Scalable Continuous Query System for Web Databases

Ather Saeed, Savitri Bevinakoppa

School of Computer Science and Information Technology, RMIT University,
Melbourne, Australia

Abstract. Continuous Queries (CQ) help users to retrieve results as soon as they become available. The CQ keeps track of two important events. If there is any change in the source information, it immediately notifies the user about that particular change. Secondly, it also keeps track of the timer-based events, in case information is required after a fixed period of time. Existing techniques such as OpenCQ[1] and NiagaraCQ[2] are inadequate to optimise Continuous Queries. These techniques focus on defining the semantics for execution of CQ and much less effort was spent to define the tradeoffs required for evaluating the specific condition and generating a cost-effective query evaluation plan (QEP). The optimising scheme used in [1],[2] for information retrieval is not appropriate for an environment like the Internet. This paper also provides a new architecture and group optimisation strategy for the efficient retrieval of CQ on the web.

1 Introduction

Continuous Queries (CQ) [1],[2],[3],[4] help users to obtain new results soon after an update occurs in a system. Once a CQ is triggered, it will continuously run and monitors a particular threshold value. Result will be sent whenever the desired value becomes true.

Successful CQ execution is a major challenge in distributed environments because of the unpredictable behaviour of nodes connected to a network. Failure of CQ will schedule the same query over and over again, which is I/O and resource intensive. If an appropriate optimization strategy is not used latency would become extremely high and it will slow down the information retrieval process on the Web.

NiagaraCQ [2] addresses group query optimization to some extent by providing a comparison of pull-based and push-based approaches using signature matching and query split scheme. This is not a cost-effective strategy to generate a highly efficient global access plan necessary for CQ execution.

OpenCQ [1] address the problem of Continuous Queries and defines necessary semantics for the sequential execution of each CQ.

OpenCQ[1] generates an execution plan for each CQ separately. If CQ fails, execution would be resource and I/O intensive especially in queries that involve large join operations.

Saeed A. and Bevinakoppa S. (2004).
Scalable Continuous Query System for Web Databases.
In *Proceedings of the 2nd International Workshop on Web Services: Modeling, Architecture and Infrastructure*, pages 59-72
DOI: 10.5220/0002676500590072

In this paper we address importance of web-based CQ in regards to the optimisation problems with the existing Continuous Query systems [1],[2]. By performing experiments, we also show that the scalability of a system is compromised when an appropriate optimisation scheme is not used. This paper also provides analysis of the two systems: OpenCQ[1] and NiagaraCQ[2] which highlights the limitations of the existing optimization techniques used in both systems.

The remaining paper is organized as follows: Section 2 gives an overview of problems with the existing Continuous Query systems. Section 3 describes our proposed grouping technique. Section 4 describes the architecture of CQ system. Section 5 describes cost parameters required for the cost analysis of a CQ-System. Section 6 describes the experimental analysis. Section 7 describes related work. Section 8 describes the conclusions and future work. Section 9 gives the Web CQ-system overview. Finally section 10 contains references used in this paper.

2 Problems with Existing CQ Systems

Continuous Queries are standing and long running queries that monitor a particular update based on the specified condition. Therefore it is hard to stop CQ in the middle, particularly when the update frequency is very high. In highly dynamic environments like the Internet, major challenge is to control trade-off between the specified condition, such as the frequency of update and efficiency of the Query Evaluation Plan (*QEP*).

The other challenge is to send the most accurate and up-to-date information in an environment like the Internet, where source information is constantly changing and the behaviour of node is highly unpredictable.

Therefore we need a new multi-query optimization approach that exploits the commonalities among the sub-expression, present in queries, with a focus on providing a shortest access path, which can be used for answering a particular query. It is only possible if the global access plan is used to answer queries.

The problem with NiagaraCQ [2] is that, its major focus was on routing a huge amount of data using XML-QL and retrieving information from the XML type files and data integrity was not the major concern. We have observed through experimentation that sometimes no result was retrieved at all but the cost for generating a valid execution plan was also very high.

In NiagaraCQ [2], a query split scheme was used, which group queries by matching signatures and consider it as a potential group, by ignoring the fact that it might not be the potential solution or plan, which will further increase the cost of scanning query evaluation plans. Beside that more appropriate strategy is required to cache results in order to deal with the temporary materialized results.

