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,tzhou@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, 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. As a consequence, not only can the duplicates in the query result be avoided but also the unnecessary computation cost can be minimized. This approach exhibits much better performance improvement when the information sources are large in size and evolve dynamically. The DRA design can be seen as a synthesis of previous research on differential files, incremental view maintenance, and active databases.

Index Terms: Intelligent information retrieval, information monitoring, continual queries, differential data, incremental query processing, interoperable information systems.

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, 8], active databases [22], and query optimization, while powerful by themselves, encounter difficulties in distributed interoperable environments such as the Internet. The interoperation of these database techniques with each other and with the relatively unorganized storage and management of data, such as WWW pages [5] is completely non-trivial.

To address the interoperability problem in Internet environments we have developed the Distributed Interoperable Object Model (DIOM) [13, 12], 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 [13] and Epsilon Serializability [28, 19] 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 [19], 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 [27, 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".

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 [27] 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.

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 the DRA technique (Section 5) for efficient processing of continual queries.

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. Perhaps the most significant 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*.

Active Databases: Facilities [6, 15, 20, 22] that allow users to specify, in the form of rules and 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 [27], 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.

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. 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]. Other works on distributed environments include quasi-copies for replication [1] and update anomalies in data warehouses [29]

Research Challenges: Although powerful database techniques such as active databases and materialized views have been developed to monitor updates within homogeneous and consistent databases, these techniques do not easily extend to a distributed interoperable environment such as Internet. The main reasons can be summarized by the USECA properties we introduced previously [13]: Uniform access to heterogeneous data, Scalability to very large networks, Evolution of data sets, Composability of software, and Autonomy of information producers as well as information consumers. Using the DIOM model [13], we simplify information gathering and delivery, providing uniform access and composability of software. However, the autonomous data sources may not support the same abstraction, making their composition and change notification of individual evolution difficult. Therefore, the first research challenge of CQ work is to simplify the input to the continual query processing so that autonomous data sources need only to support a simple data exchange format. This is addressed by our differential relations that are simple to implement.

Another fundamental problem with the active database and view materialization approaches is that they operate on the base data, therefore have difficulties with scalability as the number of information producers (servers) and information consumers (clients) grow exponentially, like the case with Internet. To make the system more scalable, we need to shift the processing to the client side as much as possible. The second research challenge of CQ work is therefore to store the query results at the client side and perform the differential re-evaluation at the client. This is our motivation for proposing the *differential re-evaluation algorithm* (DRA), described in Section 5, for processing continual queries. DRA builds on previous research in differential files [25], hypothetical relations [8, 26], and incremental updating materialized views [3, 23, 29].

A third important research challenge is the support of general updates, including insertions, deletions, and modifications. As mentioned above, continuous queries [27] only supports append operations to the data. We capture the general updates to the data sources of information producers in the differential relations and send them to the information consumers for differential re-evaluation of the continual queries. Our simple definition of CQ makes the DRA algorithm workable, compared to attempts to extend continuous queries directly [2].

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 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 [28, 17] 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 ϵ-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 ϵ-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 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 terminology is used in this paper to specify continual queries, to record and manipulate changes by differential relations and associated operators. This notation does not constrain our algorithm and solutions to relational database management systems. In DIOM, information consumers formulate queries with GUI tools, which are then translated into appropriate query languages for backend processing, such as SQL. On the other side, information producers need only to generate the differential relations, which are simple tables of update operations, to communicate with the consumers.

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, outer-join, union, and difference operators by σ, Π, $\bowtie$, `Outerjoin`[1], $\bigcup$, 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

[1] `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].

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 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:

$\texttt{tid}^{<old>}$	$\texttt{Name}^{<old>}$	$\texttt{Price}^{<old>}$	$\texttt{tid}^{<new>}$	$\texttt{Name}^{<new>}$	$\texttt{Price}^{<new>}$	`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`, `insertions` 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. $\texttt{combine}(R, \Delta R) = (R - \texttt{deletions}(\Delta R)) \bigcup \texttt{insertions}(\Delta R)$. □

These high-level projection operators are derivations of the basic operations for differential relations. We define them here mainly for presentation convenience, because they are used frequently in our differential re-evaluation strategies for continual query processing.

