumption in our work and borrow the concept of an expression signature from their work. We mainly focus on the incremental grouping of a subset of the most frequently used expression signatures, which are in the format "Attribute op Constant", where op is one of "<", "=" and ">". The major differences between NiagaraCQ and TriggerMan are:

1. NiagaraCQ uses an incremental group optimization strategy.
2. NiagaraCQ uses a query-split scheme to allow the shared computation to become an individual query that can be monitored and executed using a slightly modified query engine. TriggerMan uses a special in-memory predicate index to evaluate the expression signature.
3. NiagaraCQ supports grouping of timer-based queries, a capability not considered in [HCH+99].

Sellis's work [Sel86] focused on finding an optimal plan for a small group of queries (usually lower than ten) by recognizing a containment relationship among the selection predicates of queries with both selection and join operators. This approach for group optimization was very expensive and not extendable to a large number of queries.

Recent work [ZDNS98] on group optimization mainly focuses on applying group optimization to solve a specific problem. Our approach also falls into this category. Alert [SPAM91] was among the earliest active database systems. It tried to reuse most parts of a passive DBMS to implement an active database.

7. CONCLUSION

Our goal is to develop an Internet-scale continuous query system using group optimization based on the assumption that many continuous queries on the Internet will have some similarities. Previous group optimization approaches consider grouping only a small number of queries at the same time and are not scalable to millions of queries. We propose a new "incremental grouping" methodology that makes group optimization more scalable than the previous approaches. This idea can be applied to very general group optimization methods. We also propose a grouping method using a query-split scheme that requires minimal changes to a general purposed query engine. In our system, both timer-based and change-based continuous queries can be grouped together for event detection and group execution, a capability not found in other systems. Other techniques to make our system scalable include incremental evaluation of continuous queries, use of both pull and push models for detecting heterogeneous data source changes and a caching mechanism. Preliminary experiments demonstrate that our incremental group optimization significantly improves the execution time comparing to non-grouping approach. The results of experiments also show that the system can be scaled to support very large number of queries.

8. ACKNOWLEDGEMENT

We thank Zhichen Xu for his discussion with the first author during initial writing of the paper. We are particularly grateful to Ashraf Aboulnaga, Navin Kabra and David Maier for their careful review and helpful comments on the paper. We also thank the anonymous referees for their comments. Funding for this work was provided by DARPA through NAVY/SPAWAR Contract No. N66001-99-1-8908 and NSF award CDA-9623632.

9. REFERENCES

[CM86] U..S. Chakravarthy and J. Minker. Multiple Query Processing in Deductive Databases using Query Graphs. VLDB Conference 1986: 384-391.

[DFF+98] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu. XML-QL: A Query Langaage for XML. http://www.w3.org/TR/NOTE-xml-ql.

[HCH+99] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J.B.Park and A. Vernon. Scalable Trigger Processing. In proceeding of 15th ICDE, page 266-275, Sydney, Australia, 1999.

[HJ94] E. N. Hanson and T. Johnson. Selection Predicate Indexing for Active Databases Using Interval Skip List. TR94-017. CIS department, University of Florida, 1994.

[LPBZ96] L. Liu, C. Pu, R. Barga, T. Zhou. Differential Evaluation of Continual Queries. ICDCS 1996: 458-465.

[LPT99] L. Liu, C. Pu, W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. TKDE 11(4): 610-628 (1999).

[MD89] D. McCarthy and U. Dayal. The architecture of an active database management system. SIGMOD 1989: 215-224.

[RC88] A. Rosenthal and U. S. Chakravarthy. Anatomy of a Modular Multiple Query Optimizer. VLDB 1988: 230-239.

[Sel86] T. Sellis. Multiple query optimization. ACM Transactions on Database Systems, 10(3), 1986.

[SJGP90]M. Stonebraker, A. Jhingran, J. Goh and S. Potamianos. On Rules, Procedures, Caching and Views in Data Base Systems. SIGMOD Conference 1990: 281-290.

[SK95] E. Simon, A. Kotz-Dittrich. Promises and Realities of Active Database Systems. VLDB 1995: 642-653.

[SPAM91] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An architecture for transforming a passive dbms into an active dbms. VLDB 1991: 469-478.

[TGNO92] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous Queries over Append-Only Databases. SIGMOD 1992: 321-330.

[WF89] J. Widom and S.J. Finklestein. Set-Oriented Production Rules in Relational Database Systems. SIGMOD Conference 1990: 259-270.

[ZDNS98] Y. Zhao, P. Deshpande, J. F. Naughton, A. Shukla. Simultaneous Optimization and Evaluation of Multiple Dimensional Queries. SIGMOD 1998: 271-282.