Although OpenCQ[1] provides data integrity to some extent but it does not provide a cost-effective solution to the problem of scalability because it sequentially executes each CQ. Many researches such as [5],[6],[7] have shown that sequential execution of queries is expensive compared to the global execution in many circumstances. Especially when queries are routed to heterogeneous data sources with large join expressions.

The major problem is to look at one fundamental question which is: How to deal with the scalability issues in an environment like the Internet? Especially when the

query involves a large number of joins and triggered condition needed to check the frequency of update is very high?
The answer to the above question lies in the new scalable architecture that optimises a group of Continuous Queries together with a view to exploit commonalities by providing a cost-effective solution to the evaluation of monotone or redundant queries which will be discussed in the next section.

3 The Proposed Grouping Technique

Existing techniques [1],[2] for optimising CQ are not adequate, which is clear from the above discussion. The technique is not suitable due to three reasons:
A group might not produce an optimal solution to the problem.
Unsuccessful execution of CQ will increase the latency rate of data retrieval on the Internet due to which query evaluation cost would become extremely high.
Efficiency of query evaluation plan (qep) was not taken into account in order to get cost-effective solution.

Our grouping strategy takes care of the above-mentioned problems by dynamic re-grouping of CQ and generates a global access plan which is as follows:

A single query might have many plans; the role of an MQO optimiser is to find a best plan with a minimum cost. For simplicity we will use the same notion mentioned in many database research papers [5], [6], [7].

Suppose we have two plans, plan (A) is $X \rhd\lhd Y \rhd\lhd Z$ and plan (B) is $Y \rhd\lhd Z \rhd\lhd T$, where $\rhd\lhd$ represents a join operation.

MQO (Multiple Query Optimisation) is a two-step process. In first step the optimiser will decompose queries into sub queries to find common sub expressions, then executes the partial plans and materialize the results as shown in fig.3.2. In the second step it explore the search space using some heuristic algorithm and starts merging the best plans and generates a global plan that will minimize the cost of a query evaluation. Our optimiser will filter the timer-based queries and store the temporary query-id (qid) in cache memory for a certain period of time. As joins are the most expensive operation on any database system. The main purpose of the query decomposition scheme is to reduce the query evaluation time by efficiently retrieving results from the primary storage then incrementally evaluate each query before sending it to the user on the Internet. The Join trees of Plan (A) and Plan (B) using a bottom-up approach is shown below on the next page.

In the fig.3.1, leaf nodes represent relation and inner nodes represent join operators. The whole query tree is not shown in the figure, because major goal was just to show that many queries usually use the same base relations and contains common sub-expressions.

Previous researches in MQO [5],[6] have shown that left-deep trees are more efficient to execute, but due to the sensitive nature of continuous queries, as soon as the optimiser finds the best plan, it starts merging them together and store the temporary query-id (Qid) in the main memory for a certain period of time instead of keeping the whole result in the main memory. Materialized Result (Mres) can be added or deleted from the cache and copied to hard disk. In this case additional disk

I/O will be required. The aim is to reuse the same result for the next group when required. Mres is also shown in result table fig.3.3 and Triggeri etc are triggers defined on data soures.

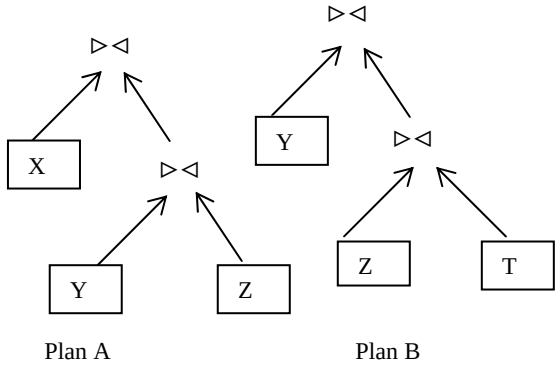

Fig. 3.1. Right deep join trees of Query plans

3.1 Grouping Strategy

Our grouping strategy will partially execute each Plan as follows:

