Differential Evaluation of Continual Queries

Ling Liu
University of Alberta
Department of Computing Science
GSB 615, Edmonton, Alberta
T6G 2H1 Canada
email: lingliu@cs.ualberta.ca

Calton Pu, Roger Barga, Tong Zhou
Oregon Graduate Institute of Science and Technology
Department of Computer Science
P.O. Box 91000 Portland
Oregon 97291-1000 USA
email: calton,barga,tszhou@cse.ogi.edu

Abstract

We define continual queries as a useful tool for monitoring of updated information. Continual queries are standing queries that monitor the source data and notify the users whenever new data matches the query. In addition to periodic refresh, continual queries include Epsilon Transaction concepts to allow users to specify query refresh based on the magnitude of updates. To support efficient processing of continual queries, we propose a differential re-evaluation algorithm (DRA), which exploits the structure and information contained in both the query expressions and the database update operations. The DRA design can be seen as a synthesis of previous research on differential files, incremental view maintenance, and active databases.

1 Introduction

An unprecedented amount of new information is becoming available by the exponential growth of the Internet [4]. However, on the one side this flood of information can easily overwhelm the users, and on the other side naive query processing can easily overload the database system and network. Both problems can be aggravated when constant addition and updates of information sources on the Internet force users to revisit Web sites and to re-issue their queries frequently to obtain the new data that match their queries. Trying to monitor the constant information flow in the Internet is both labor intensive for the user and resource intensive for the system. Unfortunately, traditional database techniques such as view materialization [3, 7], active databases [20], and query optimization, while powerful by themselves, encounter difficulties in distributed interoperable environments such as the Internet. The interoperability of these database techniques with each other and with the relatively unorganized storage and management of data, such as WWW pages [4] is completely non-trivial.

To address the interoperability problem in Internet environments we have developed the Distributed Interoperable Object Model (DIOM) [10], which supports a methodical and explicit composition of query results from interoperable information sources. The key idea of this paper is to generate new query results by incrementally updating previous query results, rather than re-evaluating the queries from scratch. This approach builds on our previous results from the Diorama architecture [10, 11] and Epsilon Serializability [24, 17] to support flexible and efficient monitoring of information updates in the Internet environments.

The first contribution of this paper is our use of DIOM and Epsilon Serializability to define a powerful mechanism for user specification of update monitoring. Epsilon specifications [17], a measure of distance in the database state space, is used in the marking of the triggering of query re-evaluation. Epsilon specifications allow the users to specify and activate query refresh based on the magnitude of updates, in addition to triggering query refresh periodically as proposed in previous works [23, 2]. An example of the new specification capability is the query "show the IBM stock transactions that differ by more than $5 from $75 per share" (Q3).

The second and more substantial contribution of this paper is an algorithm for incremental evaluation of continuous queries that supports general updates, including modifications, deletions, and inserts. This is a significant improvement over Continuous Queries [23] by Terry et al., which restricts database updates to append-only, disallowing deletions and in-place modifications. Besides the support for general updates, our differential re-evaluation algorithm (DRA) reduces data transmission by processing the query primarily on the changed data, rather than re-evaluating the entire database. We also show that the differential re-evaluation of these queries is functionally equivalent to the complete re-evaluation solution.

We use the relational model terminology and concepts in the design and description of the DRA for clarity and simplicity. It should be pointed out that the DRA itself takes as input the updates from different information sources. These updates are described as differential relations in this paper, as differential relations have a very simple form and content for representing updates in terms of modifications, insertions, and deletions. Therefore, simple translators (as part of DIOM) will be able to support the integration of information sources other than relational databases. For example, file system updates can be captured by either operating system or middleware and translated into a differential relation and fed into DRA. This is in contrast to the conceptual difficulties in the integration of active databases and view materialization, as well as the practical difficulties of implementing these powerful database techniques on non-database environments such as file systems.

Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)
1063-6927/96 $5.00 © 1996 IEEE
Proceedings of the 16th ICDCS 450
2 Related Work

There has been considerable research done in the monitoring of information change in databases. The following discussion should not be seen as a critique of these techniques. Rather, these techniques have been proposed primarily for "data-centric" environments, where data is well organized and controlled. When applied to an Information Superhighway environment (for brevity and concreteness we refer to this environment as the Internet), these assumptions no longer hold (see [10] for a summary of desired system properties in the Internet). Therefore, we discuss these assumptions before we launch into our definition of continual queries (Section 3) and the DRA technique (Section 4.2) for efficient processing of continual queries.

