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,roger,tong@cse.ogi.edu

Abstract

Information Superhighway environments such as the Internet have brought us ready access to large amount of information. However, Internet data is notoriously unorganized and autonomously managed in a distributed fashion. Large scale information monitoring in the Internet environment requires support beyond traditional database techniques. Two of the key issues are the increasing reward in monitoring a fast growing information base and the similarly increasing processing cost.

To improve the expressiveness of queries for information monitoring, 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 [27], continual queries include Epsilon Transaction [18, 19] 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.

Keywords: Information Superhighways, Internet, information monitoring, continual queries, differential data, incremental query processing, epsilon serializability.

1 Introduction

The exponential growth of the Internet [4] makes an unprecedented amount of new information available. Large scale information monitoring has become tantalizingly close to reality. However, this flood of information can easily overwhelm the users, if queries are formulated naively. Furthermore, naive query processing can easily overload the database system and network. For example, a user monitoring new multimedia research may ask the system to "show all the multimedia projects described in WWW pages of universities and companies" (Q_1); this is a query that may return a tremendous amount of information, if the query is submitted to all the `.edu` and `.com` sites running WWW servers. The problem can be aggravated when constant addition and updates of information sources on the Internet force users 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. Therefore, the support for flexible and efficient monitoring of information updates is an important goal when accessing data in the Internet.

Traditional database techniques such as view materialization [3, 8], active databases [22], and query optimization cannot be directly applied in distributed interoperable environments such as the Internet. The reasons include the lack of integration of information sources, autonomy and heterogeneity in the Internet, and the relatively unorganized storage and management of data, such as WWW pages [5]. A promising proposal is Terry et al's *continuous queries* [27], which can be processed incrementally to reduce both system overhead and user overload. However, the severe assumptions they made about their environment, such as append-only databases with no deletions, render their work unsuitable for many kinds of queries and information sources in the Internet.

The challenges presented by the Internet starts with the formulation of queries that express the different ways users want to be notified of the information changes. Once queries are formulated, optimization in monitoring, processing, and transmission is needed. After these necessary two steps, integration of heterogeneous data sources, legacy databases, file systems and unorganized data are important for enlarging the data available. Further refinements include approximate queries, partial matching and correlations, and other deep information semantics.

In this paper, we focus our attention on the first two issues. We make two contributions towards flexible and efficient monitoring of information updates, motivated by a distributed interoperable environment such as DIOM [12]. First, we describe continual queries with epsilon specification, namely *epsilon continual queries* (ECQ), to simplify continual query formulation on the user side. In addition to triggering query refresh periodically as proposed in previous work [27, 2], ECQs allow the users to

specify and activate query refresh based on the magnitude of updates. For example, the query "show all the new multimedia projects and those that have their project description text changed by more than 10% since last week" (Q_2) would bring more significant information to the user monitoring new multimedia research, compared to Q_1 above. Another example of new ECQ capability is the query "show the IBM stock transactions that differ by more than \$5 from \$75 per share" (Q_3).

Our second contribution is aimed at reducing query processing cost. We develop a differential strategy to evaluate ECQs incrementally, and prove that the differential re-evaluation of a ECQ is functionally equivalent to the complete re-evaluation solution. This *differential re-evaluation algorithm* reduces data transmission by processing the query primarily on the changed data, rather than re-evaluating the entire database. Furthermore, differential re-evaluation supports general updates, including insertions, deletions, and modifications. This is a significant improvement over the no-deletion assumption in [27].

The rest of the paper is organized as follows. In Section 2, we discuss the motivations of our work and the related research. Section 3 presents the semantics of *continual queries* with epsilon specification. Section 4 summarizes the notations and terminology used in the processing of continual queries. Section 5 describes a method for differentially evaluating continual queries. Section 6 discusses some implementation issues and topics of future research. Section 7 concludes the paper.

2 Motivation and 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 [13] 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 a technique (Section 5) for efficient processing of continual queries.

2.1 Continuous Queries

Terry et al [27] proposed continuous queries for monitoring information change. However, their proposal made several assumptions that seriously restricted the applicability of their technique to the Internet. The first such assumption 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.

Another problem discussed in [27] is the non-determinism in continuous query results, where two users with exactly the same continuous query expression could be presented with different results. Consider the query for a product-purchase database: "select the products which have received no orders so far." Whenever a new product is inserted, it matches the query until an order for this product arrives. Suppose a particular product, say "Simpson Soap", is inserted into the database at 9:10 and an order for it arrived at 9:50, then the query would not return "Simpson Soap" if the continuous query runs every hour on the hour, but "Simpson Soap" would be returned by a system that runs it every hour on the half hour since the product would match the query at 9:30.

In our view, the main cause of this non-determinism comes from the different database states due to updates, which affect query results as they should. Therefore, instead of rejecting it as an anomaly, we capture it. In Section 3, we define a continual query as a sequence of executions of a conventional database query, which is re-executed when a triggering condition becomes true. Two continual queries with the same query expression and triggering conditions should observe the same query results over time, while continual queries with the same query expression but different triggering conditions should be considered different. Other advantages of our semantics will be discussed in Section 3.

2.2 Active Databases

Active databases [6, 15, 20, 22] allow users to specify, in the form of rules, actions to be performed following 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 [9] 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 [22], is yet another form of ECA rules, similar to continuous queries (Section 2.1), 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 [27], the append–only assumption seriously restricts the applicability of active queries to the Internet environment where data is appended, removed, or replaced constantly.

2.3 Maintenance of Materialized Views

Another related area of research is the incremental maintenance of materialized views, which store a snapshot of selected database state. As a database is updated, the materialized view must be refreshed to reflect the updates. A naive solution is to rematerialize the view from the base data. In contrast, incremental update algorithms are believed to carry lower execution cost if changes to the database are moderate [8, 11]. 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 [21]. The third refreshes the view periodically [11]. 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, 11, 8, 10, 7] 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 [24, 23].

In contrast to database view work, which assumes well organized data and schema, as well as system-defined periodic refresh, continual queries should give users more flexibility in specifying the conditions for refresh and tighter control over the distance between "view result" and the current database state. This is similar to the specification of update propagation in quasi-copies [1].

A recent paper [29] has identified the problem of using incremental view maintenance approaches in a data warehousing environment,[1] where the materialized view and its base relations are loosely coupled. While the warehouse attempts to update a view, the base data at source may keep changing. An anomaly may occur whenever the warehouse performs view update operations using a state of the base relations which differs from the state at which this view update was issued. An eager compensating algorithm is developed in [29] to address this problem. Other possible solutions include locking the relevant data at source while the results of CQs are being refreshed, or the source maintains timestamps for its data.

2.4 Research Challenges

An important distinction between CQs and previous work is the handling of general updates, including insertions, deletions, and modifications. For example, let us assume that a job opening database is updated whenever a posted position is filled (deletion) or a new job posting arrives (insertion). Some

[1] A warehouse is a repository of integrated information gathered from remote data sources for queries and analysis (e.g., decision support, or data mining).

users may desire to be informed of both the incoming new openings from some particular type of companies and the positions that were available before (e.g., in the previous execution of the query) but have been recently filled or canceled due to budget disapproval. Simply recomputing the query for each execution does not guarantee the correct results, because conventional SQL query manager can only filter data *currently matching* the query. To provide adequate support for flexible incorporation of updates, the CQ manager must exploit the knowledge implied by both the query expressions and database update operations.