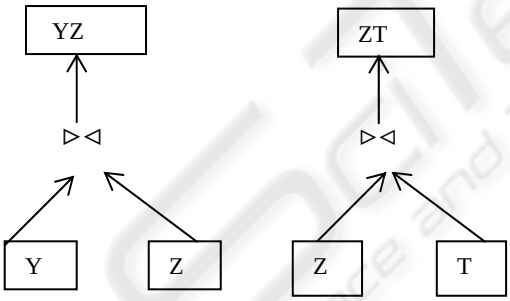

Fig. 3.2. . Partial results of Plan A and Plan B

Global access plan for the plan A and plan B is shown on the next page. YZ and ZT are the partial result of Plan A and Plan B, which is a DAG (Directed Acyclic Graph). For simplicity we are showing the joins only. Our model is not limited to executing joins. Although it holds the projection for a certain period of time until a global access plan is finalized and passed to the optimiser.
In order to make our idea more general, the whole query tree of Plan A and Plan B is not shown. More details can be found in [5],[7],[9]. The select operations are pushed down the query tree in order to reduce the search space for efficient join processing.

Our MQO algorithm is an extension of [5],[7],[9]. These algorithms are used for normal SQL-type queries, whereas we are dealing with event-driven and timer-based Continuous queries.

Beside that algorithms [5],[6],[7] were not designed for an environment like the Internet. In order to define the Multi-Graph strategy used in our model, we will use somewhat similar graph definition used in [5],[6]. The Main objective is to find a new technique that shows how to store, where to store and how to access the materialized nodes in case of CQ?

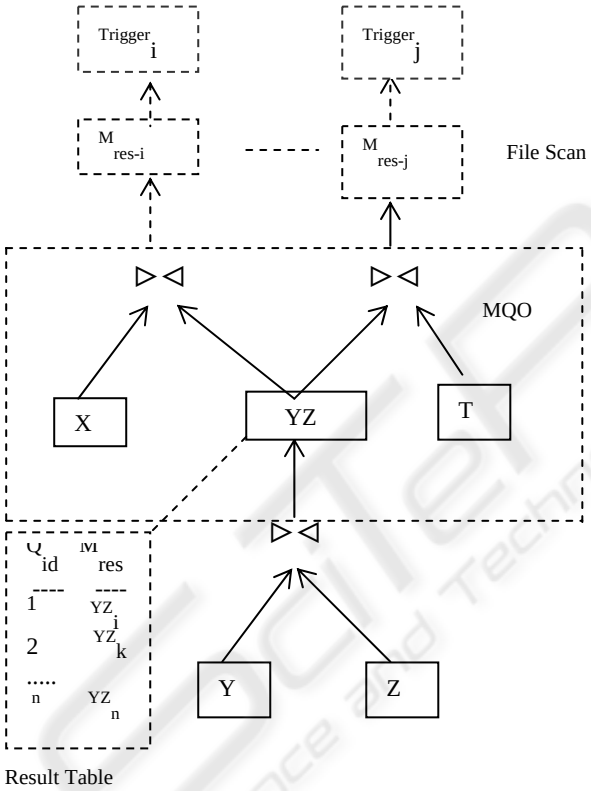

Fig. 3.3. Global Access Plan

3.2 IE-CQ Algorithm

Input: Multi-Query Graph G(V,E,L,$^{M}_{res}$), where V is vertex, E is edge, L is label of vertex $^{V}_{m}$ and $^{M}_{res}$ contains the materialized result in an intermediate file.

Decompose the queries into sub-queries, so that partial results of a query can be temporarily stored. If the query has any of the three conditions:

If a query shares common base relation
If query is identical or similar
If query subsume another query (as in [6])
materialize the node and generate a new arc that represents an edge of a vertex $^{V}_{n}$. Generate an arc from $^{V}_{m}$ to $^{V}_{n}$ apply the merge (similar to [6]) operation as shown in fig.3.2 and materialize the node.

Store $^{CQ}_{id}$ and cache result into an intermediate file $^{M}_{res,}$ Starts assembling the result of queries in regards to there Cqid and send it to the CQ-Scheduler as shown in fig.4.1. Continue the process for the next group and perform the process for $^{CQ}_{n-1}$ number of queries.Compare new group of queries with previous group plan, if the $^{M}_{res}$ can be shared. Always select plan with a minimum cost until no common expression exists in a group

3.3 Caching Technique

In our MQO, caching improves the performance of query evaluation plans by putting the materialized results in the main memory.