Continuous Queries: Terry et al [23] proposed continuous queries for monitoring information change. One of the significant assumptions that their proposal made is the limitation of database updates to append-only, disallowing deletions and modifications. Since this assumption is used in their query transformation algorithm, it has been difficult to relax it [2], when following their definition of continuous queries. This is one of the motivations for our new definition in Section 3 under a new name, continual queries.

Active Databases: Rules are used extensively in active databases [5, 14, 18, 20] for monitoring changes of database state. Despite their conceptual generality, rules have been so far supported in a fairly restrictive form in practical systems, for example, by triggers in relational database management systems such as Oracle, Sybase, and Informix. A trigger is an event--condition--action (ECA) rule in a restrictive form. Typically, triggers can be specified only on a single base table. Active query, introduced in Alert [20], is a yet another form of ECA rules, similar to continuous queries [23], and usually assumed to work in an append-only environment. Active queries are more sophisticated than triggers, since they can be defined on multiple tables, on views, and can be nested within other active queries. However, like continuous queries [23], the append-only assumption seriously restricts the applicability of active queries to the Internet environment where data is appended, removed, or replaced constantly.

Materialized Views: Materialized views store a snapshot of selected database state. When a database is updated, the materialized view must be refreshed to reflect the updates. Three approaches have been described previously. The first approach refreshes the view immediately after each update to the base table [3]. The second defers the refresh until a query is issued against the view [9]. The third refreshes the view periodically [8]. The main tradeoff in choosing among these approaches is the staleness of the view data vs. the cost of updating it. Most of the algorithms in the literature [3, 9, 7, 8, 6] work in a centralized database environment, in which the materialized view and its base tables co-reside. The study on distributed materialized view management has been primarily focused on determining the optimal refresh sources and timing for multiple views defined on the same base data [22, 21]. Other works on distributed environments include quasi-copies for replication [1] and update anomalies in data warehouses [25].

3 Continual Query Semantics

3.1 Basic Semantics

A continual query CQ is a triple \((Q, TCQ, Stop)\), consisting of a normal query \(Q\) (e.g., written in SQL), a triggering condition \(TCQ\) as specified below, and a termination condition \(Stop\). (Since \(TCQ\) and \(Stop\) in general may depend on many different parameters, we usually omit their parameters for clarity.) Let us denote the result of running query \(Q\) on database state \(S_i\) as \(Q(S_i)\). We define the result of running a continual query \(CQ\) as a sequence of query answers \(\{Q(S_1), Q(S_2), \ldots, Q(S_n)\}\) obtained by running query \(Q\) on the sequence of database states \(S_i, 1 \leq i \leq n\), each time triggered by \(TCQ\), i.e., \(VS_i, TCQ \land \neg Stop\).

We assume that there is a mapping from the sequence of database states \(S_i, 1 \leq i \leq n\) into a sequence of monotonically increasing timestamps \((t_1, t_2, \ldots, t_n)\), i.e., when given a state \(S_i\) we can refer to its time \(t_i\). Each \(CQ\) has its own sequence of database states, defined by the triggering condition \(TCQ\). The triggering condition \(TCQ\) is a specification of when the query \(Q\) should be executed. It can take several forms:

- A direct specification of time. For example, \(Q\) should be executed once every Monday. This is the policy adopted by the Harvest information gatherers [4].
- A specification of time interval from a previous query result. For example, "a week since \(Q(S_{n-1})\) was produced". (This can be interpreted as a time specification relative to an event.)
- A condition on the database state. For example, \(Q\) should be executed whenever a deposit of one million dollars is made. This is supported by epsilon specifications (Section 3.2).
- A relationship between a previous query result and the current database state. For example, \(Q\) should be executed when a total of 1 million dollars worth of deposits have been made since \(Q(S_{n-1})\) was produced. This is also supported by epsilon specifications (Section 3.2).