Another important problem in monitoring of information updates is the cost of query evaluation. A naive query evaluation strategy would recompute the same query from scratch for each execution request and highlight the difference. In the Internet environment, such strategy can easily overwhelm the users and overload the system and networks, especially when a large number of records match the query, but only a small fraction is changed each time, or when some users want to see only the data that have been updated since the last query result. This is our motivation for proposing the *differential re-evaluation algorithm* (DRA), described in Section 5, for processing ECQs. The solution we promote avoids the overhead of recomputing the query from scratch, and relaxes the restrictions of append-only database and monotonic-transformable queries. Our design builds on the research in differential files [25], hypothetical relations [8, 26], and incremental updating materialized views [3, 23, 29], and offers several advantages over the previous solutions [27, 2]. The basic idea is to compute the result of the current execution of a continual query by combining the result produced by the most recent execution with the updates occurred in between these two consecutive executions. We use differential relations to record changes to the source relations (or materialized views). We define a set of differential operators to compute the updates occurred in between two consecutive executions of a continual query.

3 Continual Query Semantics

3.1 Basic Semantics

A continual query CQ is a triple $(Q, T_{CQ}, \texttt{Stop})$, consisting of a normal query Q (e.g., written in SQL), a triggering condition T_{CQ} as specified below, and a termination condition `Stop`. (Since T_{CQ} 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 T_{CQ}, i.e., $\forall S_i,\ T_{CQ} \wedge \neg\texttt{Stop}$.

We assume that there is a mapping from the sequence of database states $S_i, 1 \leq i \leq n$ and $n \in \{1, \ldots, \infty\}$, 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 T_{CQ}. The triggering condition T_{CQ} 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 [5].
- 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 $Q(S_\infty)$. 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 T_{CQ} is triggered, conceptually Q is run against the current state of the database S_n, 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 5 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) [17, 18, 19]. ESR was originally proposed as a correctness criterion to generalize concurrency control in extended transaction processing. The key idea of ESR is that we can generalize serializability by using

the notion of "distance to a serializable state", in database state spaces that define such a distance (e.g., numerical state spaces). 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 (ϵ-spec). A formal characterization of ESR is presented in [19]. Divergence control algorithms that guarantee ESR efficiently have been devised [28, 17], and a prototype implementation [16] done on top of Transarc Encina, a commercial transaction processing monitor.

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, ϵ-spec), where the ϵ-spec is used as the triggering condition T_{CQ}. This way, we can define an ECQ 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 [28, 17] allow limited non-serializable conflicts between updates and the epsilon query to happen, to increase system execution flexibility and concurrency. In an ECQ environment, we can consider the ϵ-spec as the triggering condition that bounds the distance between the previous element in the sequence of ECQ results to the next. Each time the current ECQ query result is about to violate the ϵ-spec, the system generates the next ECQ 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 5 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 CQ

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 5), 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 Notations and Terminology

The relational model is used in this paper to specify continual queries, to record and manipulate changes by differential relations and associated operators. We assume that the reader is familiar with the basic concepts and notation concerning relational database, as described in [14]. In this paper we refer to relational selection, projection, join, outerjoin, union, and difference operators by σ, Π, $\bowtie$, `Outerjoin`[2], $\bigcup$, and $-$ respectively. For presentation convenience, we sometimes use $\Pi(R;X)$ to denote $\Pi_X(R)$, $\sigma(R;F)$ to denote $\sigma_F(R)$, and a SPJ expression to denote a `Select-Project-Join` expression.

We use differential relations 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 define an operator that computes differential relations for arbitrary SPJ expressions. The concept of differential relations is, to some extent, similar to the concept of hypothetical relations used for incremental updating materialized views[3, 8]. 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

[2] `Outerjoin` keeps all tuples in both the left and right relations when no matching tuples are found, padding them with `null` values as needed [14].

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 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.

4.1 Differential Relations

We introduce the concept of differential relation, a relation that can represent changes to another relation, *and* design a set of basic operations to facilitate the manipulation of such relations. The goal for defining differential relations, instead of using hypothetical relations described in [3, 8], is to provide a unified treatment of changes, not separate treatments of insertions, deletions, and modifications resulting in several algorithms for generating and combining individual results.

Let $\mathcal{R}$ denote a relation scheme described by a set of attributes $A_1, A_2, \ldots, A_n$. Let R denote a relation instance of $\mathcal{R}$ consisting of a collection of tuples whose values are taken from the domains of the set of attributes of $\mathcal{R}$. $t.A_i$ $(1 \leq i \leq n)$ denotes the value of attribute A_i for tuple t. Each tuple has an attribute, denoted as *tid*, which provides a unique immutable identifier. When a tuple is deleted and later re-appended to R, it will have a new *tid* assigned. The unique tuple identifier *tid* makes it easier to connect tuples that hold values of the same object before and after changes. In fact, the primary key can be used as the unique identifier for each tuple of R.

Definition 1 (Differential relation)
Let $\mathcal{R} = (tid, A_1, A_2, \ldots, A_n)$ be a relation scheme. For each relation R of scheme $\mathcal{R}$, we define a differential relation, denoted by ΔR, to represent changes.

$$\Delta R = (tid^{<old>}, A_1^{<old>}, \ldots, A_n^{<old>}, tid^{<new>}, A_1^{<new>}, \ldots, A_n^{<new>}, \texttt{timestamp}),$$

where

- $A_i^{<old>}$ refers to old attribute values and
- $A_i^{<new>}$ $(1 \leq i \leq n)$ refers to new attribute values.

- For any tuple $t \in \Delta R$,
  - $tid^{<old>}$ and $tid^{<new>}$ cannot both be null.
  - If $tid^{<old>}$ is null, so are all $t.A_i^{<old>}$ $(1 \leq i \leq n)$.
  - Similarly, if $tid^{<new>}$ is null, so are all $t.A_i^{<new>}$ $(1 \leq i \leq n)$.
  - If both $tid^{<new>}$ and $tid^{<old>}$ are not null, then $tid^{<new>} = tid^{<old>}$.
- For each tuple in ΔR, the system assigns a `timestamp` at its creation time as its identifier. □

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 $A_i^{<old>}$ $(1 \leq i \leq n)$ are null. For deletion, the attributes $A_i^{<new>}$ are null. For modification, attributes $A_i^{<old>}$ hold old values and attributes $A_i^{<new>}$ hold newly modified values. The `timestamp` field is set to the current time (from a system clock, or any other monotonically increasing source of timestamps) whenever a tuple is appended to ΔR.

Example 1 Consider the relation `Stocks` with attributes such as name and price per 100 units.

