Continuous Queries in Oracle

Andrew Witkowski, Srikanth Bellamkonda, Hua-Gang Li, Vince Liang, Lei Sheng, Wayne Smith,

Sankar Subramanian, James Terry, Tsae-Feng Yu

Oracle USA 400, Oracle Parkway, Redwood Shores, CA 94065, U.S.A.
First.Last@oracle.com

Abstract

This paper describes Continuous Queries (CQ) in Oracle RDBMS, a feature that incorporates stream and complex event processing into an RDBMS, the first such attempt in commercial databases. The feature is based on the concept of query difference and allows us to monitor real time changes to the query as the result of changes to its underlying tables. The result of a continuous query can be deposited into historical tables or queues for further asynchronous de-queuing, or can invoke a synchronous trigger for procedural processing. The main contribution of our CQ engine is that it allows us to react to complex scenarios of changes to data such as mixed INSERT, DELETE and UPDATE changes, unlike the existing stream processing systems that deal with INSERTS only. We support a wide range of query shapes including inner, semi and anti-joins, aggregates and window functions. More details are given to the efficient computation of query difference for general cases and their optimizations based on semantic constraints. They are shown to improve the response time for practical cases by more than an order of magnitude. We also show how delaying CQ re-computation can improve its performance by batch processing the changes to the base tables.

1. Introduction

There has been a significant interest in processing, within an RDBMS streams of changes to the data and reporting when the result of a query defined over the data changes. These queries are referred to as the continuous queries (CQ) since they continually produce results whenever new data arrives or existing data changes. The interest is driven by sensor and event data processing (RFID in particular), real-time reporting of Key Performance Indicators for Business Intelligence, and by security monitoring to discover sequence of prohibited or dangerous events. One primary reason for an RDBMS solution for stream computation is the declarative power of SQL that enables rapid and uniform (via a single language) application development and offers potential for optimizations of multiple stream queries using multi-query optimization techniques.

Stream processing has been researched extensively in the recent literature ([Sul96], [CC+02], [CC+03], [CD+00], [LPT99], [JMS95], [ABB+03], [SLR94], [CJSS03], [MSHR02]). It also has some prototype implementations ([CC+02], [CJSS03], [ABB+03], [CC+03]). In most of these works, streams are treated as append-only tables or transient queues and queries are expressed with SQL extensions using window constructs on the streams. Stream processing attracted previous interest in form of active databases [PD99] (driven by triggers and procedural programming) and in form of materialized views ([JMS95]).

In contrast to the literature where the sources of continuous queries are objects called streams, the sources for our CQ engine are transactions to the relational tables that can insert (like in streams), delete or modify data. Continuous query is defined as a relational set difference between the query result at time t_1 and time t_2 ($t_2 > t_1$) and is best explained using the analogy of materialized views (MVs). Given query Q and its materialized image MV_1 at time t1, we accept committed DML changes to the Q's tables, re-compute MV_2 at time t_2, and return query delta $\Delta Q = MV_2 - MV_1$. The notion of query delta is very natural to monitoring applications that define data of interest using a SQL query (e.g., customers with balance less than 0) and need to monitor records entering the set (the insert delta), records leaving it (the delete delta), and records in the original set that change (the update delta). Frequency of re-computation is defined by the user and similarly to MVs, it can be *on commit*, *on demand* and *periodic*. Continuous query is a transactionally consistent query delta i.e., only committed changes are emitted by it. This allows us to build continuous queries that are restartable and consistent across system crashes.

The continuous results of CQ can be deposited either to a queue or appended to a table. Users are expected to asynchronously de-queue the results from the queue or retrieve it directly from the table. Management of the destination (queue or table) is left to the user since he/she may decide to preserve it for historical purposes. We also provide for procedural extensions for CQ. The process of emitting data from a CQ can invoke an RDBMS trigger or a user defined callback where further programmatic processing can be done.

Not surprisingly, we use algorithms similar to incremental refresh of materialized views to compute query delta. Our algorithms, however, have many novel features and cover a larger set of queries than the published ones [GMS93], [GHQ95], [RB+98]. We provide optimization of the query delta computation for queries with joins which is based on constraints. This improves performance of the existing methods by more than an order of magnitude. We also investigate usage of the snapshot features of Oracle RDBMS to retrieve the pre-update state of the tables for delta computation expressions. Previously, the pre-update states had to be computed using SQL queries over tables and their logs. To support non-event detection, we extend the algorithms for NON EXIST sub-queries. For real-time Business Intelligence and security applications we add incremental refresh of window functions with a monotonically increasing attribute in their ORDER BY. To efficiently process these CQs, we introduce a new

VLDB '07, September 23-28, 2007, Vienna, Austria.

relational constraint of monotonically increasing attribute; in our case the time-stamp of events. The constraint is satisfied by many applications that store historical, time-stamped data in RDBMS.

Due to the real-time nature of most CQs, performance of refresh for small OLTP transactions is critical and we address it with novel refresh algorithms. Equally important is a non-blocking (with respect to OLTP commits) nature of CQ. Hence our refresh is asynchronous and avoids any locks on the base data. We show how this can be done using new logging structures.

Finally, we present performance of our algorithms and compare it to the existing ones. This not only demonstrates (expected) improvements due to optimized refresh expression based on constraints, but also discovers an unexpected fact that pre-update state retrieval using rdbms-native snapshot technology does not significantly outperform re-computing the state using SQL queries.

The rest of the paper is organized as follows. Section 2. gives example schemas that are used throughout the paper. Section 3. discusses basic concepts and language elements for CQs. In Section 4. , we describe the sources of changes, clogs. Section 5. describes CQ refresh and concurrency control. Section 6. presents the computation of continuous join queries, followed by the optimizations by exploiting constraints in Section 7. Section 8. and 9. discuss the computation of continuous queries with aggregations and window functions respectively. In Section 10. , we study the performance of our CQ computation model. Section 11. concludes the paper.

2. Example Schemas and Example CQs

We use two schemas in this paper. The *e-store schema* consists of three tables: `orders(oid,time,uid)` records user orders, `orderline(oid,itemid,price)` records items purchased in each order, and `users(uid,address,name)` records users and their attributes. Our example query tracks items purchased by an under-age user:

```
Q1  :
SELECT ol.item, o.oid, u.name
FROM orderline ol, order o, user u
WHERE ol.oid=o.oid AND u.uid=o.uid AND u.age<18;
```

The *banking schema* consists of only one table `banking_transactions(acct,time,amt)` that records amount of deposits and withdrawals from an account. Our example query will monitor overdrawn accounts:

```
Q2  :
SELECT acct, sum(amt) amt
FROM banking_transactions
GROUP BY acct
HAVING sum(amt) < 0;
```

3. Basic Concepts and Language Elements

Query Delta

Consider a relational query Q over tables T1,..,TN. As the tables undergo transactional changes, result of Q changes as well. Continuous Query is a transactionally consistent sequence of changes to Q. Let $Q1$ and $Q2$ be the result set of Q at times t_1 and t_2 correspondingly. The delta change to Q, ΔQ, is

```
Q3  ΔQ = (Q2 MINUS+ Q1) UNION ALL (Q1 MINUS+ Q2)
```

The first term, ($Q2$ MINUS$^+$ $Q1$), represents new rows in the results set and we refer to them as the *insert delta*, Δ^+Q. The second, ($Q1$ MINUS$^+$ $Q2$), represents rows that disappeared from the result set, and we refer to them as the *delete delta*, Δ^-Q.

The MINUS$^+$ operator above doesn't remove duplicates unlike the MINUS operator in ANSI SQL as otherwise we would lose useful information. For example {(1,1,1), (1,1,1), (2,2,2)} MINUS$^+$ {(1,1,1), (2,2,2)} = {(1,1,1)} and not, as in relational MINUS, the empty set {}.