If the termination condition \(Stop\) is nil, \(CQ\) will produce results from \(Q(S_1)\) to \(Q(S_n)\). Otherwise, \(CQ\) will produce results from a starting time \(t_1\) to a final time \(t_n\), when \(Stop\) becomes true. In other words, \(CQ\) (the sequence) ends when the termination condition becomes true.

Each time \(TCQ\) is triggered, conceptually \(Q\) is run against the current state of the database \(S_i\), and the result \(Q(S_n)\) is sent to the user who issued \(CQ\). In general it is expensive to re-evaluate the whole query over the entire source data for each execution of a continual query, although in some circumstances (e.g., legacy databases and some file systems) it may be unavoidable to reprocess the query from scratch. Therefore, it is important to find optimization steps that can bypass the complete re-evaluation and thus avoid the duplicate computation and unnecessary data transmission. In Section 4.2 we describe a strategy to generate \(Q(S_n)\) from \(Q(S_{n-1})\) incrementally, thus reducing both processing time and network transmission bandwidth.
3.2 Epsilon Specifications

Epsilon specifications have been introduced as part of the work on Epsilon Serializability (ESR) [15, 16, 17]. A simple example of an epsilon query is the checking account sum-up query "a bank manager wants to know how many millions of dollars she has in all the checking accounts" with the error tolerance rate of half a million. Since the round-off error of the query is half a million dollars, the query could contain other errors (say due to non-serializable conflicts with some updates) up to half a million dollars, and still return a meaningful result. The specification of the error tolerance, in the above example, i.e., half a million dollars, is called an epsilon specification ($\varepsilon$-spec).

Although ESR was originally proposed for transaction processing, we have found an interesting application of epsilon queries in the context of CQs. We can model an epsilon query as a pair $(Q, E$-spec), where the $E$-spec is used as the triggering condition $T_{CQ}$. This way, we can define a CQ from an epsilon query plus a termination condition. Intuitively, the original epsilon query was defined to allow a bounded amount of inconsistency to be introduced into the query result. For example, divergence control algorithms [24, 15] allow limited non-serializable conflicts between updates and the epsilon query to happen, to increase system execution flexibility and concurrency. In a CQ environment, we can consider the $E$-spec as the triggering condition that bounds the distance between the previous element in the sequence of CQ results to the next. Each time the current CQ query result is about to violate the $E$-spec, the system generates the next CQ result in the sequence. For example, suppose the last execution of the checking account sum-up epsilon query returned a result of $125$ million. If the accumulated amount of withdrawals from, and deposits to, the checking accounts have exceeded half a million, then the system generates the next sum-up result. Instead of re-evaluation of complete sum-up query, a differential re-evaluation algorithm (DRA) described in Section 4.2 can be used. If a triggering condition or a termination condition contains references to the database state, then updates in the database need to trigger the evaluation of these conditions.

3.3 Differential Processing of CQs

Although we have defined CQ as a sequence of query results, there are situations where users are more interested in the difference between $Q(S_i)$ and $Q(S_{i+1})$. Typically, when $Q(S_i)$ is relatively large, and only a small percentage of the result changes from $Q(S_i)$ to $Q(S_{i+1})$, then users would want to have the difference emphasized. This can be accomplished by naively executing the entire query and then filtering out the part of the query result that is the same as the previous result. This simple and straightforward approach can be quite expensive, especially in the Internet environment where query results need to be gathered from multiple source data repositories. This is a motivating example for differential query processing and result presentation.

Using differential re-evaluation (Section 4.2), we produce $Q(S_{i+1})$ by incrementally updating $Q(S_i)$. This way, we avoid reprocessing the entire query from scratch. When the changes are substantially smaller compared with the latest query execution result, this differential update will be more efficient than reprocessing the entire query.

Since a differential update is performed on top of $Q(S_i)$ to produce $Q(S_{i+1})$, we assume that the underlying system maintains a copy of $Q(S_{i+1})$, i.e., it is not deleted as the result is scrolled off the user's screen. The copy is maintained at the site where the differential query refresh is carried out. If the user is interested only in being notified of the difference between the current query result and the next query result (called differential query result), then we may skip the maintenance of a copy of the complete result of the previous query. But if the user requests a complete query result, we may have to reprocess the query upon such requests. The trade-off in choosing the best strategy is analogous to the optimization trade-off in choosing between complete system backups and incremental backups.