Stocks:

tid	Name	Price
120992	DEC	150
092394	OLI	145
032090	ODI	120
041977	USL	100
⋮	⋮	⋮

Assume that the transaction `T` updates the `Stocks` relation by insertion, deletion and modification:

```
Transaction T {
    Insert (101088, MAC, 117);
    Modify (120992, DEC, 150) = (120992, DEC, 149);
    Delete (092394);
 }
```

The following differential relation Δ`Stocks` captures the changes that the transaction `T` made to `Stocks`:

ΔStocks:

tid$^{<old>}$	Name$^{<old>}$	Price$^{<old>}$	tid$^{<new>}$	Name$^{<new>}$	Price$^{<new>}$	timestamp
-	-	-	101088	MAC	117	10
120992	DEC	150	120992	DEC	149	25
092394	OLI	145	-	-	-	50

In the rest of this section, for presentation clarity the field `timestamp` is omitted when no confusion is incurred. We sometimes simply use "delta relations" to refer to differential relations.

4.2 Basic Operations

We first define renaming functions that add or remove the superscripts *old* and *new* from attribute names in a relation scheme $\mathcal{R}$.

Definition 2 Let $\mathcal{R} = (tid, A_1, A_2, \ldots, A_n)$ be a relation scheme and R be a relation of scheme $\mathcal{R}$. Let the relation S be of relation scheme $\mathcal{S} = (tid^{<old>}, A_1^{<old>}, \ldots, A_n^{<old>})$. The differential relation schema of $\mathcal{R}$, is denoted by $\Delta\mathcal{R}$.

1. $\texttt{old}(\mathcal{R}) = \{tid^{<old>}, A_1^{<old>}, \ldots, A_n^{<old>}\}$.
2. $\texttt{new}(\mathcal{R}) = \{tid^{<new>}, A_1^{<new>}, \ldots, A_n^{<new>}\}$.
3. $\texttt{detach}[S; \texttt{old}(\mathcal{S})]$ returns the same relation with each attribute name detached from the superscript <old>.
4. $\texttt{attach}[R; \texttt{old}(\mathcal{R})]$ returns the same relation with each attribute name attached by the superscript <old>. □

The second group of operators is used to project the *old* or *new* values of a differential relation, and to compute the updated relation, say R', by combining the relation R with its differential relation ΔR.

Definition 3 (High-level projection operators)

Let R be a relation of $\mathcal{R} = (tid, A_1, A_2, \ldots, A_n)$ and ΔR be a relation of the differential relation schema $\Delta\mathcal{R}$. We define the operators `deletions`, `additions` and `combine` as follows:

1. $\texttt{deleteLog}(\Delta R) = \Pi(\Delta R, \texttt{old}(\mathcal{R}))$.

2. $\texttt{insertLog}(\Delta R) = \Pi(\Delta R, \texttt{new}(\mathcal{R}))$.

3. $\texttt{deletions}(\Delta R) = \texttt{detach}[\texttt{deleteLog}(\Delta R); \texttt{old}(\mathcal{R})]$.

4. $\texttt{insertions}(\Delta R) = \texttt{detach}[\texttt{insertLog}(\Delta R); \texttt{new}(\mathcal{R})]$.

5. $\texttt{combine}(R, \Delta R) = (R - \texttt{deletions}(\Delta R)) \bigcup \texttt{insertions}(\Delta R)$. □

Example 2 Consider the relation Stocks and the differential relation ΔStocks in Example 1.

deleteLog(ΔStocks)

tid$^{<old>}$	Name$^{<old>}$	Price$^{<old>}$
120992	DEC	150
092394	OLI	145

insertions(ΔStocks)

tid	Name	Price
101088	MAC	117
120992	DEC	149

5 Differential Evaluation of Continual Queries

In this section we present a differential re-evaluation algorithm (DRA) 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 epsilon specification 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 for 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 3 for an illustration). We also formally study the correctness of the DRA with respect to the complete re-evaluation solution.

In what follows, we first formally define an operator that computes the differential relations for the data derived by arbitrary query expressions in Section 5.1. In Section 5.2 we introduce the differential forms for the three basic relational algebraic operations: `Select`, `Project`, and `Join`. We prove that instantiation of `Propagate`(Q; PL) for relational select, project, and join are functionally equivalent to their differential forms: `DiffSelect, DiffProj` and `DiffJoin`. The differential re-evaluation algorithm is described in Section 5.3.

5.1 Computing the Differential Results for Continual Queries

In this section we introduce a high level operator, called `Propagate`, to describe how the result relation of a continual query 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.

Before giving the definition of the operator `propagate`, we define an operator that compute the difference of two relations of the same scheme.

Definition 4 (The operator `Diff`)
Let R_1 and R_2 be two relations of the same scheme $\mathcal{R}$. The operator $\texttt{Diff}(R_1, R_2)$ is defined as the $\texttt{Outerjoin}_{\texttt{tid}}$ of tuples in $R_1 - R_2$ and $R_2 - R_1$ under the join condition "$\texttt{tid}^{<old>} = \texttt{tid}^{<new>}$".

$$\texttt{Diff}(R_1, R_2) = \texttt{Outerjoin}_{\texttt{tid}}(\texttt{attach}[(R_1 - R_2);\texttt{old}(\mathcal{R})], \texttt{attach}[(R_2 - R_1);\texttt{new}(\mathcal{R})]).$$

It returns a differential relation ΔR that describes their differences. Null values are used to pad tuples that appear in only $R_1 - R_2$ or $R_2 - R_1$. The relation scheme of ΔR is $\texttt{old}(\mathcal{R}) \cup \texttt{new}(\mathcal{R})$. □

Definition 5 (The `Propagate` operator)
Let R_i be a relation of scheme $\mathcal{R}_i$ $(1 \leq i \leq n)$, ΔR_i be a differential relation of R_i, and R_i' denote $\texttt{combine}(R_i, \Delta R_i)$. Let $\mathrm{Q}(R_1, \ldots, R_n)$ denote an arbitrary continual query and $\mathrm{SL} = \{[R_i, \Delta R_i] \mid (1 \leq i \leq n)\}$ denote the differential substitution list. The `Propagate` operator is defined as follows:

$$\texttt{Propagate}(\mathrm{Q}(R_1, \ldots, R_n); \mathrm{SL}) = \texttt{Diff}(\mathrm{Q}(R_1, \ldots, R_n), \mathrm{Q}(R_1', \ldots, R_n')) \qquad (n \geq 1).$$

It returns the differential relation of query Q, denoted by ΔR_Q, with the scheme $\mathcal{R}_Q$. □

We may also denote Propagate(Q($R_1, \ldots, R_n$); SL) by ΔQ($R_1, \ldots, R_n$).

Example 3 Consider the query Q: $\sigma_{\texttt{price}>120}$(Stocks) as a continual query. Let E_i be the ith execution of Q at time t_i and E_i(Q, t_i) denote the result of the ith execution of Q ($i = 0, \ldots, \infty$). Now assume that the base relation Stocks is changed, after the last execution E_i and before the current execution E_{i+1}, by the update transaction T given in Example 1. Let Q(Stocks) = E_i(Q, t_i) denote the result of the ith execution of Q over Stocks. Let Stocks' denote the base relation after updates to Stocks by transaction T, and Q(Stocks') = E_{i+1}(Q, t_{i+1}) denote the result of the current execution of Q, over the relation Stocks'.

Based on Definition 5, to express how the result E_i(Q, t_i) may change after the updates by the transaction T, we may simply compute Propagate(Q(Stocks);[Stocks,ΔStocks]), the difference between the result relation before the updates: Q(Stocks) and the result relation after the update: Q(Stocks').