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

deleteLog(ΔStocks)

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

insertions(ΔStocks)

tid	Name	Price
101088	MAC	117
120992	DEC	149

Stocks' = combine(Stocks, ΔStocks)

tid	Name	Price
120992	DEC	149
032090	ODI	120
041977	USL	100
101088	MAC	117
⋮	⋮	⋮

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 $\mathtt{Diff}(R_1, R_2)$ is defined as the $\mathtt{Outerjoin}_{\mathtt{tid}}$ of tuples in $R_1 - R_2$ and $R_2 - R_1$ under the join condition "$\mathtt{tid}^{<old>} = \mathtt{tid}^{<new>}$".

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

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 $\mathtt{old}(\mathcal{R}) \cup \mathtt{new}(\mathcal{R})$. □

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 $\mathtt{combine}(R_i, \Delta R_i)$. Let $\mathrm{Q}(R_1, \ldots, R_n)$ denote an arbitrary continual query and $\mathrm{SL} = \{[R_i, \Delta R_i] \mid (1 \leq i \leq k,\ k \leq n)\}$ denote the differential substitution list. The `Propagate` operator is defined as follows:

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

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

We may also denote $\mathtt{Propagate}(\mathrm{Q}(R_1, \ldots, R_n); \mathrm{SL})$ by $\Delta\mathrm{Q}(R_1, \ldots, R_n)$.

Example 3 Consider the query Q: $\sigma_{\mathtt{price}>120}(\mathtt{Stocks})$ as a continual query. Let E_i be the ith execution of Q at time t_i and $\mathrm{E}_i(\mathrm{Q}, t_i)$ 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 $\mathrm{Q}(\mathtt{Stocks}) = \mathrm{E}_i(\mathrm{Q}, t_i)$ 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 $\mathrm{Q}(\mathtt{Stocks'}) = \mathrm{E}_{i+1}(\mathrm{Q}, t_{i+1})$ denote the result of the current execution of Q, over the relation `Stocks'`.

Based on Definition 5, to express how the result $\mathrm{E}_i(\mathrm{Q}, t_i)$ may change after the updates by the transaction `T`, we may simply compute $\mathtt{Propagate}(\mathrm{Q}(\mathtt{Stocks});[\mathtt{Stocks},\Delta\mathtt{Stocks}])$, the difference between the result relation before the updates: $\mathrm{Q}(\mathtt{Stocks})$ and the result relation after the update: $\mathrm{Q}(\mathtt{Stocks'})$.

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

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

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

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

The differential result ΔQ($\texttt{Stocks}$) 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($\texttt{Stocks}$) = $\texttt{Diff}$(Q($\texttt{Stocks}$),Q($\texttt{Stocks'}$))

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

□

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

Observe that when (i)the size of the source relation $\texttt{Stocks}$ is large, (ii)the selectivity factor of the query Q over $\texttt{Stocks}$ 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 $\Delta\texttt{Stocks}$ instead. It is because computing $\texttt{Propagate}(\sigma_{\texttt{price}>120}(\texttt{Stocks});[\texttt{Stocks},\Delta\texttt{Stocks}])$ is functionally equivalent to the evaluation of the differential query: $\sigma_F(\Delta\texttt{Stocks})$, where F denotes the predicate $\texttt{price}^{<old>} > 120 \wedge \texttt{price}^{<new>} > 120 \wedge \texttt{timestamp} > t_i$[2]. This is true even when the user needs the complete composite set of the query result, because computing the union of Q($\texttt{Stocks}$) and $\texttt{insertions}(\sigma_F(\Delta\texttt{Stocks}))$ is cheaper than recomputing the expression $\sigma_{price>120}(\texttt{Stocks'})$ 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 $\texttt{deletions}(\sigma_F(\Delta\texttt{Stocks}))$. In general, for any continual query Q over the relation R, let $\mathrm{E}_i(\mathrm{Q},t_i)$ 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:

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

[2]The condition $\texttt{timestamp} > t_i$ is appended by the CQ manager for incorporating correct amount of updates and limiting the query search space. Section 6.4 provides some further discussion.