4 Evaluation of CQs

In this section, differential relations are used to represent the net effect of a collection of updates to a relation, either stored or derived. The differential relation for a stored relation is instantiated by the system when the source is updated by insertion, deletion or modification (see Example 1). We discuss the operator, namely Propagate, that computes differential relations for arbitrary SPJ expressions. We prove that the processing of a continual query, after its initial execution, can be reduced to the evaluation of the differential form of the query. We assume that readers are familiar with the basic concepts and notations concerning relational database, as described in [13].

4.1 Differential Relations

We introduce the concept of differential relation, a relation that can represent changes to another relation, either stored or derived. The differential relation for a stored relation is instantiated by the system when the source is updated by insertion, deletion or modification (see Example 1). We discuss the operator, namely Propagate, that computes differential relations for arbitrary SPJ expressions. We prove that the processing of a continual query, after its initial execution, can be reduced to the evaluation of the differential form of the query. We assume that readers are familiar with the basic concepts and notations concerning relational database, as described in [13].

Example 1 Consider the relation Stocks with attributes such as name and price per 100 units.
Assume that the transaction T updates the Stocks relation by insertion, deletion and modification:

Begin Transaction T
Insert (101088, MAC, 117);
Modify (120992, DEC, 150) = (120992, DEC, 149);
Delete (092394);
End Transaction

The following differential relation \( \Delta \text{Stocks} \) captures the changes that the transaction T made to Stocks. The operators insert(\( \delta R \)) and delete(\( \delta R \)) are used to compute the objects that are newly inserted into the base relation \( R \) and the objects that are recently deleted from the base relation \( R \) respectively.

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{tid}^{\text{old}} & \text{Name}^{\text{old}} & \text{Price}^{\text{old}} & \text{tid}^{\text{new}} & \text{Name}^{\text{new}} & \text{Price}^{\text{new}} & \text{ts} \\
\hline
100000 & DEC & 156 & 101088 & MAC & 119 & 10 \\
092394 & QLI & 145 & - & - & - & - \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{deletion}(\Delta \text{Stocks}) \\
\hline
\text{tid} & \text{Name} & \text{Price} \\
120992 & DEC & 149 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{insertions}(\Delta \text{Stocks}) \\
\hline
\text{tid} & \text{Name} & \text{Price} \\
101088 & MAC & 117 \\
120992 & DEC & 149 \\
\hline
\end{array}
\]

Differential relations can represent tuples (or objects) where only the tid field is nonnull. No tid can appear in multiple rows. For addition, the attributes \( A^{\text{old}} \) (\( 1 \leq i \leq n \)) are null. For deletion, the attributes \( A^{\text{new}} \) are null. For modification, attributes \( A^{\text{old}} \) hold old values and attributes \( A^{\text{new}} \) hold newly modified values. The timestamp field \( \text{ts} \) set to the current time (from a system clock, or any other monotonically increasing source of timestamps) whenever a tuple is appended to \( \Delta R \).

The concept of differential relations is, to some extent, similar to the concept of hypothetical relations used for incremental updating materialized views[3, 7]. The difference lies in the usage and the detailed structure. In the eager mechanism for materialized view update, a hypothetical relation represents the net changes made by a single transaction to a base relation and can be dropped after the transaction is committed and the materialized view is updated. In the continual query refresh method, a differential relation actually maintains changes made by several transactions to a base relation. Data in the differential relation can only be dropped when their timestamps are "older" than the timestamp of the latest execution of every relevant continual query.

We would like to mention also that the relational model is not essential to our approach, but it simplifies the representation of database changes, allows use of the relational algebra, and avoids the need to explain the semantics of a particular object model.

4.2 Informal Overview

The differential re-evaluation algorithm (DRA) is designed for processing a continual query efficiently. In contrast to a complete re-evaluation, differential re-evaluation means that after the initial execution of a CQ, the re-evaluation of each subsequent execution of this CQ will be performed by using the differential form of the query. The DRA is invoked by the CQ manager based on the proper timestamp predicate(s) of the query associated with the given CQ. We assume that the information available when the DRA is invoked includes: (i) the CQ definition; (ii) the contents of each base relation after the last execution of the CQ; (iii) the differential relations of each of those operand relations that have been changed since the last execution of the CQ; (iv) the timestamp of the last execution of the CQ; (v) the complete set of the result of the CQ produced by the last execution. Note that the CQ manager will use (iv) to append the proper timestamp predicate(s) into the differential form of the CQ, which limits the search space to only those tuples that were written to the differential relations by the updates occurred after the last execution of this CQ (recall Example 2 for an illustration). We also formally study the correctness of the DRA with respect to the complete re-evaluation solution.