Q(Stocks) = $\sigma_{\texttt{price}>120}$(Stocks) = {(120992,DEC,150), (092394,OLI,145)}.

Q(Stocks') = $\sigma_{\texttt{price}>120}$(Stocks') = {(120992,DEC,149)}.

Q(Stocks)$-$Q(Stocks') = {(120992,DEC,150), (092394,OLI,145)}.

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

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

ΔQ(Stocks) = Diff(Q(Stocks),Q(Stocks'))

tid$^{<old>}$	Name$^{<old>}$	Price$^{<old>}$	tid$^{<new>}$	Name$^{<new>}$	Price$^{<new>}$
120992	DEC	150	120992	DEC	149
092394	OLI	145	-	-	-

□

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

Observe that when (i)the size of the source relation Stocks is large, (ii)the selectivity factor of the query Q over Stocks is not high, and (iii)the number of tuples written 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 ΔStocks instead. It is because computing Propagate($\sigma_{\texttt{price}>120}$(Stocks);[Stocks,$\Delta$Stocks]) is functionally equivalent to the evaluation of the differential query: $\sigma_F(\Delta\texttt{Stocks})$, where F denotes the predicate $\texttt{price}^{<old>} > 120 \wedge \texttt{price}^{<new>} > 120 \wedge \texttt{timestamp} > t_i$[3]. This is true even when the user needs the complete composite set of the query result, because computing the union of Q(Stocks) and insertions($\sigma_F(\Delta\texttt{Stocks})$) is cheaper than recomputing the expression $\sigma_{price>120}$(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 deletions($\sigma_F(\Delta\texttt{Stocks})$). In general, for any continual query Q over the relation 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:

$$(E_i(Q,t_i) - \sigma_{timestamp>t_i}(\texttt{deletions}(\Delta R))) \cup \sigma_{timestamp>t_i}(\texttt{insertions}(\Delta R)).$$

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

5.2 Optimization based on Differential Operators

For many forms of queries, Propagate(Q; PL) can be computed more efficiently than by directly evaluating the definition of the Propagate operator. These computations use the differential relations heavily and sometimes exclusively, instead of computing on the base relations, which tend to be much larger.

In this section we define the differential forms for the three basic relational algebraic operations: Select, Project, and Join. We prove that instantiation of Propagate(Q; PL) for relational select, project, and join are functionally equivalent to their differential forms: DiffSelect, DiffProj and DiffJoin. The predicates contained in these operators can be atomic or composite by logical "AND" and logical "OR". The attributes involved in the predicates or the projection list are single valued attributes. Aggregate functions such as SUM, COUNT, MAX, MIN are allowed in this framework with some additional consideration. For example, when the query expression contains the aggregate function MAX(A), we need to compare the A field of each $< new >$ tuple in the differential relation with the value of MAX(A), and to reset MAX(A) if necessary. Similar treatment applies to SUM, COUNT, MIN. However when the query

[3]The condition timestamp$> t_i$ is appended by the CQ manager for incorporating correct amount of updates and limiting the query search space. Section 6.3 provides some further discussion.

contains the aggregate function AVG, the recomputation of the query is needed.

5.2.1 The Differential Select Operator

Definition 6 (Differential select)
Let F denote a predicate defined on the relation R of scheme $\mathcal{R}$, and $\sigma[R;F]$ denote the associated relational selection operator. Let $F^{<old>}$ and $F^{<new>}$ denote the same predicate with all attribute names superscripted accordingly. We define the operator DiffSelect as follows:

$$\texttt{DiffSelect}[\Delta R;F] = \texttt{Outerjoin}_{tid}(\sigma[\texttt{deleteLog}(\Delta R);F^{<old>}], \sigma[\texttt{insertLog}(\Delta R);F^{<new>}]). \quad \square$$

Proposition 1 $\texttt{Propagate}(\sigma[R;F];(R,\Delta R)) \equiv \texttt{DiffSelect}[\Delta R;F]$.
Proof (see Appendix A).

This proposition shows that Definition 6 gives an implementation of DiffSelect which does not reference the base relation, and which can be implemented by one pass over ΔR, determining for each tuple of ΔR whether it induces an insertion, deletion, or modification to $\sigma[R;F]$.

When $|R| \geq |\Delta R|$, it is cheaper to re-evaluate the query expression $\sigma_P(R)$ by using the differential select operator $\texttt{DiffSelect}(\Delta R; P)$ rather than recomputing the expression $\sigma_P(R)$ from scratch.

5.2.2 The Differential Project Operator

Definition 7 (Differential project)
Let R be a relation (base or derived) of scheme $\mathcal{R} = (A_1, \ldots, A_n)$. Let $\mathbf{X} = \{A_{i_1}, \ldots, A_{i_k}\}$ denotes a subset of the attributes in $\mathcal{R}$, $\texttt{tid} \in \mathbf{X}$, and $i_j \in \{1, \ldots, n\}$ $(1 \leq j \leq k)$. Let P denote the predicate $\bigvee_{j=1}^{k}(A_{i_j}^{<old>} \neq A_{i_j}^{<new>})$ if R is a (persistent) base relation; otherwise P denote the truth value true. The differential project operator DiffProj is defined as follows:

$$\texttt{DiffProj}[\Delta R;\mathbf{X}] = \sigma_P(\Pi[\Delta R; \texttt{old}(\mathbf{X}) \bigcup \texttt{new}(\mathbf{X})]). \quad \square$$

The following proposition shows that DiffProj is an efficient differential form that can be used to implement $\texttt{Propagate}(\Pi[R;\mathrm{X}];(R,\Delta R))$, because they are functionally equivalent, DiffProj runs one pass over ΔR, and the Propagate runs over R' which tends to be much larger than ΔR.

Proposition 2 $\texttt{Propagate}(\Pi[R;\mathbf{X}]; (R,\Delta R)) \equiv \texttt{DiffProj}[\Delta R; \texttt{old}(\mathbf{X}) \bigcup \texttt{new}(\mathbf{X})]$.
Proof (see Appendix A).

Example 4 Consider the following two queries over the relation `Stocks` in Example 1:

$$Q_1 = \Pi[\texttt{Stocks}; \texttt{Name}].$$

$$Q_2 = \Pi[\sigma_{\texttt{price}>120}(\texttt{Stocks}); \texttt{Name}].$$

Let `HighStocks` = σ(`Stocks`; `Price>120`). By Definition 5, the evaluation of `Propagate`($\sigma_{\texttt{price}>120}$(`Stocks`); [`Stocks`,$\Delta$`Stocks`]) requires as the inputs both the base relation `Stocks` before the updates and the base relation after the updates, which is `Stocks'`. According to Proposition 2 and Definition 7, it would be more efficient to process both queries (Q_1 and Q_2) using the differential operator `DiffProj` than directly using the definition of `Propagate`.