In the above, an update to Q result set is typically modelled as a delete followed by an insert and this is not intuitive. Nevertheless, if Q has a primary key, then, for each value of the primary key, we can group its deleted-inserted rows, and return a single row with updates to the non-primary key columns. We refer to this as *update delta*, Δ^uQ. Updates to the primary key will continue to produce, as before, delete and insert delta.

It is useful for the Δ^uQ to return not only the new values of the updated columns, but their old values as well. We provide a new sql function for this, cq_old_value(column). To distinguish the three deltas, there is a new function, cq_delta(), returning 'I', 'D', and 'U' for Δ^+Q, Δ^-Q, Δ^uQ respectively.

Continuous Query Definition

Continuous Query is a new SQL object that is defined similar to views. It specifies the query Q, constraints on Q such as its primary or foreign key, destination where ΔQ is deposited, type of the delta to be returned, and computational characteristics (when and how often to compute CQ results). For example, the following CQ monitors accounts with negative balance, and returns into destination table `dest_table` Δ^+Q, Δ^-Q, Δ^uQ deltas.

```
Q4
CREATE CONTINUOUS QUERY negative_balance_cq
  PRIMARY KEY (acct)
  COMPUTE ON COMMIT
  DESTINATION dest_table
SELECT acct, sum(amt) amt
FROM banking_transactions
GROUP BY acct
HAVING sum(amt) < 0;
```

The CQ object itself, `negative_balance_cq` above, is not queryable. Its destination, however, is. Note that the CQ has a primary key, hence can deliver Δ^uQ. We note that the primary key applies to defining query and not to the destination.

Destinations of CQ

The destination of a CQ can either be a relational table or an Oracle queue. In the former case, the shape of the table must be compatible with the CQ query. A queue is also implemented as a table, but in addition, it has an API that allow users to subscribe to the queue, de-queue messages (rows) stored there, and to manage the underlying tables, for example deleting messages from them after all the subscribers have consumed them.

In order to programmatically process the delta rows, we support row triggers on CQ. The triggers are semantically treated as if an INSERT operation occurred on an object represented by the CQ and behave the same as row triggers on tables . Hence a trigger is invoked for each row of the delta. Observe that CQ does not need a destination table to have a trigger. As usual, trigger procedures have access to all columns of CQ. For example, users can write a procedure, `process_negative_balance`, to alert a branch manager about customer defaulting on his balance:

```
Q5
CREATE TRIGGER negative_balance_trigger
  ON negative_balance_cq FOR EACH ROW
  BEGIN process_negative_balance(acct, amt) END;
```

CQ Computation

CQ computation is performed asynchronously to the transaction traffic over the base tables and without locking them, hence, its impact on OLTP is minimal (given enough resources for CQ computation). Still the CQ produces transactionally consistent deltas – a non-trivial task explained in Section 5.

The computation of CQ can be invoked in three modes: on commit of a transaction that modified its base tables, on demand by calling an explicit `cq_recompute(<cq_name>)` API, or periodically with a given start time and periodicity. For example, CQ `negative_balance_cq` can recomputed every hour using:

```
Q6
ALTER CONTINUOUS QUERY negative_balance_cq
  COMPUTE START WITH '01-01-2007' PERIOD 1 HOUR;
```

In many cases application needs to react to the effect of each transaction but may tolerate a delay in notification. Moreover it is more performant to compute CQ refresh for a batch of transactions at one time rather than processing them individually. Therefore, for on demand and periodic computations we provide two modes of CQ delta calculation: transactional and batched. The former applies the change logs in transaction increments and in the transaction order. The result is similar to the result of "ON COMMIT" computation but with a delay. The latter produces the result as the delta between state of the query at the last refresh time and its current state. For example, to compute Q4 transactionally every 1 hour we say:

```
Q7
ALTER CONTINUOUS QUERY negative_balance_cq
  COMPUTE TRANSACTIONAL DELTA
  START WITH '01-01-2007' PERIOD 1 HOUR;
```

Note that if Q7 were computed in batch (and not transactional mode), we could miss periods when user balance went below 0.

Managing of CQ.

Continuous Queries are created with "CREATE CONTINUOUS QUERY" statement and managed with a set of "ALTER CONTINUOUS QUERY" statements. As expected, CQs can be in three states: ACTIVE, SUSPENDED, and INACTIVE. In the SUSPENDED state, query delta is not produced, however, the database monitors and remembers changes to the table and will apply them to produce delta when CQ is activated. An INACTIVE CQ just records its own metadata and doesn't require monitoring of the changes to base tables. Activation of such CQ is equivalent to re-creating it. For example,

```
Q8
ALTER CONTINUOUS QUERY negative_balance_cq
SUSPEND;
```

suspends the CQ Q4.

Using ALTER statements users can change other attributes of CQ like its destination, computation options, etc.

Supported Query Shapes.

Some of our CQ computation algorithms originate from incremental refresh of MVs [BLT86, GMS93, PSCP02]. We support single block queries with joins (CQJ) and CQJ with algebraic aggregation (CQJA). Extensions to CQJ algorithms allow us to support query blocks with semi-joins CQJS (EXISTS or IN sub-queries) and blocks with anti-joins CQAJ (NOT EXISTS or NOT IN sub-queries). CQJS algorithms are simple extensions to CQJ. Efficient CQJA are challenging, however, due to space limitations, the algorithms are not given here. For CQJ, CQJS, and CQJA we significantly extend known MV algorithms – see Section 6. and 8.

We propose algorithms to support a very useful class of continuous queries – single block queries with joins and window functions (CQW). CQW is frequently used for sensor processing where comparison of current and past measures is critical, for example, detection of temperature raise above a threshold within a given time. Here, refresh is provided for windows over a monotonically increasing attribute. A new constraint is added to RDBMS for CQW computation.

In addition to CQJ, CQJA, CQJS, CQAJ and CQW, we support their composition with UNION ALL and simple filtering in an outer query block.

New Constraints for CQ

In many cases measurements are always produced (and stamped) with increasing time values. The monotonically increasing constraint guarantees that column values in a table or its partition can only increase. The constraint has two modes: STRICTLY INCREASING and INCREASING with obvious semantics. The (strictly) increasing constraint is equivalent to maintaining a max value for a column over the life span of a table or its partition, and verifying that new rows, or updated columns will always have value (greater) no less than this maximum. This constraint has a natural syntax:

```
CONSTRAINT const_name INCREASING column
       [PARTITION BY column] ON table
```

The constraint is used for computation of CQW.

4. Sources of changes for CQ – clogs

Changes made by transactions are recorded in auxiliary tables called clogs. There is one clog for each table participating in a CQ, and by convention its name is `clog_<table_name>`, for example `clog_orderline`. The clogs store pre-images and post-images of the rows modified by the transactions. As expected, INSERT, DELETE, UPDATE DML produce post-images, pre-images, and pre- and post- images, respectively. We note that a single update is represented as a delete of the updated row followed by an insert of the row with new values. Clogs contain the original columns of the tables plus several accounting columns. The `old_new_mark` column indicates if the row is a pre-image (marked by 'O') or post-image (marked by 'N') and `dml_type` indicates the DML that caused the change ('I','D','U' for insert, delete and update respectively). The `trans_scn` column records the System Commit Number (SCN) of the originating transaction. In an Oracle RDBMS, every transaction is given a unique stamp called SCN. In general SCNs form an increasing sequence of integers. SCN serves an important role for concurrency control, recovery and for retrieval of transactionally consistent versions of the database. SCNs are recorded in our undo-logs and are used for computing past versions of the tables as of given SCN by applying the undo-images to the current image of the table [ORA10GC].

Our clogs are populated shortly after the transaction has committed, hence they do not block OLTP update traffic. Note that we get the changes from our transaction layer, which are actually from undo/redo logs.

During CQ refresh we retrieve records from clogs pertaining to the refresh by selecting those with `trans_scn` between the SCN of the last refresh of the CQ and the current SCN. They represent changes to the tables that occurred since last refresh.