To ensure that the DRA algorithm is functionally equivalent to the complete re-evaluation solution, we formally define the operator Diff to compute the difference of two relations of the same scheme, and introduce a high level operator based on Diff, namely propagate, to describe how the result relation of a continual query \( \text{Q} \) changes when at least one of its operand relations changes. This operator computes the difference between two consecutive executions of a CQ by complete re-evaluation of the query for each execution. It can be considered as an instantiation of complete re-evaluation solution. The main purpose of introducing the operator Propagate is to formally prove that our differential re-evaluation algorithm to continual queries is functionally equivalent to the "recompute the query from scratch" solution, and, in many situations is more efficient. We first formally introduce the differential forms for the three basic relational algebraic operations: Select, Project, and Join. Then we prove that instantiation of Propagate for relational select, project, and join are functionally equivalent to their differential forms: DiffSelect, DiffProj and DiffJoin. Therefore, the processing of a continual query, after its initial execution, can be reduced to the evaluation of the differential form of this query.

Due to the space limitation, instead of presenting the formal development of these operators and proofs, we below provide some examples to illustrate some of the main operators. Readers may refer to [12] for further detail.

Example 2 Consider query \( \sigma_{\text{stock} \leq 120} (\text{Stocks}) \) as a continual query \( \text{Q} \). Let \( \text{E}_i \) be the \( i \)th execution of \( \text{Q} \) at time \( t_i \) and \( \text{E}_i (\text{Q}, t_i) \) denote the result of the \( i \)th execution of \( \text{Q} \) \( (i = 0, \ldots, \infty) \). Now assume that the base relation Stocks is changed, after the last execution \( \text{E}_i \) and before the current execution \( \text{E}_{i+1} \), by the update transaction T given in Example 1. Let \( \text{Q}(\text{Stocks}) = \text{E}_i (\text{Q}, t_i) \) denote the result of the \( i \)th execution of \( \text{Q} \) over Stocks. Let Stocks \( \ast \) denote the base relation after updates to Stocks by transaction T, and \( \text{Q}(\text{Stock} \ast) = \text{E}_{i+1} (\text{Q}, t_{i+1}) \) denote the result of the current execution of \( \text{Q} \) over the relation Stocks \( \ast \).

Based on the definition of Propagate, to express
how the result $E_i(Q,t_i)$ may change after the updates by the transaction $T$, we may simply compute $\text{Propagate}(Q(\text{Stocks});[\text{Stocks}, \Delta \text{Stocks}])$, i.e., the difference between the result relation before the updates: $Q(\text{Stocks})$ and the result relation after the update: $Q(\text{Stocks}')$.

$$Q(\text{Stocks}) = \sigma_{\text{price}>120}(\text{Stocks})$$

$$= \{(120992, \text{DEC}, 150), (992394, \text{OLL}, 145)\}.$$  

$$Q(\text{Stocks}') = \sigma_{\text{price}>120}(\text{Stocks}')$$

$$= \{(120992, \text{DEC}, 149)\}.$$  

$$Q(\text{Stocks}) - Q(\text{Stocks}') = \{(120992, \text{DEC}, 149)\}.$$  

The differential result $\Delta Q(\text{Stocks})$ below represents the net effect made by all the updates occurred between the last execution $E_i$ and the current execution $E_{i+1}$.

$$\Delta Q(\text{Stocks}) = \text{Diff}(Q(\text{Stocks}), Q(\text{Stocks}'))$$

In this example, the differential result of the query $\sigma_{\text{price}>120}(\text{Stocks})$, since the last execution of $Q$, is computed by directly evaluating the function: $\text{Propagate}(\sigma_{\text{price}>120}(\text{Stocks});[\text{Stocks}, \Delta \text{Stocks}])$. Assume the previous execution result $E_i(Q, t_i)$ is saved. The evaluation of $\text{Propagate}$ directly from its definition requires first a scan of the relation Stock $'$ in order to compute $Q(\text{Stocks}')$. Such recomputing from scratch is often wasteful, and in many cases unacceptable.