- `Propagate`(Π[`Stocks`; `Name`]; [`Stocks`,Δ`Stocks`])
  = `DiffProj`[Δ`Stocks`; `Name`].

tid$^{<old>}$	Name$^{<old>}$	tid$^{<new>}$	Name$^{<new>}$
-	-	101088	MAC
092394	OLI	-	-

- `Propagate`(Π[`HighStocks`; `old`(`Name`)$\cup$`new`(`Name`)]; [`HighStocks`,Δ`HighStocks`])
  = `DiffProj`[Δ`HighStocks`; `old`(`Name`)$\cup$`new`(`Name`)] (by Proposition 2)
  = `DiffProj`[`Propagate`(`HighStocks`;[`Stocks`,Δ`Stocks`]); `old`(`Name`)$\cup$`new`(`Name`)] (by Lemma 2)
  = `DiffProj`[`DiffSelect`(Δ`Stocks`); `old`(`Name`)$\cup$`new`(`Name`)]. (by Proposition 1)

tid$^{<old>}$	Name$^{<old>}$	tid$^{<new>}$	Name$^{<new>}$
120992	DEC	120992	DEC
092394	OLI	-	-

□

5.2.3 The Differential Join Operator

We first consider the join over two relations R_1 and R_2 (i.e., $n = 2$). In this case changes to the resulting relation can be induced by changes to either input operand or both of them. Therefore, the differential join operator `DiffJoin` should consider the following three cases when computing the total changes to the result relation of $R_1 \bowtie R_2$: (i)only R_1 changes; (ii)only R_2 changes; and (iii)both R_1 and R_2 change.

Definition 8 (Differential join)
Let P_{join} be a predicate on $\mathcal{R}_1$ and $\mathcal{R}_2$, and let $\texttt{tid}_1 \in \mathcal{R}_1$ and $\texttt{tid}_2 \in \mathcal{R}_2$ respectively. Let $P_{join}^{<old>}$ and

$P_{join}^{<new>}$ denote the predicates obtained from P_{join} by attaching each attribute name with superscript $< old >$ and $< new >$ respectively. Let $P(tid_1, tid_2)$ denote the predicate "$(tid_1^{<old>} = tid_1^{<new>})$ AND $(tid_2^{<old>} = tid_2^{<new>})$". Let $\mathcal{R}$ denote $\mathcal{R}_1 \cup \mathcal{R}_2$. We define the operator DiffJoin as follows:

$$\texttt{DiffJoin}_{P(tid_1,tid_2)}(R_1, R_2)$$
$$= \texttt{DJoin}_{P_{join}}(\Delta R_1, R_2) \bigcup \texttt{DJoin}_{P_{join}}(R_1, \Delta R_2) \bigcup \texttt{DJoin}_{P_{join}}(\Delta R_1, \Delta R_2).$$

where

- $\texttt{DJoin}_{P_{join}}(\Delta R_1, R_2)$
  $= \texttt{Outerjoin}_{P(tid_1,tid_2)}(\texttt{attach}[\bowtie_{P_{join}^{<old>}}(\texttt{deletions}(\Delta R_1),\ R_2);\ \texttt{old}(\mathcal{R})],$
  $\texttt{attach}[\bowtie_{P_{join}^{<new>}}(\texttt{insertions}(\Delta R_1),\ R_2);\ \texttt{new}(\mathcal{R})]).$
- $\texttt{DJoin}_{P_{join}}(R_1, \Delta R_2)$
  $= \texttt{Outerjoin}_{P(tid_1,tid_2)}(\texttt{attach}[\bowtie_{P_{join}^{<old>}}(R_1,\ \texttt{deletions}(\Delta R_2);\ \texttt{old}(\mathcal{R})],$
  $\texttt{attach}[\bowtie_{P_{join}^{<new>}}(R_1, \texttt{insertions}(\Delta R_2);\ \texttt{new}(\mathcal{R})]).$
- $\texttt{DJoin}_{P_{join}}(\Delta R_1, \Delta R_2)$
  $= \texttt{Outerjoin}_{P(tid_1,tid_2)}(\bowtie_{P_{join}^{<old>}}(\texttt{deleteLog}(\Delta R_1),\ \texttt{deleteLog}(\Delta R_2)),$
  $\bowtie_{P_{join}^{<new>}}(\texttt{insertLog}(\Delta R_1),\ \texttt{insertLog}(\Delta R_2)).$ □

Proposition 3 $\texttt{Propagate}(\bowtie_{P_{join}}(R_1, R_2);\ [(R_1, \Delta R_1), (R_2, \Delta R_2)]) \equiv \texttt{DiffJoin}_{P(tid_1,tid_2)}(R_1, R_2)$.

Proof (see Appendix A).

This development can be generalized to the join of an arbitrary number of base relations. Let $\mathrm{Q} = R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_n$ $(n \geq 2)$ denote an arbitrary join expression. The equation $\texttt{Propagate}(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n;\ \mathrm{SL}) \equiv \texttt{DiffJoin}_{P_{tid_1,\ldots,tid_n}}(R_1, R_2, \ldots, R_n)$ holds in general. Also, when the number k $(k \leq n)$ relations have been changed since the last execution of Q, to evaluate $\texttt{DiffJoin}_{P_{tid_1,\ldots,tid_n}}(R_1, R_2, \ldots, R_n)$, we need to consider only $2^k - 1$ cases, each representing one type of change effects. The total changes to the result of Q will be the union of these cases. According to the associative and symmetric property of relational join, we may assume that the first k relations (i.e., $R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_k, k \leq n$) are those that have been changed without loss of generality.

By associating a truth table T, with k columns and $p = 2^k - 1$ rows, to the query Q, it is easy to compute each possible combination of joins, which needs to be considered when computing the changes to the result of Q after k operand relations have been changed. Each column of the T table corresponds to an updated relation in Q since the last execution of Q. Each row represents one possible case that the computation of $\texttt{DiffJoin}(R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_n)$ considers.

Example 5 For a query Q $= R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_n$ $(n \geq 2)$, consider the case when $k = 3$ and $k < n$. Based on the T table associated with Q below, we need to consider only seven cases: (1)R_1, R_2, R_3 all change; (2)only R_1, R_2 change; (3)only R_1, R_3 change; (4)only R_2, R_3 change; (5)only R_1 changes; (6)only R_2 changes; (7)only R_3 changes. Each row in the T table corresponds to one possible combination of join required to compute the changes to the result of Q.

T1	T2	T3	DiffJoin$(R_1, R_2, R_3, \ldots, R_n) =$
1	0	0	DJoin$(\Delta R_1, R_2, R_3, \ldots, R_n)$
0	1	0	$\cup$ DiffJoin$(R_1, \Delta R_2, R_3, \ldots, R_n)$
0	0	1	$\cup$ DJoin$(R_1, R_2, \Delta R_3, \ldots, R_n)$
1	1	0	$\cup$ DJoin$(\Delta R_1, \Delta R_2, R_3, \ldots, R_n)$
1	0	1	$\cup$ DJoin$(\Delta R_1, R_2, \Delta R_3, \ldots, R_n)$
0	1	1	$\cup$ DJoin$(R_1, \Delta R_2, \Delta R_3, \ldots, R_n)$
1	1	1	$\cup$ DJoin$(\Delta R_1, \Delta R_2, \Delta R_3, \ldots, R_n)$

Let $\mathcal{R}_i$ be the scheme of relation R_i $(1 \leq i \leq 3)$ and $\mathcal{R}$ denote $\cup_{i=1}^{n} \mathcal{R}_i$. The definitions for the above DJoins are similar to those in Definition 8. For instance,

DJoin$_{P_{join}}(\Delta R_1, R_2, R_3, \ldots, R_n)$

$=$ Outerjoin$_{tid}$(attach[deletions$(\Delta R_1) \bowtie R_2 \bowtie R_3 \bowtie \ldots \bowtie R_n)$; old$(\mathcal{R})$],

attach[insertions$(\Delta R_1) \bowtie R_2 \bowtie R_3 \bowtie \ldots \bowtie R_n)$; new$(\mathcal{R})$]).