5. CQ refresh and Concurrency Control

Our challenge was to not slow down the transaction traffic by CQ (possibly thousands of them) re-computation. Critical and negative factors include locking of objects that prevent their OLTP updates and synchronous refresh that holds transaction commit points. We address them as follows:

Our CQ engine does not place read locks on the base tables or their clogs. Instead, we rely on Oracle RDBMS's versioning system to provide a consistent view of the database for the refresh expressions. A table T in a query can be qualified with an SCN number and our storage layer will retrieve the version of T as of the SCN (provided that the undo-log is large enough to store undo-records back to that SCN). As explained in the following section, Oracle CQ refresh expressions consist of SQL statements that compute query delta and pass it on to the destination tables or triggers. In some cases refresh expressions are comprised of multiple SQL statements. At the refresh time, we get current system SCN, `current_scn`, and decorate all tables in the refresh expressions with it, hence computations are done on a transactionally consistent view of the database. For simplicity, in our examples we do not show this decoration since it is applied universally.

We remember in the CQ metadata, the SCN of its last refresh `last_refresh_scn`. This SCN is used to retrieve from the clog records that represent table changes since the last refresh. Hence retrieval from clogs always has a predicate on clog.trans_scn:

```
clog.trans_scn BETWEEN last_refresh_scn AND
current_scn
```

Again since this predicate is applied to all clogs, we do not show it in the refresh expressions.

As mentioned before our CQ computation is asynchronous with the transaction traffic, therefore refresh doesn't hold up the commit points. Furthermore, our batched refresh mode, allows us to use efficient query plans suited for large transaction deltas and minimize the overhead of CQ startup.

6. Computation of CQJ

In this section, we present the incremental computation of CQJ. First, we describe general incremental computation algebra and then show how to generate efficient SQL queries for different DML scenarios such as inserts-only, deletes-only, and mixed DML.

Incremental Computation Algebra

Consider a CQ joining tables T1 and T2: CQ = T1 >< T2. Let pre(CQ) and pre(Ti) denote the initial, before the update state (image) of the CQ and table Ti, respectively; dlt(Ti) denote the updates (e.g., inserts) to table Ti, and pst(CQ) and pst(Ti) denote the state (image) of CQ and table Ti after the update, respectively. Then:

```
Q9
pst(CQ) = pst(T1) >< pst(T2)
  = (pre(T1)+dlt(T1)) >< (pre(T2)+dlt(T2))
  = pre(T1) >< pre(T2) + pre(T1) >< dlt(T2) +
    dlt(T1) >< pre(T2) + dlt(T1) >< dlt(T2)
  = pre(CQ) + pre(T1)><dlt(T2)+ dlt(T1)><pst(T2)
```

Hence, query delta, dlt(CQ) is

```
Q10 dlt(CQ) = pre(T1)><dlt(T2)+ dlt(T1)><pst(T2)
```

Note that, following Q9, dlt(CQ) can also be expressed as:

```
Q11 dlt(CQ) = pst(T1)><dlt(T2) + dlt(T1)><pre(T2)
Q12 dlt(CQ) = pst(T1)><dlt(T2) + dlt(T1)><pst(T2)
            - dlt(T1)><dlt(T2)
```

Choice of the incremental computation algebra expression will affect the performance of CQ refresh. Typically, pre-images must be recomputed from post-images and table deltas, hence we prefer to minimize their usage or re-compute them only for smaller tables. Ideally, we would generate all possible expressions, then choose the least expensive one, but this computation is exponential and prohibitively expensive when the number of involved tables is large. We found in most of our internal experiments that Q10 and Q11 are more performant than Q12. To choose between Q10 and Q11 we chose the one that computes the pre-image of the smaller table: Q10 wins if *cardinality*(T2) > *cardinality*(T1).
In general, for a CQ joining N tables, we use

```
Q13 dlt(CQ) =
      dlt(T1)><pre(T2)><pre(T3)>< .. pre(TN) +
      pst(T1)><dlt(T2)><pre(T3)>< .. pre(TN) +
       ...
      pst(T1)><pst(T2)><pst(T3)>< .. dlt(TN)
```

Again, since inner joins are commutative Q13 is applicable to any permutation of the tables and we choose one that requires fewer pre-image computations of larger tables. Note that if only a subset of tables changed, refresh expressions account for them. For example, if only T1 and T2 changed in Q13, it can be reduced to:

```
      dlt(T1)><pre(T2)><pre(T3)>< ... >< pre(TN) +
      pst(T1)><dlt(T2)><pre(T3)>< ... >< pre(TN)
```

Hence, it is also critical to know which tables are changed since last refresh of a CQ. In the following we will show how to implement the incremental computation algebra in SQL for different update scenarios.

Incremental Computation in SQL

INSERT only case

When inserts are the only DMLs which happened on base tables, CQJ delta is composed of only insert deltas and will be tagged as 'I' delta type. This CQJ delta can be computed using expression Q13 that requires computing the pre-images of N – 1 tables for a CQJ involving N tables. To implement Q13 in SQL, we investigated two approaches to compute the pre-images of the involved tables. One is to use a "NOT IN" subquery and the other is to obtain pre-images directly from the storage layer in Oracle RDBMS, by requesting a version of the table as of some previous time or previous SCN. In the latter case, Oracle RDBMS applies its internal undo logs to the table and rolls it back to that SCN. The assumption is that storage layer does this very efficiently since it is the basis for Oracle concurrency control. Versioning, aka

flashback, is available from Oracle SQL using the SCN sub-clause on the table reference, for example: orders AS OF SCN (<scn-value>). Typically, for a relatively small amount of updates on the base tables, "AS OF SCN" approach outperforms the "NOT IN" subquery approach if there is an index available and refresh uses it. If there is no index available, the performance between the two is similar. For a relatively large amount of updates on the base tables, flashback may not be able to provide enough undo for rollback. In that case flashback based refresh would fail. Detailed performance studies between these two approaches is presented in Section 10.

Next, we give concrete examples of these two approaches using a continuous join query CQJ_2 joining two tables T1 and T2: CQJ_2 = T1 >< T2.

"NOT IN" Subquery Approach

```
SELECT <cols of T1>,<cols of T2>,'I' dlt_type
FROM                  /* dlt(T1) >< pre(T2) */
  (SELECT <cols of T1> FROM CLOG_T1) DLT_T1,
  (SELECT rowid, <cols of T2> FROM T2) PRE_T2,
WHERE (join condition) AND PRE_T2.rowid NOT IN
       (SELECT rowid FROM CLOG_T2 )
UNION ALL
SELECT <cols of T1>,<cols of T2>,'I' dlt_type
FROM                  /* pst(T1) >< dlt(T2) */
  (SELECT <cols of T1> FROM T1) PST_T1,
  (SELECT <cols of T2> FROM CLOG_T2) DLT_T2
WHERE (join condition);
```

"AS OF SCN" Approach

```
SELECT <cols of T1>,<cols of T2>,'I' dlt_type
FROM                  /* dlt(T1) >< pre(T2) */
  (SELECT <cols of T1> FROM CLOG_T1) DLT_T1,
  (SELECT <cols of T2> FROM T2 AS OF SCN :VSCN
  ) PRE_T2,
WHERE (join condition)
UNION ALL
SELECT <cols of T1>,<cols of T2>,'I' dlt_type
FROM                  /* pst(T1) >< dlt(T2) */
  (SELECT <cols from T1> FROM T1) PST_T1,
  (SELECT <cols from T2> FROM CLOG_T2) DLT_T2
WHERE (join condition);
```

In the above VSCN is the SCN of the last refresh of the CQ.

DELETE only case

When deletes are the only DMLs happened on base tables, then CQJ delta is composed of only delete delta and will be tagged as 'D' delta type. The implementation of the incremental algebra expression in Q13 for the deletes-only case is the same as that for the inserts-only case. Hence, we will omit the details here.