For simplicity we are using LRU-K [8] type caching technique. Due to the advancement in technology, we assume that main memory is large enough to hold results of queries for certain period of time as shown in the fig. 3.2. After the execution of a particular group, stored results and plans will be transferred to the disk in the form of small delta file somewhat similar to NiagaraCQ[2], which further involves additional I/O cost.

Benefit is that the next group might share results from the previously materialized nodes. This process will be continued for a certain period of time say from 24 to 36 hours. If new groups are getting benefit from the cached results and plans which are materialized for a particular time then it can be used for other newly created groups. Main objective is to reduce the number of hits to data sources as much as possible and evaluate results incrementally.

4 Web CQ System Architecture

Details about each component in the Continuous Query system is described as follows and is shown in fig.4:

4.1 CQ-Scheduler

The Continuous Queries are *content sensitive* and *event driven* {same term is used by Ling Liu}in [1], therefore the global plan created by the plan merger cannot be discarded because of the specified triggered condition. We are assuming that there is enough memory to hold the materialised results at least for one hour, if the query needs a time greater than one hour it can be written to disk and will be activated,

which will incur additional I/O cost. It schedules each query when the triggered condition becomes true.

4.2 Time Extractor

It will assign a unique ID to each query by extracting the time, before sending it to the global optimizer, which will be reassembled again before sending the result of a query to user.

4.3 CP- Analyzer

The Common Part analyser (CP-Analyser) is the most important part of the query optimiser that analyse all queries present in a particular group and exploits the commonalities with a view that result of a common sub-expression can be used for answering other queries. The $^{Q}_{id}$ will be cached and partial results of a query will be stored in a result table for certain time as shown in fig.3.2.

4.4 Cost Estimator

The main purpose of this module is to estimate the cost of generating a valid CQ execution plan *(QEP*) before sending to the data source. A valid plan is necessary to achieve scalability. we have already discussed above that if the query evaluation Plan is not valid then efficiency of the system will be compromised, which ultimately results in high latency.

4.5 Incremental Evaluator

The role of this particular module is to incrementally evaluate each query in order to avoid repetitive computation and has similarities with the techniques discussed in OpenCQ[1] and NiagaraCQ[2].

4.6 CQ-Client Interface

This section provides an interface, where user will be able to add or delete the Continuous Queries. This module will work as a front-end to Web CQ-System. It also assist users in writing XML and XSQL type queries.

4.7 Data Handler

The role of a Data handler module is to analyse the type of data request coming from a source on the Internet, whether it is a text file, XML type files or database systems, it retrieves the information and sends result in a required format to the user.

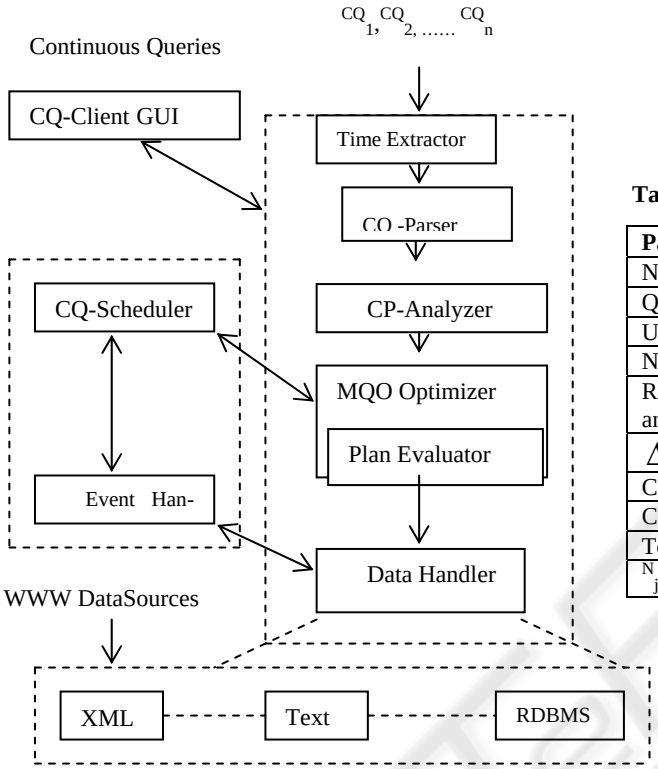

Fig. 4. Web CQ-System Architecture

Table 1. Cost Parameters

Parameters	**Values**
Network bandwidth	100 (mbps)
QEP evaluation time	Milli (secs)
Update frequency of node	Dynamic
Number of tuples	100-5000
R,S,T are XML, RDBMS and text files	400 rows in each data soruce.
ΔR, ΔS, ΔT	delta file for R,S,T
Cost of global access plan.	
Cost of evaluating each CQ	
Total number of queries	200-2000
N_j, F_{cq}, and T_C	Details section-6