DJoin$_{P_{join}}(\Delta R_1, R_2, \Delta R_3, \ldots, R_n)$

$=$ Outerjoin$_{tid}$(attach[deletions$(\Delta R_1) \bowtie R_2 \bowtie$deletions$(\Delta R_3) \bowtie \ldots \bowtie R_n)$; old$(\mathcal{R})$],

attach[insertions$(\Delta R_1) \bowtie R_2 \bowtie$insertions$(\Delta R_3) \bowtie \ldots \bowtie R_n)$; new$(\mathcal{R})$]). □

Observe that the above example exhibits an interesting optimization problem, namely, the efficient execution of a set of n-ary join expressions in which intermediate results can be reused among several *SPJ* expressions. For instance, when $n > 4$ in the above query Q, let $W_1 = R_4 \bowtie \ldots \bowtie R_n$ and $W_2 = R_2 \bowtie R_3 \bowtie W_1$. Saving W_1 and W_2 as intermediate results, and then re-using them in the evaluation of each of the seven DJoin expressions above, we may easily speed up the processing of DiffJoin. This mechanism works effectively when n is larger than k.

The idea of using the truth table to facilitate the combination of possible joins was borrowed from the research in updating materialized view [3, 8]. We minimize the cost of constructing such a table by using as the columns of the table only the number of changed relations, instead of the n operand relations in the query expression $\Pi_X(\sigma_F(R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_n))$.

5.3 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 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 ΔR_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: the result of the current execution of the 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 ΔR_i when the binary variable $\mathrm{T}_{ij} = 1$. 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 `DJoin`$(S_1 \bowtie S_2 \bowtie, \ldots, \bowtie S_n)$.
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 ΔR_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 $\mathrm{E}_i(\mathrm{Q}, t_i)$, without deletion notification, the result to be returned can be computed by $\sigma_{timestamp > t_i}$(`insertions`(ΔR_Q)).

- If the user needs to see the complete set of the result matching the query, we return $\mathrm{E}_i(\mathrm{Q},t_i) \bigcup \sigma_{timestamp>t_i}(\texttt{insertions}(\Delta R_Q))$.
- If the user wants to be notified all the deleted tuples since the last execution of the CQ, we simply compute the $\sigma_{timestamp>t_i}(\texttt{deletions}(\Delta R_Q))$.

6 Discussion

In previous sections we have defined the differential forms for the three basic relational algebraic operations: `Select`, `Project`, and `Join`, and developed a differential re-evaluation algorithm (DRA). The key idea of the DRA method is to transform a continual query over the source data (base relations) into a differential query that runs over the corresponding delta relations. We have proved that, for any SPJ expression, using differential re-evaluation method is functionally equivalent to the complete re-evaluation of the query.

In this section we will discuss a number of issues related to the implementation of continual queries in the Diorama project [13].

6.1 Improving Performance in Compilation and Execution

Our differential algorithm for query refresh does not automatically provide the most efficient way of processing CQs. There are a number of possibilities to further improve the performance of differential processing of continual queries.

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. Another benefit is that the DBMS supports relational operators (`select, project, join` etc.) and an interpreter for the strategies that are produced. Otherwise, we need to implement these capabilities as part of the CQ manager. However, there are also serious drawbacks to using a conventional DBMS that is non extensible. In this case, integration of the CQ manager may be difficult or impossible. One must find a way to add operators that manipulate differential relations.

6.2 Differential Evaluation of Continual Query Conditions

As described in Section 3.1, a continual query CQ consists of a normal query Q (e.g., written in SQL), an epsilon specification (ϵ-spec) as the triggering condition, and a termination condition `Stop`. We use the term continual query condition to refer to the refresh condition T_{CQ} (ϵ-spec) and the termination condition `Stop`. Whenever the ϵ-spec of a given ECQ is satisfied, the CQ manager will invoke the differential re-evaluation algorithm to execute the query Q. A continual query ceases to exist when its termination condition `Stop` becomes true.

The next question that needs to be answered is *when* and *how* the CQ manager checks the refresh condition T_{CQ}, usually an ϵ-spec, and the termination condition `Stop`. Given an ECQ, if its ϵ-spec is a direct specification of time (e.g., every Monday) or a time interval with respect to the previous query result (e.g., a week since the last execution of query Q), then the CQ manager will use the ϵ-spec as the default for triggering the CQ and testing the `Stop` condition. However, if the ϵ-spec and the `Stop` condition are bounded to a database state or a relationship between a previous query result and the current database state (recall examples in Section 3.1), then the CQ manager will decide when and how to evaluate T_{CQ}.

There are two general strategies concerning when to test the T_{CQ}. The first is to evaluate T_{CQ} immediately after each update to the source data (base relations). The second is to evaluate T_{CQ} 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 CheckAcc`. Suppose it is installed as a continual query with the following T_{CQ} (and ϵ-spec): $|$`Deposits − 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 ϵ-spec of this query, the CQ manager can decide when to evaluate T_{CQ} by a system-defined default interval, say every day at midnight. If the ϵ-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 T_{CQ} whenever possible. In this example the CQ manager first rewrites T_{CQ}, originally specified as |`Deposits − Withdrawals`| $\geq 0.5M$, which requires testing of the base relation `CheckAcc`. The new T_{CQ} has a differential form that requires only scanning Δ`CheckAcc`:

`Deposits:= SELECT SUM(account) FROM insertions(`Δ`CheckAcc) WHERE timestamp` $> t_{i-1}$.

`Withdrawals:= SELECT SUM(account) FROM deletions(`Δ`CheckAcc) WHERE timestamp` $> t_{i-1}$.

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 T_{CQ} is cheaper than the complete re-evaluation of T_{CQ} over the entire base relations. In this example, it is true when |`CheckAcc`| > |Δ`CheckAcc`|.

6.3 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 $Q(S_i)$ 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.

7 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 increases overhead similarly, making information monitoring more difficult and expensive. In this paper, we address the two issues with two complementary ideas.

First, we define *Continual Queries* (CQ) to facilitate the query specification for information monitoring. Each CQ is a sequence of query executions, marked by a normal database query, for example, formulated using SQL. The sequence is started at CQ submission time, refreshed from time to time by a triggering condition T_{CQ}, and terminated by a stopping condition. Our triggering conditions are more expressive than previous proposals such as continuous queries [27], since T_{CQ} can describe time of day, intervals of time, the magnitude of updates, or accumulated updates, in a way similar to epsilon transactions [18, 19]. These new CQs are called Epsilon CQs (ECQs).