Observe that when (i) the size of the source relation $\text{Stocks}$ is large, (ii) the selectivity factor of the query $Q$ over $\text{Stocks}$ is not high, and (iii) the number of tuples updated by the updates, occurred between the two consecutive executions $E_i$ and $E_{i+1}$, is relatively small, it would be more efficient to compute the differential result of the query $Q$ before and after the updates by evaluating the query over the differential relation $\Delta \text{Stocks}$ instead. It is because computing $\text{Propagate}(\sigma_{\text{price}>120}(\text{Stocks});[\text{Stocks}, \Delta \text{Stocks}])$ is functionally equivalent to the evaluation of the differential query: $\sigma_{F}(\Delta \text{Stocks})$, where $F$ denotes the predicate $\text{price}_{\text{old}} > 120 \land \text{price}_{\text{new}} > 120 \land t > t_i$. This is true even when the user needs the complete composite set of the query result, because computing the union of $Q(\text{Stocks})$ and the differences $\sigma_{F}(\Delta \text{Stocks})$ is cheaper than recomputing the expression $\sigma_{\text{price}>120}(\text{Stocks}')$ from scratch.

Furthermore, using a differential evaluation approach, we can show those tuples that were removed between the two consecutive executions of $Q$, simply by computing $\text{deletions}(\sigma_{F}(\Delta \text{Stocks}))$. In general, for any continual query $Q$ over the relation $\mathcal{R}$, let $E_i(Q,t_i)$ be the last execution of $Q$ at time $t_i$. A complete set of the result of current execution of $Q$ can be obtained by computing the expression:

$$\text{Et}_i(Q,t_i) - \sigma_{t_i>t_j}([\text{deletions}(\Delta R)])$$

The expression $\sigma_{t_i>t_j}([\text{deletions}(\Delta R)])$ returns all the records that have been appended to $\mathcal{R}$ since the last execution of $Q$; whereas the expression $\sigma_{t_i>t_j}([\text{insertions}(\Delta R)])$ returns all the records removed from $\mathcal{R}$ since the last execution of $Q$.

### 4.3 The Differential Re-evaluation Algorithm

We now outline an algorithm for re-evaluating continual queries (limited to SPJ expressions) differentially.

**Algorithm 1 (The DRA algorithm)**

**Input:**

- the SPJ definition of the continual query $Q$, i.e., $Q = \pi_x(\sigma_F(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n))$, where $\pi_x$ denotes the projection list and $F$ denotes the selection predicate over $R_1, \ldots, R_n$;
- the contents of the base relations $R_i$ $(1 \leq i \leq n)$ after the last execution of the CQ;
- the differential relations $\Delta R_i$ $(1 \leq i \leq n)$;
- the timestamp of the last execution of this CQ, say $E_i$;
- the complete set of the result produced by the last execution of the CQ.

**Output:** result of the current execution of query $Q$.

**Procedural Steps:**

1. Build the truth table $T$ with $k$ columns $(k \leq n)$ and $p$ rows, $p = 2^n$. Each column is corresponding to a relation in the SPJ expression, which has been changed since the last execution of $Q$.
2. For each row $i$ $(1 \leq i \leq p)$ of the table $T$, construct the associated SPJ expression, by substituting $R_i$ in $Q$ with $\Delta R_i$ when the binary variable $T;j = 1$. For each of these SPJ expression, denoted by $G = S_1 \bowtie S_2 \bowtie \ldots \bowtie S_n$, evaluate $G$ by its differential form $\text{DJoin}(S_1, S_2, \ldots, S_n)$.
3. Perform the union of the results obtained from each computation in Step 2.
4. Based on the epsilon specification of the CQ, assemble the final set of the result to be returned to the users. ⚫

For example, let $\Delta R_Q$ denote the result generated by Step 3.