Mix of INSERTS, DELETES, and UPDATES

CQ refresh becomes complicated when tables go through a mix of DMLs. CQ delta in this case can consist of insert, delete and update deltas. We typically model "update" as delete followed by insert and log it in clog as two rows. For the simplicity of presentation, we model update deltas as an insert and delete pair. They can be combined into a single update delta provided that the CQ has a primary key that allows us to clump deletes and inserts corresponding to the same primary key.

Similar to the above two cases, CQ refresh with mixed DMLs is also based on the incremental computation expression on Q10 or its general form Q13. Let us consider an example continuous join query CQJ_2 joining two tables: CQJ_2 = T1 >< T2. Before we show how to process the CQJ_2 refresh for the case with mixed DMLs, we rearrange the terms in Q13 making use of the commutative characteristic of inner join to get:

```
dlt(CQJ_2) = pre(T1)><dlt(T2) +  dlt(T1)><pst(T2)
```

We first process deletes (rows marked "U" and "O") from clog to generate delete delta. We then process inserts (rows marked "N") from the clog to generate the insert delta. In the end, we combine these deltas to report update delta. The following terminology will be needed when describing the CQ refresh for the case with mixed DMLs. del(Ti) refers to the delete delta on table Ti. ins(Ti) refers to the insert delta on table Ti. pst'(Ti) refers to the post-image after deletes on table Ti. pre'(ti) refers to the pre-image before inserts on table Ti. Note that pst'(Ti) = pre'(Ti) = pst(Ti) – ins(Ti), pre(Ti) = pst(Ti) – ins(Ti) + del(Ti) = pst'(Ti) + del(Ti). We then have two deltas to process:

- the delete delta

```
Q14 del(CQJ_2) = pre(T1) >< del(T2) +
                 del(T1) >< pst'(T2)
```

- the insert delta

```
Q15 ins(CQJ_2) = pre'(T1) >< ins(T2) +
                 ins(T1) >< pst(T2)
```

As compared with either insert-only case or delete-only case, we can see that one more image of a table Ti needs to be computed based on the post-image of table Ti, i.e., pst'(Ti) or pre'(Ti), for notational convenience, denotes the intermediate image of table Ti. To implement the algebra expressions of Q14 and Q15, we can follow the same execution logic as what we do for insert only case or delete only case but with the adaptation to this particular scenarios of mixed DML changes. Due to the space limit, we will not give the detailed SQL statement.

7. Optimization of CQJ Computation

INSERT only case

Refresh expressions from Section 6. can be significantly simplified in the presence of Foreign-Primary Key (FK-PK) constraints. Consider a CQJ with N-tables. If all n-tables are modified, then dlt(CQ), is given by Q13. Suppose that tables T1, T2, T3 .. Tn are joined (in that order) via FK-PK constraints, i.e., T1.fk=T2.pk & T2.fk=T3.pk... Assume further that tables undergo inserts only. Observe that rows inserted in T2,..TN cannot join with pre(T1) (since all rows of pre(T1) had to join with pre(T2) and T2 has a primary index on the join key), hence:

```
pre(T1)><dlt(T2)><..pst(TN) = NULL
pre(T1)><pre(T2)><..dlt(TN) = NULL
```

Hence:

```
Q16 dlt(CQ) = dlt(T1)><pst(T2)><..pst(TN)
```

The above expression is significant since it reduces the number of refresh expressions from N to 1, and allows us to use only the post images of the tables.

For many practical cases, transactions come in FK-PK units. In this mode whenever we insert/delete rows on FK side, we always insert/delete corresponding row on the PK side. For instance, in the e-store application, users will place a single order for multiple items. In this case we insert a new "order" row into the orders table and insert into the orderline table rows corresponding to the items bought. In this case we have dlt(orders) and dlt(orderline) that exclusively join with each other. To get delta of the CQ it is sufficient to join dlt(orders) and dlt(orderline). There is no need to join dlt(orderline) to the entire pst(orders). In these cases Q16 can be simplified further to:

```
Q17 dlt(CQ) = dlt(T1)><dlt(T2)><..dlt(TN)
```

For another example, consider an e-store CQ which alerts us if an under-age user placed an order

```
Q18   :
  SELECT ol.item, o.oid, u.name
  FROM orderline ol, order o, user u
  WHERE ol.oid=o.oid AND u.uid=o.uid ANDu.age<18
```

We have a very efficient expression operating only on deltas as in:

```
 dlt(CQ) =
     dlt(orderline) >< dlt(orders) >< pst(users)
```

The problem is that we don't know if the delta on the FK side, dlt(orderline) in our example, joins only with delta on the PK side, dlt(orders) in the example. To detect this we could find out if the cardinality of the dlt(orderline) >< dlt(orders) is the same as dlt(orderline), i.e., all rows from dlt(orderline) join with dlt(orders) and hence cannot join with pre(orders). One way to discover it is via an outer join. If

```
Q19 :
dlt(CQ) =
  dlt(orderline) OUTER><dlt(orders)><pst(users)
```

has no anti-join tuples, then all rows from delta on the FK side (`dlt(orderline)`) joined with delta on the PK side, (`dlt(orders)`). If there are anti-join tuples, then we could join them with `pst(orders)` to get the full result.

Suppose query delta `dlt(CQ)` of Q19 will have an anti-join marker (returning 0 for inner and 1 for anti-join tuples). Based on this we will insert (using multi-table insert) the delta into two tables. The inner join tuples will be placed in the destination table, dest, and anti-join tuples will be placed in a temporary table, anti_join:

```
Q20 :
INSERT
  WHEN aj_mark=1 THEN INTO anti_join(item, oid)
  WHEN aj_mark=0 THEN INTO dest(item, oid, user)
SELECT ol.item, ol.oid, u.user,
 CASE o.oid IS NULL THEN 1 ELSE 0 END aj_mark
FROM dlt(orderline) ol,dlt(order) o, pst(user) u
WHERE ol.oid = o.oid(+) AND u.user(+) = o.user
   AND u.age < 18(+)
```

If anti_join table is empty, then there were no anit-join rows and we are done generating `dlt(CQ)`. Checking can be done either via a trigger on the anti-join table or by issuing a query: "select count(*) from anti-join".

If anti_join table is not empty, then we have to join all anit-join rows with pre(orders) and `pst(users)`. Observe that due to FK-PK relationship, we could join to `pst(orders)` rather than `pre(orders)`. In this case to complete generation of `dlt(CQ)` we would issue one more query:

```
 anti_join(MV) >< pst(orders) >< pst(users)
```

i.e.

```
Q21 :
INSERT INTO dest (item, oid, user)
SELECT ol.item, ol.oid, u.user,
  CASE o.oid IS NULL THEN 1 ELSE 0 END aj_mark
FROM anti_join ol, pst(order) o, pst(user) u
WHERE ol.oid = o.oid AND u.user = o.user
  AND u.age < 18
```

Queries Q20 and Q21 can be combined into a single query:

```
Q22 :
INSERT INTO dest (item, oid, user)
WITH
oj AS
 (SELECT ol.item, ol.odi, u.user,
    CASE o.oid IS NULL THEN 1 ELSE 0 END aj_mark
  FROM dlt(orderline) ol,dlt(order) o, pst(user)u
  WHERE ol.oid=o.oid(+) AND u.user(+)=o.user
  AND u.age < 18(+))
anti_join AS
 (SELECT * FROM oj WHERE aj_mark = 1)
SELECT item, oid, user FROM oj WHERE aj_mark = 0
UNION ALL
SELECT item, oid, user
FROM anti_join ol, pst(order) o, pst(user) u
WHERE ol.oid = o.oid AND u.user = o.user
  AND u.age < 18
```

The above query is efficient provided that the subquery OJ is small (or typically empty) compared to orderline and order tables. In addition, if anti_join is empty (or very small), the second branch of UNION ALL takes not time (or is very short). Section 10. shows that this formulation speeds up the general refresh algorithm of Q9 by orders of magnitude and the FK-PK optimized algorithm Q16 by an oder of magnitude.