Second, we describe a differential re-evaluation algorithm to support efficient processing of ECQs. Given the distributed nature of Internet, complete reprocessing of the ECQ query each time it is triggered would be prohibitively expensive. The differential re-evaluation algorithm is based on *differential relations*, summaries of updates that allow efficient incremental processing of the next result in the ECQ sequence by combining the differential relations with the current ECQ result. We have defined a high level operator `Propagate` as an instantiation of *complete re-evaluation* solution and 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. In addition to a formal description of delta relations (Section 4) and the differential re-evaluation algorithm (Section 5), we also discuss a number of implementation issues such as differential evaluation of continual query conditions and strategies for garbage collection of delta relations (Section 6).

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, integration of information from heterogeneous sources, other algorithms for differential or incremental evaluation of ECQs, 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.

References

[1] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems*, 15(3):359–384, September 1990.

[2] D. Barbara and R. Alonso. Processing continuous queries in general environments. Technical report, Matsushita Information Technology Laboratory, Princeton, NJ, June 1993.

[3] J. Blakeley, P. Larson, and F. Tompa. Efficiently updating materialized views. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 61–71, Washington, DC, May 1986.

[4] C. Bowman, P. Danzig, D. Hardy, U. Manber, and M. F. Schwartz. The harvest information discovery and access system. In *Proceedings of the Second International World Wide Web Conference*, pages 763–771, Chicago, Illinois, October 1994.

[5] C. Bowman, P. Danzig, D. Hardy, U. Manber, M. F. Schwartz, and D. P. Wessels. Harvest: A scalable, customizable discovery and access system. Technical report, Department of Computer Science, University of Colorado, Technical Report CU-CS-732-94, August 1994.

[6] S. Chakravarthy. Architectures and monitoring techniques for active databases: An evaluation. In *Technical Report TR-92-041, University of Florida*, Gainesville, FL, 1992.

[7] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 157–166, Washington, DC, May 1993.

[8] E. N. Hanson. A performance analysis of view materialization strategies. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 440–453, San Francisco, CA, May 1987.

[9] Informix Software, Inc. *Informix Guide to SQL: Syntax (Version 6.0)*, 1994.

[10] B. Kahler and O. Risnes. Extending logging for database snapshot refresh. In *Proceedings of the International Conference on Very Large Data Bases*, pages 389–398, Brighton, England, September 1987.

[11] B. Lindsay, L. Haas, and C. Mohan. A snapshot differential refresh algorithm. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 53–60, Washington, DC, May 1986.

[12] L. Liu and C. Pu. The distributed interoperable object model and its application to large-scale interoperable database systems. Technical report, Portland, OR, March 1995.

[13] L. Liu and C. Pu. The distributed interoperable object model and its application to large-scale interoperable database systems. Technical Report OGI-CSE-95-003, Department of Computer Science and Engineering, Oregon Graduate Institute, February 1995.

[14] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[15] D. McCarthy and U. Dayal. The architecture of an active database management system. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 215–224, May 1989.

[16] C. Pu and S. Chen. ACID properties need fast relief: Relaxing consistency using epsilon serializability. In *Proceedings of Fifth International Workshop on High Performance Transaction Systems*, Asilomar, California, September 1993.

[17] C. Pu, W. Hseush, G. Kaiser, K.-L. Wu, and P. Yu. Divergence control for distributed database systems. *International Journal on Distributed and Parallel Databases*, 3(1):85–109, January 1995.

[18] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 377–386, Denver, May 1991.

[19] K. Ramamrithan and C. Pu. A formal characterization of epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering*, June 1995.

[20] A. Rosenthal, S. Chakravarthy, B. Blaustein, and J. Blakely. Situation monitoring for active databases. In *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 469–478, Amsterdam, Holland, September 1989.

[21] N. Roussopoulos and H. Kang. Preliminary design of adms+: A workstation-mainframe integrated architecture fopr database management systems. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 355–364, Kyoto, Japan, August 1986.

[22] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An architecture for transforming a passive dbms into an active dbms. In *Proceedings of the International Conference on Very Large Data Bases*, pages 469–478, Barcelona, Spain, September 1991.

[23] A. Segev and W. Fang. Currency-based updates to distributed materialized views. In *Proceedings of the 6th International Conference on Data Engineering*, pages 512–520, Los Alamitos, February 1990.

[24] A. Segev and J. Park. Maintaining materialized views in distributed databases. In *Proceedings of the 5th International Conference on Data Engineering*, pages 262–270, Los Angeles, February 1989.

[25] D. G. Severance and G. M. Lohman. Differential files: Their application to the maintenance of large databases. *ACM Transactions on Database Systems*, 1(3):256–267, September 1976.

[26] M. Stonebreaker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 281–290, Barcelona, Spain, 1990.

[27] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 321–330, San Diego, CA, January 1992.

[28] K. Wu, P. S. Yu, and C. Pu. Divergence control for epsilon-serializability. In *Proceedings of Eighth International Conference on Data Engineering*, pages 506–515, Phoenix, February 1992. IEEE/Computer Society.

[29] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, San Jose, CA, May 1995.

Appendix A Lemmas and Proposition Proofs

Lemma 1 The following equations hold: 1. $\texttt{attach}[\texttt{additions}(\Delta R), \texttt{new}(\mathcal{R})] = \texttt{insertLog}(\Delta R)$.

2. $\texttt{attach}[\texttt{deletions}(\Delta R), \texttt{old}(\mathcal{R})] = \texttt{deleteLog}(\Delta R)$.

3. $\texttt{Outerjoin}_{\texttt{tid}}(\texttt{deleteLog}(\Delta R), \texttt{insertLog}(\Delta R)) = \Delta R$.

4. $R - \texttt{combine}(R, \Delta R) = \texttt{deletions}(\Delta R)$.

5. $\texttt{combine}(R, \Delta R) - R = \texttt{insertions}(\Delta R)$. □

The proof of the equations in Lemma 1 follows straightforward from Definition 2 and Definition 3.

Lemma 2 The following equations hold:

1. $\texttt{Propagate}(R; [R, \Delta R]) = \Delta R$.

2. $\texttt{combine}(\mathrm{Q}(R_1, \ldots, R_n), \texttt{Propagate}(\mathrm{Q}; \mathrm{PL})) = \mathrm{Q}(R'_1, \ldots, R'_n)$. □

Proposition 1 $\texttt{Propagate}(\sigma[R;F];(R, \Delta R)) \equiv \texttt{DiffSelect}[\Delta R;F]$.

Proof

Let RHS and LHS represent the right and left hand side of the equation. Let the condition `tid` of the $\texttt{Outerjoin}_{\texttt{tid}}$ denote "$\texttt{tid}^{<old>} = \texttt{tid}^{<new>}$", $F^{<old>}$ and $F^{<new>}$ denote the same predicate with all attribute names superscripted accordingly. By Definition 6,

RHS = $\texttt{Outerjoin}_{tid}(\sigma[\texttt{deleteLog}(\Delta R);F^{<old>}], \sigma[\texttt{insertLog}(\Delta R);F^{<new>}])$.

Let E1 = $\sigma[R;F] - \sigma[\texttt{combine}(R, \Delta R);F] = \sigma[R - \texttt{combine}(R, \Delta R); F]$

$= \sigma[\texttt{deletions}(\Delta R); F]$ (by Lemma 1(4)).