- If the user wishes to see only the differential result since the last execution of the $Q$, say $E_i(Q,t_i)$, without deletion notification, the result to be returned can be computed by $\sigma_{t_i>t_j}([\text{deletions}(\Delta R)])$.
- If the user needs to see the complete set of the result matching the query, we return $E_i(Q,t_i) \cup \sigma_{t_i>t_j}([\text{insertions}(\Delta R)])$.
- If the user wants to be notified all the deleted tuples since the last execution of the CQ, we simply compute the $\sigma_{t_i>t_j}([\text{deletions}(\Delta R)])$.
5 Discussion

5.1 Strawman Performance-Arguments

Although there is no space in this paper for a detailed performance analysis, we argue informally that there are many important scenarios in which DRA wins over algorithms that operate on the base data instead of results.

First of all, we observe that the overwhelming majority of queries return a table of results that is much smaller than the base data. (Otherwise, the query would be considered not selective enough and the results not particularly useful.) Therefore, DRA processing of the next query execution on top of results will be much faster, reducing both I/O and CPU requirements and communication overhead. In general, caching the results on the client side makes the servers more scalable with respect to the number of clients.

Second, since results are combined from many sources into a local table, DRA processing of results will avoid both translation from the base data to an interoperable format. Moreover, if the volume of relevant updates is smaller than the results (which is the common case), then we are further reducing the network traffic.

Third, each server only generates delta relations when communicating with the clients. This is easier for interoperating than trying directly to integrate active databases and materialized views. To the best of our knowledge, there are no practical methods for combining them in a heterogeneous environment.

On the other hand, we note some limitations of the DRA algorithm. For example, when the results turn out to be large (poor selectivity of the query), then a lazy evaluation and transmission of results is necessary. Another important assumption of the DRA algorithm is the availability of delta relations from every information producer. This may not be trivial for legacy databases. But as we mentioned before, there is no easy way to integrate legacy active databases or materialized views, either.

5.2 Query Refinement

First, we should test the CQ condition based on the differential updates before every execution. If the updates occurred in between of the two consecutive executions have no impact on the previous query result set, we consider them as irrelevant updates to the continual query. Thus, no computation is performed for this CQ, because in this case, nothing needs to be returned if the user is only interested in the differential result. When the user asks for the complete answer, we simply return the result of previous execution.

In addition, for each SPJ expression in step 2 of the DRA algorithm, it is necessary to determine its execution strategy. One way to find a good execution strategy is simply to use the heuristics such as Select before Join, extracting common subexpressions, cheaper selection predicate before expensive ones). This approach might be most appropriate if one does not have access to an appropriate query optimizer. An alternative approach is to have a DBMS query optimizer generate the strategies. The differential form of a query can be regarded as a query to a database that consists of base relations and differential relations.

5.3 Asynchronous Evaluation of CQ Conditions

There are two general strategies concerning when to test the TCQ. The first is to evaluate TCQ immediately after each update to the source data (base relations). The second is to evaluate TCQ periodically. In the Diorama system, both can be implemented. For concreteness we discuss the second choice here. Consider the checking account example discussed in Section 3.2: a bank manager wants to know how many millions of dollars she has in all the checking accounts. This query Q can be expressed in SQL as: SELECT SUM(amount) FROM CheckingAccounts. Suppose it is installed as a continual query with the following TCQ (and C-spec): TCQ = \( |\text{Deposits} - \text{Withdrawals}| \geq 0.5M \) and Stop: nil. Deposits and Withdrawals denote the total amount of deposits and withdrawals since the previous execution of the query, respectively. Since there is no specific time in the C-spec of this query, the CQ manager can decide when to evaluate TCQ by a system-defined default interval, say every day at midnight. If the C-spec is greater than half a million, the query Q is executed.

In the Diorama system, the differential evaluation method will be used to verify TCQ whenever possible. In this example the CQ manager first rewrites TCQ, originally specified as: \( |\text{Deposits} - \text{Withdrawals}| \geq 0.5M \), which requires testing of the base relation CheckingAccounts. The new TCQ has a differential form that requires only scanning \( \Delta \text{CheckingAccounts} \):

- \( \Delta \text{Deposits} := \text{SELECT SUM(amount) FROM insertions(\Delta \text{CheckingAccounts}) WHERE ts > t_{i-1}.} \)
- \( \Delta \text{Withdrawals} := \text{SELECT SUM(amount) FROM deletions(\Delta \text{CheckingAccounts}) WHERE ts > t_{i-1}.} \)

where \( t_{i-1} \) denotes the timestamp of the previous execution \( Q(S_{i-1}) \). In general, the cost of evaluating the differential form of TCQ is cheaper than the complete reevaluation of TCQ over the entire base relations. In this example, it is true when \( |\text{CheckingAccounts}| > |\Delta \text{CheckingAccounts}| \).