DELETE only case

Assume DELETES only to T1, T2, .. Tn. If the foreign key 'CASCADE DELETE' option is active then deleting a row from T2 will delete the dependent foreign key rows in T1. If the option is not active, then we cannot delete rows from T2 that have foreign key in T1. Rows from T1 have to be removed first. Thus deleting rows from T2,..Tn which join with T1, will eventually propagate to T1. Hence deletes on T1, T2, ..Tn, is equivalent from CQ refresh point of view to deletes only on T1. Thus `dlt(MV)` of Q13can be simplified to:

```
Q23 :
dlt(CQ) = dlt(T1) >< pre(T2)... ><  pre(Tn)
```

This expression is very efficient, if RDBMS provides an easy access to pre refresh images of the tables. Otherwise, pre images must be re-computed using post images and change logs on the tables.

Q23 can be further processed by replacing `pre(Ri)` with `pst(Ri)+dlt(Ri)` to get only the post images of the relations which are immediately available. For example, if i=3, then:

```
dlt(CQ) = dlt(T1) >< pre(T2) ><  pre(T3)
        = dlt(T1) >< (pst(T2)
        + dlt(T2) >< (pst(T3) + dlt(T3))
        = dlt(T1) >< pst(T2) >< pst(T3)
        + dlt(T1) >< dlt(T2) >< pst(T3)
        + dlt(T1) >< pst(T2) >< dlt(T3)
        + dlt(T1) >< dlt(T2) >< dlt(T3)
```

The third factor above:

```
  dlt(T1) >< pst(T2) >< dlt(T3) = NULL
```

since `pst(T2)><dlt(T3) = NULL`, i.e., pst(T2) cannot join with dlt(T3) since rows in dlt(T3) would also cause deletion in T2, hence they cannot belong to pst(T2). Thus:

```
dlt(CQ) = dlt(T1) >< pre(T2) >< pre(T3)
        = dlt(T1) >< pst(T2) >< pst(T3)
        + dlt(T1) >< dlt(T2) >< pst(T3)
        + dlt(T1) >< dlt(T2) >< dlt(T3)
```

This is an important result as it allows us to express refresh expressions in terms of deltas and post-images. In general:

```
Q24 :
dlt(CQ) =  dlt(T1) >< pre(T2) ><  ... >< pre(Tn)
        =  dlt(T1) >< pst(T2) ><  ... >< pst(Tn)
        +  dlt(T1) >< dlt(T2) ><  ... >< pst(Tn)
        ...
        +  dlt(T1) >< dlt(T2) ><  ... >< dlt(Tn)
```

Now consider scenario when deletes affect only a subset of tables: DELETES on { Ti, .. TN }, i>1. If 'ON DELETE CASCADE' option is ON, then this delete will cause deletes on T1, T2, Ti-1 as well, and Q23 and Q24 still hold. If 'ON DELETE CASCADE' is OFF, then delete on { T2, .. TN } has no effect on T1 >< T2 >< T3 >< .. >< Tn as the deletes cannot affect T1.

Note that for this case (DELETE ONLY) case we can apply an optimization for transactions which come in FK-PK units. It is similar to the INSERT case above. If on the FK side we delete only rows which are deleted on the PK side, then Q23 can be significantly simplified to:

```
dlt(MV) = dlt(T1) >< dlt(T2) >< ... >< dlt(Tn)
```

The validity of this optimization can be verified in the same way as in the INSERT case.

Mix of DELETES, INSERTS and UPDATES

Assume tables T1, T2, .. TN go through a series of modifications which (logically) can be represented by set of DELETES followed by set of INSERTS. Then, based on the above two paragraphs, `dlt(CQ)` can be represented as two deltas: delete followed by insert:

```
dlD(CQ) = dlD(T1) >< pre(T2) ><.. pre(Tn)
dlI(CQ) = dlI(T1) >< pst(T2) ><.. pst(Tn)
```

If desired, these two deltas can be represented as three deltas – delete, insert, or update. This can be done by using the primary key of CQ. In the simplest case, the primary key consists of all rowids of the tables. Rows from `dlD(CQ)` and `dlI(CQ)` with the same PK are group together to form an update delta.

8. Computation of CQJA

We compute the results of a CQJA using a backing materialized aggregate view (MAV) that will store the aggregates of query defining CQ up to the last refresh. Result of CQJA is obtained by joining the MV to aggregates on the current deltas. Efficient incremental refresh algorithms for MAVs were proposed in literature [GMS93], [GH195] and also implemented in commercial databases [RB-98], [ORA10GC] and hence we skip their explanation here.

We explain CQJA on an example. Consider the CQ *negative_balance_cq* that notifies whenever a customer's account balance goes below $0. For this CQ, we will create the following backing MAV:

```
CREATE MATERIALIZED VIEW mav_banking_transactions
REFRESH FAST ON DEMAND
SELECT
  acct, sum(amt) sum_amt, count(amt) count_amt,
  count(*) count_star
FROM banking_transactions
GROUP BY acct;
```

We include count(amt) and count(*) in the MAV definition since our incremental refresh [Oracle10GC] requires it.

CQJA result can be obtained by outer joining the delta query ("V" in the below example) with the MV, and then refreshing the MV.

```
INSERT INTO dest_table
SELECT acct, CASE mv.sum_amt is null THEN 0
                  ELSE mv.sum_amt END + v.dlt_sum
FROM (SELECT
        acct,
        SUM(CASE dml='I' THEN 1 ELSE -1 END*amt)
                                              dlt_sum
    FROM CLOG$_BANKING_TRANSACTIONS
         AS OF SNAPSHOT <refexp_scn>  M
    WHERE M.COMMIT_SCN IN <SCN_RANGE>
    GROUP BY acct) V,
RIGHT OUTER JOIN
    mav_banking_transactions mv
ON v.acct = mv.acct
WHERE CASE mv.sum_amt is null THEN 0
         ELSE mv.sum_amt END + v.dlt_sum < 0;
```

We are also investigating extending the MERGE syntax to deposit the results into multiple destinations – one destination will be the MAV and the other would be the destination table/queue specified in the CQ specification.

9. Computation of CQW

In this section we describe techniques to support refresh of CQW. Since one of our goals is to not affect OLTP negatively, we restrict ourselves to the cases where CQW refresh will be efficient and they are:

1. Tables are append-only and,
2. Window functions are on monotonically increasing column, and
3. Windows extend only to the preceding rows.

With the above restrictions, CQW refresh is based on maintaining a buffer of past rows (remember that tables are insert only) needed for window function computation. These are the preceding rows that will be in the window of the incoming rows. Rows that no longer needed will be deleted from the buffer. This is like MV maintenance and prompted us to use an MV to materialize the rows needed for future refreshes of the CQ. The MV definition depends on the type of the window functions used in CQW and is explained later. Incremental maintenance of this MV will take place along with CQW result computation by the refresh process. The size of this MV and the expressions for refreshing it depend on the type of the window functions used – physical (using ROWS

qualifier) vs. logical windows (using RANGE qualifier), and cumulative vs. moving [Z99].