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; SL) 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; SL) 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 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:

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

Proposition 1 Propagate($\sigma[R;F];[R,\Delta R]$) $\equiv$ 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 DiffSelect(ΔR; 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 $\mathcal{R}$, 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:

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

The following proposition shows that DiffProj is an efficient differential form that can be used to implement 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 Propagate($\Pi[R;\mathbf{X}]$; $(R, \Delta R)$) $\equiv$ DiffProj[ΔR; old($\mathbf{X}$)$\bigcup$new($\mathbf{X}$)].
Proof (see Appendix A).

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

$\mathrm{Q}_1 = \Pi[\texttt{Stocks};\ \texttt{Name}]$.

$\mathrm{Q}_2 = \Pi[\sigma_{\texttt{price>120}}(\texttt{Stocks});\ \texttt{Name}]$.

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,ΔStocks])
  = DiffProj[ΔStocks; Name].

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

- $\texttt{Propagate}(\Pi[\texttt{HighStocks}; \texttt{old(Name)}\cup\texttt{new(Name)}]; [\texttt{HighStocks},\Delta\texttt{HighStocks}])$

  $= \texttt{DiffProj}[\Delta\texttt{HighStocks}; \texttt{old(Name)}\cup\texttt{new(Name)}]$ (by Proposition 2)

  $= \texttt{DiffProj}[\texttt{Propagate(HighStocks;[Stocks,}\Delta\texttt{Stocks]); old(Name)}\cup\texttt{new(Name)}]$ (by Lemma 2)

  $= \texttt{DiffProj}[\texttt{DiffSelect(}\Delta\texttt{Stocks); old(Name)}\cup\texttt{new(Name)}]$. (by Proposition 1)

$\texttt{tid}^{<old>}$	$\texttt{Name}^{<old>}$	$\texttt{tid}^{<new>}$	$\texttt{Name}^{<new>}$
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

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

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

- $\texttt{DJoin}_{P_{join}}(\Delta R_1, \Delta R_2)$
  $= \texttt{Outerjoin}_{P(tid_1,tid_2)}(\bowtie_{P^{<old>}_{join}}(\texttt{deleteLog}(\Delta R_1),\ \texttt{deleteLog}(\Delta R_2)),$
  $\bowtie_{P^{<new>}_{join}}(\texttt{insertLog}(\Delta R_1),\ \texttt{insertLog}(\Delta R_2)).$ □

Proposition 3 Let the differential substitution list SL be $\{[R_1, \Delta R_1]$,$[R_2, \Delta R_2]\}$.

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

Proof (see Appendix A).

This development can be generalized to the join of an arbitrary number of base relations. Let 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	$\texttt{DiffJoin}(R_1, R_2, R_3, \ldots, R_n) =$
1	0	0	$\texttt{DJoin}(\Delta R_1, R_2, R_3, \ldots, R_n)$
0	1	0	$\cup\ \texttt{DJoin}(R_1, \Delta R_2, R_3, \ldots, R_n)$
0	0	1	$\cup\ \texttt{DJoin}(R_1, R_2, \Delta R_3, \ldots, R_n)$
1	1	0	$\cup\ \texttt{DJoin}(\Delta R_1, \Delta R_2, R_3, \ldots, R_n)$
1	0	1	$\cup\ \texttt{DJoin}(\Delta R_1, R_2, \Delta R_3, \ldots, R_n)$
0	1	1	$\cup\ \texttt{DJoin}(R_1, \Delta R_2, \Delta R_3, \ldots, R_n)$
1	1	1	$\cup\ \texttt{DJoin}(\Delta R_1, \Delta R_2, \Delta R_3, \ldots, R_n)$

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,

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

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

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

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

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

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

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 $\Pi_X(\sigma_F(R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_n))$.