5 Cost Parameters

Δr, Δs and Δt are the update, insert or delete event on a relation R. In our experimental analysis, we are separately storing the three possible events that can occur on a database system, which is not done in [1][2]. The benefit is less chances of getting errors, while sending final results to the CQ-scheduler. Keeping the whole snapshot of delta files [2], is not a cost-effective strategy. It is better to keep the delta file as small as possible.

The benefit of using IE-CQ algorithm is that, it can decompose a single query into multiple queries. Finally by using some graph traversal strategy like depth-first search somewhat similar to [6], we can prune the redundant nodes in a graph. Cost parameters are shown in the Table.1.

5.1 CQ Cost Estimation

Join based decomposition of CQ has similarity with the selection base decomposition. IE-CQ decomposes and assembles the results of $^{CQ}_{n}$ queries using very similar cost estimation scheme as mentioned in [5],[9],[10] for evaluating group of queries.

$^{CQ}_{1} = {}^{R}_{1} \rhd\lhd {}^{R}_{2} \rhd\lhd {}^{R}_{3} \rhd\lhd {}^{R}_{4}$
$^{CQ}_{2} = {}^{M}_{res} ({}^{CQ}_{1}) \rhd\lhd {}^{R}_{5}$
$^{CQ}_{3} = {}^{M}_{res} ({}^{CQ}_{2}) \rhd\lhd {}^{R}_{6}$
$^{CQ}_{n-1} = {}^{M}_{res} ({}^{CQ}_{n-2}) \rhd\lhd {}^{R}_{n}$

$$^{CQ}_{n} = \pi(\sigma({}^{M}_{res}({}^{CQ}_{n-1}))) . \quad (1)$$

If the ^{CQ} is of type $[\pi(\sigma(\rhd\lhd(\text{RELATIONS})]$ then materialized result ($^{M}_{res}$) of a query can be assembled as shown in equation (1). The above query evaluation as shown in (1) has similarity with [5], [6], [7], [9] and [10], but these schemes were never used in the context of CQ. In case of subsumption, if $^{CQ}_{n}$ is strongly related to $^{CQ}_{n-1\ then}\ ^{CQ}_{n-1}$ can be used to answer $^{CQ}_{n}$ or it contains a partial answer to query $^{CQ}_{n}$.

Final result of a ^{CQ} can be retrieved by scanning Δr with regards to the change in data source. In our model we are assuming that queries share a large number of common sub-expressions and that partial results of a query can be assembled using join based decomposition strategies as mentioned above, which is very similar to the strategy mentioned in [5], [6], [10].

We also assume that some queries might be answered by simply intersecting some previously stored intermediate results or could be obtained from the union of some previously stored results.

Our model provides a new approach to optimize Continuous Queries by generating a global access plan instead of blindly generating a plan by matching signature and consider it as a potential group such as in [2]. The major problem would be that; delta files will become bigger due to unnecessary information, which was not required to answer a particular query. In the worst case it might not be a potential solution to a particular group.

The cost of CQ is sum of five components; which can be obtained by slightly modifying the cost formula as mentioned in [5], [10].

Cost of scanning Relation: $R^{(CQ)}$
Cost of materializing nodes: $^{M}_{res}{}^{(CQ)}$
Cost of scanning materialized nodes: $^{M}_{res}{}^{(CQ)}$
Cost of re-scanning data source: λ
Cost of retrieving results of ^{CQ}: γ

Where λ is the ratio of update frequency with regards to the change in data sources.

$$\text{Cost}^{(CQ)} = {}^{R(CQ)} + 2\,{}^{M}{}_{res}{}^{(CQ)} + \gamma + \lambda \ . \qquad (2)$$

As the group part of a query is executed only once, therefore global plan can be obtained as follows:

$$\text{Cost}^{(MQO)} = \sum_{i=1}^{m} {}^{P}{}_{qep}{}^{(CQ_i)} + \sum_{j=1}^{k} {}^{G}{}_{qep}{}^{(CQ_i)} \ . \qquad (3)$$

Where ${}^{P}{}_{qep}$ is the partial query evaluation plans which is individually evaluated and ${}^{G}{}_{qep}$ is the group *QEP*, which is evaluated only once in an integrated way when the triggered condition becomes true.