CQW with physical moving windows

Physical window functions use ROW offsets to specify window size, for example, ROWS BETWEEN CURRENT ROWS AND 1 ROW PRECEDING. For simplicity, we explain refresh of CQW on a single table. When there are multiple tables, refresh expressions of Section 6. and 7. are used to get join delta (which is the insert delta as tables are append-only) that forms the input to window functions. Consider the CQ that sends a single notification whenever temperature in a location increases above 90 degrees.

```
CREATE CONTINUOUS QUERY notify_hightemp_cq
  COMPUTE ON COMMIT
  DESTINATION dest_table
SELECT loc, time, temp
FROM (SELECT loc, time, temp,
        LAG(temp,1) OVER (PARTITION BY loc
                          ORDER BY time) prev_temp
        FROM temps)
WHERE temp >= 90 AND prev_temp < 90;
```

This CQ emits a row if, in a given location, the previous temperature was below 90 degrees but current one is above it. Assume that table `temps` has a constraint indicating column `temps.time` is monotonically increasing within `temps.loc`. For this CQW, we define an MV to materialize the last row of each location, i.e., row with a previous temperature. The definition of this MV is:

```
CREATE MATERIALIZED VIEW MV_Q1_window
  PARTITION BY RANGE(pmarker)
    (PARTITION po VALUES LESS THAN (1),
     PARTITION p1 VALUES LESS THAN (MAXVALUE))
SELECT loc, time, temp, 0 pmarker
FROM(   SELECT loc, time, temp,
         ROW_NUMBER () OVER (PARTITION BY loc
                             ORDER BY time) rn,
         COUNT(*) OVER (PARTITION BY loc) pcard
        FROM temps)
WHERE rn > pcard-1 AND rn <= pcard;
```

The predicate "`rn>pcard-1 AND rn<=pcard`" keeps only one row (the last one) for each location. Only this row is relevant to our CQ refresh. We partition the MV into two partitions `p0` and `p1` so as to deposit CQW results and refresh MV_Q1_window together in a single DML statement. The overhead of maintaining this MV thus becomes minimal since we can use fast partition truncate operations rather than expensive DELETE dml to clean up the MV. [N+05] showed clean-up of MV using partition truncate operation is significantly better than using corresponding DELETE dml. At any point in time, only one partition has data and that is the set of rows that potentially be in the window of newly arrived rows. While refreshing the CQW, we determine the new set of rows needed for next refresh and load them into the empty partition. Partition that was used for CQW refresh will be truncated at the end of refresh. The following SQL accomplishes CQW refresh:

```
/* Assume pname and pid are the name and number
   of the partition used for CQW refresh. This
   information can be obtained from RDBMS
   internal functions */
INSERT
  WHEN temp>=90 AND prev_temp<90 AND tag=1
    THEN INTO dest_table
  WHEN rn > pcard - 1 and rn <= pcard
    THEN INTO MV_Q1_window
              (loc, time, temp, 1 - :pid)
SELECT loc, time, temp, tag,
  LAG(temp,1) OVER (PARTITION BY loc
                    ORDER BY time) prev_temp,
  ROW_NUMBER() OVER (PARTITION BY loc
                  ORDER BY time) rn,
  COUNT(*) OVER (PARTITION BY loc) pcard
FROM (  SELECT loc, time, temp, 0 tag
        FROM MV_Q1_window
        UNION ALL
        SELECT loc, time, temp, 1 tag
        FROM clog_temps) V;

ALTER TABLE MV_Q1_window TRUNCATE
        PARTITION :pname;
```

The view V produces the rows of interest – delta rows and materialized rows from MV partition. In the outer query block, we compute the original window function and two additional window functions needed by the multi-table insert to deposit rows in CQ destination and in the empty MV partition.

The above SQL can easily be generalized when there are multiple window functions having same PARTITION BY and ORDER BY keys but with different window sizes.

```
WINDOWFUNC_1(a1) over (
  PARTITION BY pk_1, pk_2, ..., pk_p
  ORDER BY ok_1, ok_2, ..., ok_o
  ROWS BETWEEN m1 PRECEDING AND n1 PRECEDING),
. . .,
WINDOWFUNC_k(ak) over (
  PARTITION BY pk_1, pk_2, ..., pk_p
  ORDER BY ok_1, ok_2, ..., ok_o
  ROWS BETWEEN mk PRECEDING and nk PRECEDING)
```

We compute max(m1, m2, ..., mk) and let it be mMX. The MV needed to handle these window functions will be similar to MV_Q1_window except for the where predicate, which will be:

```
  WHERE rn BETWEEEN pcard - mMX + 1 AND pcard;
```

SQL for incremental maintenance of the MV again will be similar to the one given above except for the WHEN predicate that filters in rows to be populated in the empty MV partition. This predicate will be:

```
  WHEN rn BETWEEN pcard - MMX + 1 AND pcard;
```

CQW with logical moving windows

Logical window functions are those with RANGE window specification and they can be supported in a fashion similar to physical window functions. The (minor) difference is in the expressions used to find rows to be materialized. Due to lack of space we omit the details.

CQ with cumulative window functions

Cumulative window function is a special case of moving window function where the window extends from the first row in the partition up to the current row. Our solution for moving windows is not viable for cumulative windows, as we would buffer the entire table. We solve this by aggregating data within each partition and materializing it in the MV. Consider CQ for notifying whenever client's average account balance falls below $100:

```
CREATE CONTINUOUS QUERY notify_lowbalance_cq
```

```
  DESTINATION dest_table
SELECT acct, time, cavg
FROM (  SELECT acct, time, amt,
          AVG(amt) OVER (
                PARTITION BY acct ORDER BY time
                ROWS BETWEEN UNBOUNDED PRECEDING
                  AND CURRENT ROW) cavg
        FROM banking_transactions)
WHERE cavg < 100;
```

We materialize SUM and COUNT aggregates for each account partition in an MV with the following definition:

```
CREATE MATERIALIZED VIEW MV_window_cumulative
  PARTITION BY RANGE(pmarker)
  (PARTITION p0 VALUES LESS THAN (1),
   PARTITION p1 VALUES LESS THAN (MAXVALUE))
SELECT  acct, SUM(amount) csum,
        COUNT(amount) ccnt, 0 pmarker
FROM banking_transcations
GROUP BY acct;
```

When the CQ is refreshed, we use the following query to compute results and data needed to maintain the associated MV:

```
INSERT
  WHEN v1m = 0 AND cavg < 100
    THEN INTO destination_table(acct, time,
       CASE ccnt=0 THEN 0 ELSE csmu/ccnt END cavg)
  WHEN (v1m = 0 AND  rn = pcard) or
       (v1m is NULL)
    THEN INTO MV_window_cumulative(acct, csum,
        ccnt, 1 - :pid marker)
SELECT v1.acct, v1.time,
  v1.csum +
    CASE v2.marker=0 THEN v2.csum ELSE 0 END csum,
  v1.ccnt +
    CASE v2.marker=0 THEN v2.ccnt ELSE 0 END ccnt,
  v1.marker v1m, v2.marker v2m,
  v2.acct, v2.csum, v2.ccnt,
  pcard, rn
FROM (  SELECT acct, time, amount,
          SUM(amount) OVER (PARTITION BY acct
            ORDER BY time ROWS BETWEEN UNBOUNDED
              PRECEDING AND CURRENT ROW) csum,
          COUNT(amount) OVER (PARTITION BY acct
            ORDER BY time ROWS BETWEEN UNBOUNDED
              PRECEDING AND CURRENT ROW) ccnt,
          COUNT(*) OVER (PARTITION BY acct) pcard,
          ROW_NUMBER() OVER (PARTITION BY acct
            ORDER BY time) rn,
          0 marker
        FROM clog_banking_transactions
      ) v1 FULL OUTER JOIN
      ( SELECT acct, csum, ccnt, pmarker, 0 marker
        FROM MV_window_cumulative
      ) v2 ON
        sys_op_map_nonnull(v1.acct) =
        sys_op_map_nonnull(v2.acct);
```

View v1 computes cumulative sum (`csum`) and cumulative count (`ccnt`) on the delta rows. It also computes the cardinality (`pcard`) of each account partition and sequentially numbers (`rn`) rows within each partition. This view is full outer-joined with the MV on the grouping key (acct). Inner-join and left-outer join rows form CQW results and they are found with the predicate `v1m=0`. To maintain the MV, we need the last row of each partition for the inner and left-outer rows and all the right outer join rows. The predicate (`v1m = 0 and rn = pcard`) or (`v1m is NULL`) finds such rows. This method can be generalized to support multiple logical and cumulative window functions.