Moreover, in the Diorama system, we apply several conventional query optimization techniques used in both centralized and distributed environment to further reduce the cost of DiffJoin operation. For example, given a continual query Q, denoted by $\Pi_X(\sigma_F(R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_{50}))$, which request access to 50 classes/relations from 20 different information sources. Assume only two relations (say R_1 and R_2)

have been updated since the last execution of the query Q. By using the commutativity of selection and projection over joins and associativity of joins, this query Q will first be decomposed into the following two subqueries: $SubQ_1 = \Pi_{X_1}(\sigma_{F_1}(R_3 \bowtie R_4 \bowtie, \ldots, \bowtie R_{50}))$ and $SubQ_2 = \Pi_{X_2}(\sigma_{F_2}(R_1 \bowtie R_2 \bowtie Result(SubQ_1)))$, where $X = X_1 \cup X_2$ and F_1 is a selection condition over $R_3, R_4, \ldots, R_{50}$ and F_2 is a selection condition over R_1, R_2, and the result of $SubQ_1$. Now the evaluation of the `DiffJoin` operator over Q is reduced to `DiffJoin` over the subquery $SubQ_2$. The system performance for processing this CQ will be greatly improved, because (i)the evaluation of $SubQ_1$ can be done directly against the previous execution result of Q cached at the client side, and (ii)comparing with the original query Q, the size of $R_1 \bowtie R_2 \bowtie Result(SubQ_1)$ is much smaller both in cardinality and in degree than $(R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_{50})$. This approach is particularly benefical when the selectivity of F_1 is high and the project list X_1 is small.

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 $T_{ij} = 1$. 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 $\texttt{DJoin}(S_1 \bowtie S_2 \bowtie, \ldots, \bowtie S_n)$.

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 $E_i(Q, t_i)$, without deletion notification, the result to be returned can be computed by $\sigma_{timestamp>t_i}(\texttt{insertions}(\Delta R_Q))$.

- If the user needs to see the complete set of the result matching the query, we return $E_i(Q,t_i) \bigcup \sigma_{timestamp>t_i}(\texttt{insertions}(\Delta R_Q))$.

- 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

6.1 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 interoperation 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.

6.2 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.

6.3 Asynchronous Evaluation of CQ Conditions

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`|.

6.4 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 $Q(S_i)$ 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.

6.5 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 [12]) will be used to catch 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.

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 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 (delta) relations (Section 4) and the DRA (Section 5), we also discuss a number of implementation issues such as asynchronous evaluation of CQ conditions and strategies for garbage collection of differential relations (Section 6).

DRA and CQ satisfy the USECA properties for information update monitoring on the Internet. Uniform access and composability are supported by DIOM [13], 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.

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 diom approach to large-scale interoperable information systems. Technical report, TR95-16, Department of Computing Science, University of Alberta, Edmonton, Alberta, March 1995.

[13] L. Liu and C. Pu. The distributed interoperable object model and its application to large-scale interoperable database systems. In *ACM International Conference on Information and Knowledge Management (CIKM'95)*, Baltimore, Maryland, USA, November 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. $\texttt{attach}[\texttt{insertions}(\Delta R), \texttt{new}(\mathcal{R})] = \texttt{insertLog}(\Delta R)$.

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 Let the differential substitution list SL be $\{[R_1, \Delta R_1],\ldots,[R_n, \Delta R_n]\}$ $(n \geq 1)$. The following equations hold:

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

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

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 Let the differential substitution list SL be $\{[R_1,\Delta R_1],[R_2,\Delta R_2]\}$.

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

Proof

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

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

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

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

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

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

By Definition 5 and Definition 4,

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

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

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

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

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

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

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

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