5.4 Garbage Collection of Differential Relations

As the source data changes, their differential relations grow accordingly. To keep the differential relations to a bounded size, we need to garbage collect the portions of the differential relations that are no longer useful. The technical details of the solution are beyond the scope of this paper. We outline the basic idea here. First let us consider the case of a single active CQ in the system. Each time a
new query result $Q(S_t)$ is obtained, we can retire the differential relations referring to database states prior to $t_i$. This is intuitively easy to understand since only the data in the differential relations with the timestamps later than $t_i$ will be needed for the processing of $Q(S_{i+1})$. Informally, we call these portions of the differential relations "active delta zone".

With multiple active CQs in the system, the garbage collection algorithm is an extension of the basic idea outlined above. For each CQ, we define its active delta zone. For the whole system, we define the system active delta zone as the union of the active delta zones of all CQs. Assuming that each CQ will make progress, its active delta zone will move forward in time. The system active delta zone will move forward as a consequence, with its boundary delimited by the "oldest" active delta zone. All the data in the differential relations that fall outside the system active delta zone can be garbage collected, since they will not be used by any active CQ.

5.5 Generation of Delta Relations

It should be pointed out that, although we use the relational model terminology and concepts in the design and description of the DRA for clarity and simplicity, the DRA itself takes as input the updates from different information sources. These updates are described as differential relations in this paper, as differential relations have a very simple and clear form and content for representing updates in terms of modifications, insertions, and deletions. For the relational information source providers, the generation of different relations is quite straightforward. For those information sources other than relational databases, simple translators (as part of the DIOM services [11]) will be used to extract the updates in the form of differential relations. For example, file system updates can be captured by either operating system or middleware and translated into a differential relation and fed into DRA. This is in contrast to the conceptual difficulties in the integration of active databases and view materialization, as well as the practical difficulties of implementing these powerful database techniques in non-database environments such as file systems.

6 Conclusion

Information monitoring in the Internet is an increasingly important research topic due to two conflicting issues. On the positive side, the exponential growth of Internet makes exponentially more data available and information monitoring similarly more rewarding. On the negative side, the same growth makes the USECA properties an urgent necessity (uniform access, scalability of query system and run-time performance, evolution of databases and applications, composability of interface schemas and software, and autonomy of both information producers and information consumers).

In this paper, we introduce a simple new definition of Continual Queries (CQ), where each CQ is a sequence of query executions. Then we describe a Differential Re-evaluation Algorithm (DRA) for incrementally computing the new query result from processing updates on top of the previous result. We prove that our differential re-evaluation algorithm to continual queries is functionally equivalent to the "recompute the query from scratch" solution, and, in many situations is more efficient. In addition to a formal description of differential relations (Section 4) and the DRA (Section 4.2), we also discuss a number of implementation issues such as asynchronous evaluation of CQ conditions and strategies for garbage collection of differential relations (Section 5).

DRA and CQ satisfy the USECA properties for information update monitoring on the Internet. Uniform access and composability are supported by DIOM [10], the foundation of our work. Scalability is supported by DRA, which encourages the shifting of the processing to the client side. Evolution and autonomy are preserved by the combination of DIOM and CQ capabilities, such that changes made by information producers on the one hand are orthogonal to the information consumer's query requests, and on the other hand can be automatically captured and passed on to the information consumers by using CQ capability.

Although we have addressed both the positive and negative aspects of differential evaluation of continual queries for large-scale information monitoring, by no means have we exhausted the topic. Many interesting challenges remain in the area of information monitoring on the Internet, for example, other algorithms for differential or incremental evaluation of CQs, a rigorous performance evaluation of the query processing techniques, and the implementation of the research ideas into actual software tools. Our current research is focused on the refinement of algorithms and their implementation.

Acknowledgement

The work was supported partially by NSERC grant OGP-0172659 and NSERC grant ST-0181014 for the first author, and by ARPA under contract DABT63-95-C-0010 and NSF under grant IRI-9510112 for the second author.

References