6 Experimental Analysis

The experimental analysis of the technique for group optimization strategy mentioned in section 3 will be discussed in the following section:

The experiments are performed using Pentium IV 2400 MHz machines with 1GB RAM, 40GB Hard disk and running Windows 2000 / XP / NT.

The initial prototype of a system was designed in J2ee and Java XSQL Servlet API for the server side scripting such as CGI related tasks. Oracle 9i and MS SQL Server 2000 were used as database servers for retrieving and storing XML, text and SQL type data.

The types of Continuous Queries performed on Oracle 9i and MS SQL-Server 2000 data sources are described below.

Type-1: Notify me when Passenger No:2 and Passenger No:15 arrive at terminal 10.

This type of query is important in tracking a passenger if the message for them is of high importance.

Type-2: *Notify me when plane B707 takes-off from the USA and arrives in Canada.*

This type of query needs a partial execution between a two specific time interval such as departure and arrival. Arrival might be after 10 hours in Canada. Therefore partial plan of a query will be saved in an intermediate file and sent to CQ-Scheduler as shown in the fig.4.1 Finally after the arrival whole plan will be executed.

Type-3: Notify me every 10 minutes about the current position of the plane B707.

This type of query will be scheduled after every 10 minutes and notify about the current position of plane, which is a timer-base query.

6.1 CQ Installation Semantics

Our semantics for the execution of CQ are very similar to [1], [2] only the optimization approach is different as discussed above:

```
CQ-Install: Airline-DB SQL-query
   Query: SELECT <attr> FROM <source>
   WHERE <join-condition-evaluation>
Trigger: <specified-condition>
Stop: <minutes><daily><monthly>
```

The XSQL type query format the results in XML before sending it to the CQ-Scheduler as:

```
CQ-Install: XSQL-Query for xml DTD
    <?xml version=1.0?><xsql:query
    xmlns:xsql=urn:oracle-xsql>
    SELECT <attr> FROM <XML-Schema>
    WHERE<join-condition-evaluation>
Trigger: <specified-condition>
Stop:    <minutes><daily><monthly>
```

Changes to the data sources are made artificially in order to simulate the results of a query. Simulation results are shown in fig.6.1 and fig.6.2 graphs.

The purpose of simulation was to observe the behavior of grouping technique in regards to the sequential execution of Continuous queries, when it involves large join operations.

The graphs are based on results obtained from web CQ-System by applying type-1, type-2 and type-3 queries. In fig.6.1 following parameters are considered: $^{F}_{cq} = {}^{T}c =$ 200 and $^{N}_{j} < 10$, where $^{F}_{cq}$ are the triggered CQ and ^{T}c are changed-base triggers which are fired in case of any change occurred in the data source. $^{N}_{j}$ is the total number of joins operation permitted. In fig.6.1 and fig.6.2 the only parameter that is changed was the join parameter such as $^{N}_{j} > 10$ and the curve obtained was almost a straight line. One can easily observe that curve obtained was highly skewed in case of traditional non-grouped approach, which shows that our grouped approach for global evaluation of CQ is outperforming the existing ones mainly due to the following three reasons. 1) The global optimization approach that incrementally evaluate each query before sending to the user. 2) It takes care of operations, which involve large number of joins. 3) Our system also provides data integrity, which was not the major concern in [2].

LRU-K [8] type caching algorithm was used to store and transfer results to the secondary storage after a certain period of time. Discussion about LRU-K techniques for caching is beyond the scope of this paper and cannot be explained due to space limitation.