10. Performance Study

We conducted experiments on the e-store schema. The `orderline`, `orders` and `users` tables had 10M, 1M and 100K rows respectively, modelling a medium sized e-store application where users typically buy about 10 items in a single order. In most experiments we assumed that user population is fixed, hence only `orderline` and `orders` tables changed. We have used a Linux Intel machine, with a single 400MHz CPU and 2GB of main memory.

FK-PK optimization

Consider CQ Q18 alerting us when an under-age user places an order. Assume only new orders are entered, where a single order, on average, contains ten items. The orderline and orders tables will then have insert delta only. Figure 1 and 2 plot computation time of CQ as a function of the delta size on the orders table. In the experiment delta changed from 100 to 200 orders (0.01% to 0.02% of the table. We show two plots: one for computation using general expressions (see Q13) and one for FK-PK optimized expressions (see Q16). In the first experiment, depicted in Figure 1, none of the tables had indexes and joins in refresh expressions were hash joins. In this case FK-PK optimized expressions performed an order of magnitude better than the unoptimized ones. The result is intuitive since Q16 performs only dlt(orderline) >< pst(orders), while Q13 in addition performs pst(orderline) >< dlt(orders), and |orderline| = 10* |orders|. Figure 2 shows the same experiment when indexes were placed on orderline.oid and orders.oid and nested loop join with index access was used for CQ computation. In this case, on average, Q16 performs seven times better than Q13. We note that presence of indexes visibly slowed down the application and in some OLTP applications, users may elect not to create them. Instead, they may elect less frequent CQ refresh. For less frequent refresh, table deltas may be sufficiently large so that the optimal refresh plan would choose full table rather than index access path.
We also conducted experiments (not shown here) where we had not only transactions with new orders but also ones with updates to the previous orders (e.g, changing the item in the order, discount on it, etc). In this case, in addition to the insert delta we also have the update deltas on base tables. In this case, the general refresh expression based on Q13 performs significantly more work than for the insert only case. Optimized refresh expressions based on Q16 and Q24 also perform more work, but proportionally much less than that of Q13. For the insert and update cases, the optimized refresh expressions performed on average fifteen times better than unoptimized ones.

FK-PK with transactions arriving in FK_PK units

Consider the e-store application where all (or a significant portion of) transactions come in FK-PK units. In this case an optimized Q22 instead of Q16 can be used. Note that Q22 accounts for scenario where transactions do not arrive in FK-PK units but incurs an extra overhead in this case. In the e-store application, this scenario corresponds to a user logging on multiple times and adding new items to an already placed order. Expression Q22 is

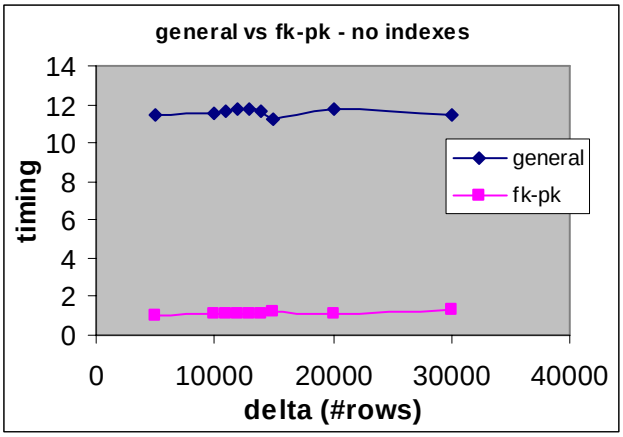

Figure 1.

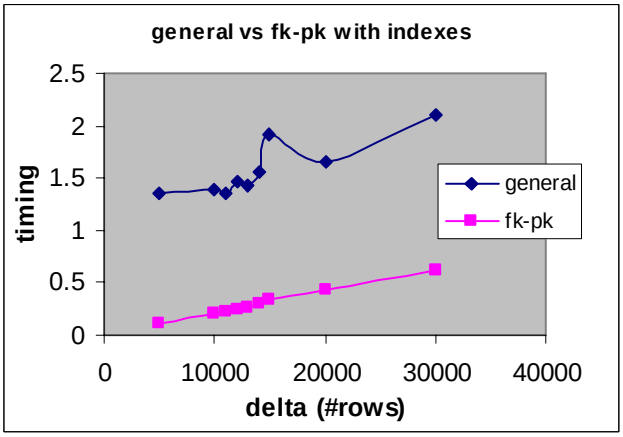

Figure 2.

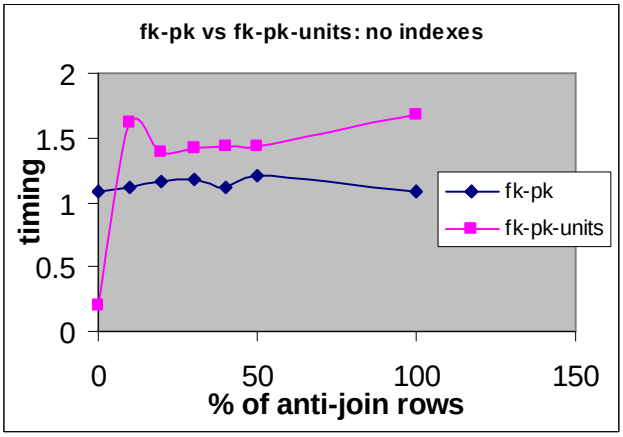

Figure 3.

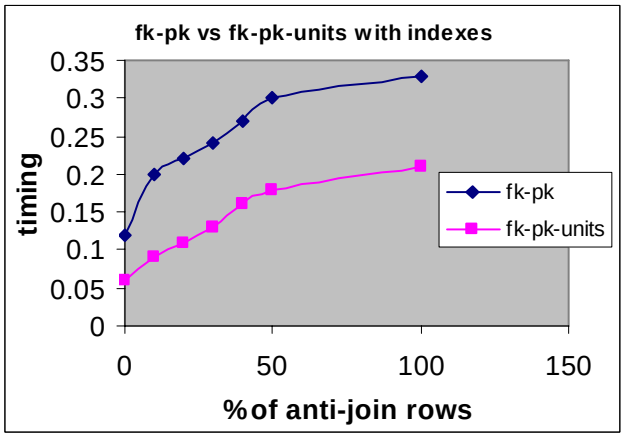

Figure 4.

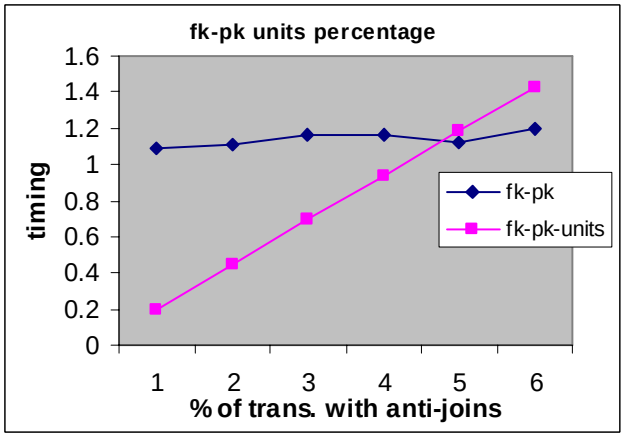

Figure 5.