E2 = $\sigma[\texttt{combine}(R, \Delta R);F] - \sigma[R;F] = \sigma[\texttt{combine}(R, \Delta R) - R; F]$

$= \sigma[\texttt{insertions}(\Delta R); F]$ (by Lemma 1(5)).

By Definition 5 (`Propagate`) and Definition 4 (`Diff`),

LHS = $\texttt{Diff}(\sigma[R;F];\sigma[\texttt{combine}(R, \Delta R);F])$

$= \texttt{Outerjoin}_{tid}(\texttt{attach}[\mathrm{E1};\texttt{old}(\mathcal{R})], \texttt{attach}[\mathrm{E2};\texttt{new}(\mathcal{R})])$

$= \texttt{Outerjoin}_{tid}(\sigma[\texttt{deleteLog}(\Delta R);F^{<old>}], \sigma[\texttt{insertLog}(\Delta R);F^{<new>}])$ (Definition 3). □

Proposition 2 $\texttt{Propagate}(\Pi[R;\mathbf{X}]; (R, \Delta R)) \equiv \texttt{DiffProj}[\Delta R; \texttt{old}(\mathbf{X}) \bigcup \texttt{new}(\mathbf{X})]$.

Proof

Let E1 = $\Pi[R;\mathbf{X}] - \Pi[\texttt{combine}(R,\Delta R);\mathbf{X}] = \Pi[R - \texttt{combine}(R,\Delta R);\mathbf{X}]$

$= \Pi[\texttt{deletions}(\Delta R);\mathbf{X}]$ (Lemma 1(4)).

E2 = $\Pi[\texttt{combine}(R,\Delta R);\mathbf{X}] - \Pi[R;\mathbf{X}] = \Pi[\texttt{combine}(R,\Delta R) - R;\mathbf{X}]$

$= \Pi[\texttt{insertions}(\Delta R);\mathbf{X}]$ (Lemma 1(5)).

The expression $\Pi[R \bigcap \texttt{combine}(R,\Delta R);\mathrm{X}]$ returns the set of $\mathbf{X}$ tuples for which their attributes are unchanged. By Definition 7,

LHS = $\texttt{Diff}(\Pi[R;\mathbf{X}], \Pi[\texttt{combine}(R,\Delta R);\mathbf{X}])$

$= \texttt{Outerjoin}_{tid}(\texttt{attach}[\mathrm{E1};\texttt{old}(\mathbf{X})], \texttt{attach}[\mathrm{E2};\texttt{new}(\mathbf{X})])$

$= \texttt{Outerjoin}_{tid}(\Pi[\texttt{deleteLog}(\Delta R);\texttt{old}(\mathbf{X})], \Pi[\texttt{insertLog}(\Delta R);\texttt{new}(\mathbf{X})])$

(by Lemma 1(1)(2) and Definition 3)

$= \sigma_P(\Pi[\Delta R;\texttt{old}(\mathbf{X}) \bigcup \texttt{new}(\mathbf{X})])$ (by Lemma 1(3) and Definition 7) = RHL. □

In the proof we have used the distributive property of projection over difference (i.e., $\Pi_X(R_1 - R_2) = \Pi_X(R_1) - \Pi_X(R_2)$), which holds under the assumption that each tuple has a unique immutable identifier (recall Section 4.1). This is because, under this assumption, the projection is re-defined as $\Pi_X(R) = \{t' | t'.tid = t.tid \wedge t'[X] = t[X]\}$, and the result relation scheme is $\{tid\} \cup X$, instead of X.

However, the distributive property of projection over difference does not hold for the traditional relational project operator which eliminates the duplicates in its result relation [14]. That is $\Pi_X(R_1 - R_2) \neq \Pi_X(R_1) - \Pi_X(R_2)$. An alternative way to maintain the distributive property of projection over difference is to associate a multiplicity *counter* to each tuple in a relation. This approach has been used by many algorithms for updating materialized views [3, 8].

Proposition 3 $\texttt{Propagate}(\bowtie_{P_{join}}(R_1,R_2); [(R_1,\Delta R_1),(R_2,\Delta R_2)]) \equiv \texttt{DiffJoin}_{P(tid_1,tid_2)}(R_1,R_2)$.

Proof

Let E1 = $\bowtie_{P_{join}}(R_1,R_2) - \bowtie_{P_{join}}(\texttt{combine}[R_1,\Delta R_1],\texttt{combine}[R_2,\Delta R_2]))$

$= \bowtie_{P_{join}}(\texttt{deletions}(\Delta R_1),R_2) \bigcup \bowtie_{P_{join}}(R_1,\texttt{deletions}(\Delta R_2))$

$\bigcup \bowtie_{P_{join}}(\texttt{deletions}(\Delta R_1),\texttt{deletions}(\Delta R_2))$.

E2 = $\bowtie_{P_{join}}(\texttt{combine}[R_1,\Delta R_1],\texttt{combine}[R_2,\Delta R_2])) - \bowtie_{P_{join}}(R_1,R_2)$

$$= \bowtie_{P_{join}}(\texttt{insertions}(\Delta R_1), R_2) \bigcup \bowtie_{P_{join}}(R_1, \texttt{insertions}(\Delta R_2))$$

$$\bigcup \bowtie_{P_{join}}(\texttt{insertions}(\Delta R_1), \texttt{insertions}(\Delta R_2)).$$

By Definition 5 and Definition 4,

$$\text{LHS} = \texttt{Diff}(\bowtie_{P_{join}}(R_1, R_2), \bowtie_{P_{join}}(\texttt{combine}[R_1, \Delta R_1], \texttt{combine}[R_2, \Delta R_2]))$$

$$= \texttt{Outerjoin}_{P(tid_1, tid_2)}(\texttt{attach}[\text{E1}; \texttt{old}(\mathcal{R}) \bigcup \texttt{new}(\mathcal{R})], \texttt{attach}[\text{E2}; \texttt{old}(\mathcal{R}) \bigcup \texttt{new}(\mathcal{R})])$$

$$= \texttt{Outerjoin}_{P(tid_1, tid_2)}(\texttt{attach}[\bowtie_{P_{join}^{<old>}}(\texttt{deletions}(\Delta R_1),\ R_2); \texttt{old}(\mathcal{R})],$$

$$\texttt{attach}[\bowtie_{P_{join}^{<new>}}(\texttt{insertions}(\Delta R_1),\ R_2); \texttt{new}(\mathcal{R})])$$

$$\bigcup \texttt{Outerjoin}_{P(tid_1, tid_2)}(\texttt{attach}[\bowtie_{P_{join}^{<old>}}(R_1, \texttt{deletions}(\Delta R_2)); \texttt{old}(\mathcal{R})],$$

$$\texttt{attach}[\bowtie_{P_{join}^{<new>}}(R_1, \texttt{insertions}(\Delta R_2)); \texttt{new}(\mathcal{R})])$$

$$\bigcup \texttt{Outerjoin}_{P(tid_1, tid_2)}(\bowtie_{P_{join}^{<old>}}(\texttt{deleteLog}(\Delta R_1), \texttt{deleteLog}(\Delta R_2)),$$

$$\bowtie_{P_{join}^{<new>}}(\texttt{insertLog}(\Delta R_1), \texttt{insertLog}(\Delta R_2)) = \text{RHS}. \qquad \square$$