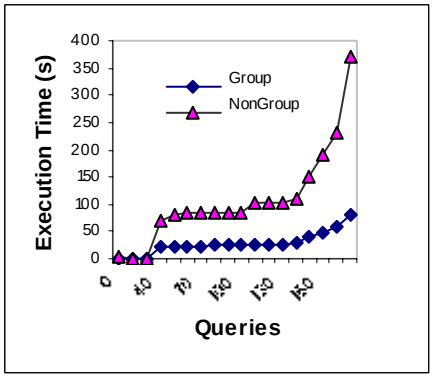

Fig. 6.1. Group and Non-grouped CQ

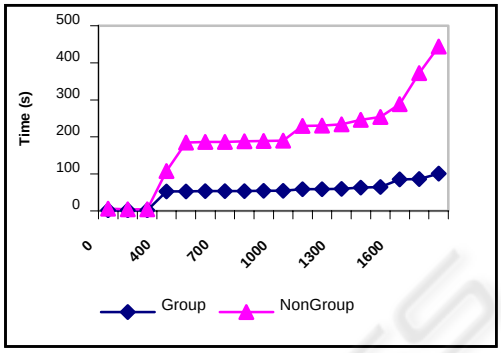

Fig. 6.2. Group and Non-grouped CQ

7 Related Work

A huge amount of research has already been done in the area of Continuous Queries. OpenCQ [1], provides semantics necessary for the execution of CQ in the form of triplet (Query, Trigger-Condition, Stop-condition), which also provides an optimization strategy for the sequential evaluation of CQ.

NiagaraCQ [2], provides a group optimization strategy using query split scheme, in which they match the signature and put it in a group. It also provides an incremental evaluation strategy so that repetitive computation of queries can be avoided.

D.Terry [3], had first provided the notion of Continuous Queries on Append-only databases, which restricts the evaluation of CQ only to append-only and also provides strategy to incrementally evaluate each query in order to get the most up-to-date result and avoids repetitive computation.

Simiarly goal of TriggerMan [11] system was also to create scalable triggers that identify unique experission signature and group predicate using trigger condition. The approach was very similar to the rules used in Active database systems using ECA (Event-Condition-Action) type rules.

Our research is different from above-mentioned work, because its main focus is on scalability problem and the efficient retrieval of information on the Internet from various data sources such as text, XML and database systems by providing high accuracy and data integrity.

Global optimization of queries is also one of the most active areas of research. [5],[6],[7], it provides an optimization strategy by exploiting similarities among the common sub-expressions present in a query and generates a global access plan by merging them together and showed that global optimization can significantly improve the performance as compared to the individual query execution plan.

We have some similarity in regards to the above-mentioned approaches but our proposed group optimisation strategy is different from the approaches for group optimisation discussed in [2],[5],[6],[7],[10] due to the following two aspects: 1) None

of them were on Continuous Queries, which are highly sensitive. 2) None of them were aimed for an environment like the Internet.

The major goal of this paper is to show some experimental analysis that can provide valid arguments on problems with the existing optimization techniques. It also shows the need for a new architecture that can solve the scalability and adaptability problems on the Internet.

8 Conclusions and Future Work

Our proposed model has significantly improved the scalability problem, which has not been addressed in the existing Continuous Query systems due to the following two reasons:

The global optimization approach, to minimize the cost of evaluating queries that involve large join operations.

A controlled trade-off mechanism that checks the specified condition and use the shortest access path that exists in the global query evaluation plan to evaluate query.

Our future work involves the optimization of CQ using Evolutionary approach, so it can support 1000's and millions of Continuous Queries efficiently.

The current CQ-System prototype is at preliminary stage and also needs some improvements in the design of the Client Interface. Time and event algebra is also not inculded in this paper on which we are currently working.

9 Web CQ-System Overview

The output from Web-based CQ-System written in Java, J2ee, XSQL Servlet using XML schema API and the Client-interface is shown above: A client can retrieve information using Web CQ-System Client interface. Triggers and timer can be added by clicking on the buttons shown in the fig.9. Actual processing strategy is also shown in the fig.9. Current simulation deals with four types of data sources such as: Oracle 9i, MS SQL-Server 2000, XML and text. As soon as the query passes through the web server, it is passed to the MQO optimizer for further processing, details about the grouping strategy is already discussed above. XSQL servlet formats the data in XML, XHTML etc before sending it to CQ-Scheduler, which then send result of queries to various clients on the Internet. Oracle XSQL template also enables to write dynamic XML data pages by traslating schema into required DTD. CQ-Scheduler can communicate directly with the group optimizer as well as with J2ee-Server depending on needs of the client. Triggers can dynamically added or deleted by clicking on appropriate button as shown in fig.9 on the next page.

References

1. Ling Liu, Calton Pu, W.,Tang,.: Continuous Queries for Internet Scale Event-Driven Information Delivery. IEEE Transactions on Knowledge and Data Engineering. Vol.

11 (1999) 610-628
2. J.Chen, David, J. ,D.,DeWitt, F.,Tian, Y., Wang.: NigaraCQ: A Scalable Continuous Query System for Internet Databases. Proceedings of the ACM SIGMOD International Conference on Management of Data. Dallas, TX, New York (2000) 379-390
3. Douglas Terry, David Goldberg, David Nichols, Brian Oki.: Continuous Queries Over Append-Only Databases. Proceedings of the 1992 ACM SIGMOD International Conference on Management of data, San Diego, California Vol. 21. (1992) 321-330
4. Ling Liu, Pu, C.,Barga, R.., T., Zhou, Differential Evaluation of Continuous Queries. Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS), ACM, New York (1996) 458 –465
5. Martin.,L.,.Keresten, M.F.N., Boer,.: Query Optimization Strategies for Browsing Sessions. Proceedings of 10th International Conference on Data Engineering, Houston Texas, USA (1994)
6. T.K.,.Sellis.: Multiple Query Optimization. ACM Transaction on Database Systems, Vol..13. (1988) 23 – 52
7. Fa.,Chung, Fred Chen, M.H.,Dunham.: Common Subexpression processing in Multiple-query Processing. IEEE Transaction on Knowledge and Data Engineering, Vol. 10. (1998) 493-499
8. Elizabeth J.O'Neil, P.E.O.,Neil, G.,Weikum.: The LRU-K Page Replacement Algorithm for Database disk buffering. Proceedings of the ACM Sigmod International Conference on Management of Data, Washington D.C., USA, (1993) 297– 306
9. P.,Roy, S.,Sudarshan, S.,Bhobe.: Efficient and Extensible Algorithm for Multi-Query Optimization. ACM SIGMOD Conference on Management of Data, TX, USA (2000) 249-260
10. On Multi-Query Optimization - Choenni, Kersten,, v., Den, Akker, Saad, Wiskunde en Informatica, REPORTRAPPORT J.F.P. van den Box 94079, 1090 GB Amsterdam, The Netherlands, Department of Computer Science, Alexandria www.cwi.nl/ftp/CWIreports/AA/CS- (1996) R9638.ps.Z
11. Hanson, E. N., Carnes, C., Huang, L., Konyala, M., Noronha, L., Parasarathy, S., Park, J., & Vernon, A.,: Scalable Trigger Processing. Proceedings of International Conference on Data Engineering, IEEE Computer Society, Los Alamitos, CA, March (1999) 266-275

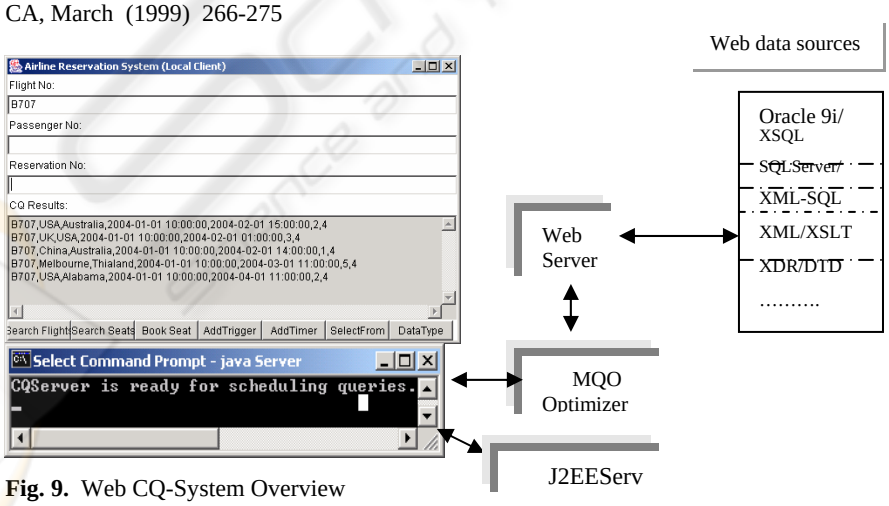

Fig. 9. Web CQ-System Overview