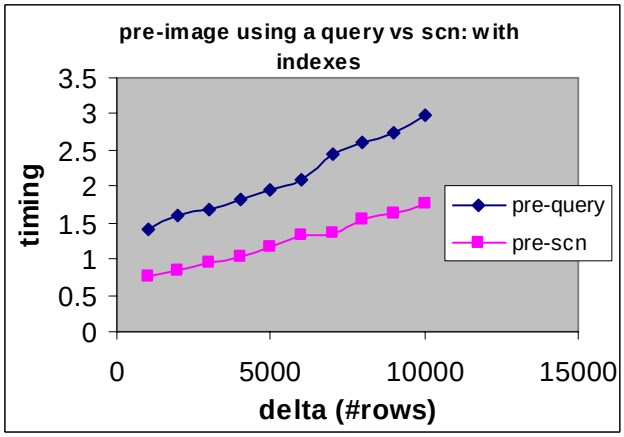

Figure 6.

more expensive than Q16 if there is a portion of the dlt(orderline) that doesn't join with dlt(orders), i.e, when "dlt(orderline) anti-jon><dlt(orders)" is not empty. Figure 3 studies this effect. We vary the portion of dlt(orderline) that doesn't join with dlt(orders), i.e., the size of |dlt(orderline) anti-join><dlt(orders)| from 0% to 100% of |dlt(orderline)><dlt(orders)|. The frequency of CQ computation is as in Figure 1 and the size of delta changes from 100 to 200 orders. Figure 3 also shows the timing for FK-PK optimized expression Q16 where there are no indexes on the base tables. It can be seen that when transactions arrive in FK-PK units (i.e., |dlt(orderline) anti-join><dlt(orders)| = 0), Q22 is about 5 times faster than Q16. If |dlt(orderline) anti-join><dlt(orders)| >0, then Q22 is 20% slower than Q16 and more importantly, that the difference is independent of the size of |dlt(orderline) anti-jon><dlt(orders)| . Figure 4, show the same scenario with indexes on the base tables. In this case Q16 is 1.5 to 2 times faster then Q22 for a wide range of the size of anti-join. This suggest that Q22 is beneficial in scenarios where most (if not all) transactions

arrive in FK-PK units. This is illustrated in Figure 5 in case where base table have no indexes. We vary the precentage of transactions that do not come in FK-PK units from 0 to 100%. For these transactions, If |dlt(orderline) anti-jon><dlt(orders)| = 0.5 * |dlt(orderline)><dlt(orders)|, i.e., 50% or the dlt(orderlines) refer to existing orders rather than to dlt(orders). It can be seen that if no more than 80% of transactions have non empty "dlt(orderline) anti-jon><dlt(orders)", it is beneficial to use Q22 over Q16.

Computing pre-images with query vs undo application

General experssion for CQJ, Q13, requires pre-images of the tables. They can be computed from their post-images and delta logs. For example, in the e-store experiment where we deal only with insert deltas, the pre-image of orders table is calculated using anti-join: pst(orders) anti-><dlt(orders):

```
SELECT * FROM orders WHERE rowid
   NOT IN (SELECT rowid FROM dlt(orders))
```

Alternatively in Oracle RDBMS, pre-images can be obtained directly from the storage layer requesting a version of a table as of previous refresh SCN. In this experiment we compare pre-image calculation using an anti-join query vs using flashback. The scenario is simlar to the one from Figure 1. We change size of the delta on orders from 100 to 1000 with increments of 100 and measure timings of Q13. Figure 6 shows a scenario where indexes on orderline.oid and orders.oid are present and Q13 execution uses index access with nested loop joins and anti-joins for query based pre-image. Figure 7 shows a scenario with no indexes and all hash joins. Figure 6, as expected, shows flashback wins over query based pre-image. In Figure 7, however, performance numbers for flashback and query based pre-images are similar and this is counter-intuitive. Observe that in Figure 6 we access only few data blocks due to nested loops and index access while in Figure 7 we access all blocks of the tables due to hash joins. In the latter case, the overhead of verifying whether a data block needs to be subjected to rollback is as comparable to a small anti-join. This suggests that for refresh expressions scanning significant amount of data, it is safe to use query based pre-images. This is important for non-frequent refreshes as flashback may not be able to provide enough undo for rollback based on the amount of past changes. In that case, flashback based refresh would fail.

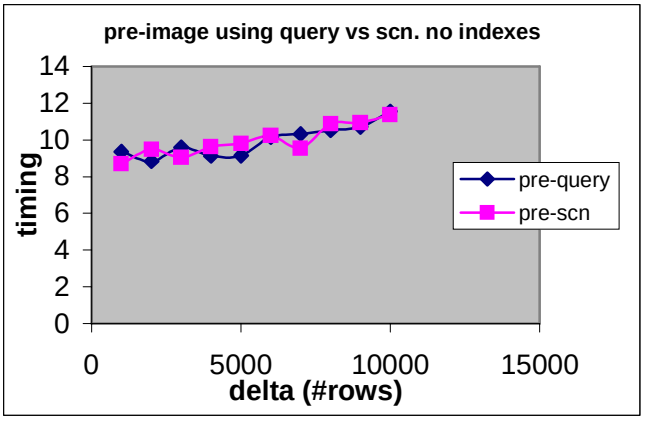

Figure 7.

Frequency of refresh

In this experiment we study the performance impact of the frequency of refresh. As before, we experiment with CQ Q18 detecting orders placed by under-age customers. We use FK-PK optimized refresh expressions of Q16. Each transaction consists of one order with, on average, 10 order items. This experiment processes 7000 such transactions and we vary the refresh frequency from every 10 transactions to every 2000 transactions, and measure the total refresh time. Figure 8 illustrates that the frequency of refreshes has significant impact on the total time to refresh the result of 2000 transactions. Note that if refresh is invoked every 10 transactions, the total refresh time is about 35 times longer than if refresh is called only once. This is due to the overhead of starting up a refresh. That start-up time involves setup of internal execution structures and retrieval of the query execution plan from the cache. If the refresh frequency is above a certain threshold, in our case about every 100 transactions, the total refresh time doesn't change significantly. This validates our initial intuition of providing a batch mode of transactional refresh: TRANSACTIONAL DELTA clause (see Q7) where refresh time can be regulated by the user. In our case, if minimizing the delay of notification is important, users should set the refresh frequency just above that threshold. We currently investigate how to set the frequency of refresh automatically and close to the threshold given user's frequency range.

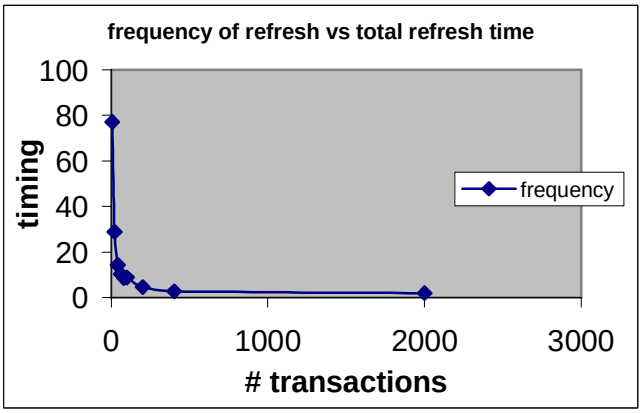

Figure 8

11. Conclusion and Future Work

The paper presented Continuous Query processing engine in Oracle. It is a first commercial attempt to provide a stream and event-processing engine within a relational database. We found that the concepts of traditional streaming systems with queries over append only objects is not sufficient in the database where modifications include update, delete, partition maintenance operations etc in addition to inserts. We based CQ semantics on the concept of query delta since most applications define a state they monitor using a query and want to observe its changes (delete, insert and update deltas) based on changes to the underlying tables.

We based our CQ computation on materialized view refresh and provided significant extensions to the MV refresh algorithms. The extensions, together with Oracle concurrency control mechanism, allowed us to build efficient, non-blocking, no-locking and asynchronous algorithms for CQ refresh.

There are several open challenges for our CQ architecture. First, is to include multi-query optimization in CQ refresh. There has been a lot of work in multi-query optimization for MV advisors and rewrite. We are in the process of incorporating that work in query plans for CQ. An interesting issue is how to deal with CQs that have different refresh schedules. Second, is to extend incremental refresh algorithms to more query shapes. Third, is to include in SQL a functionality to find patterns in sequences of rows. Some work in this area has been proposed in [SZZA+04]. This would significantly increase the applicability of CQ if we could incrementally discover patterns in sequences of updates. We are currently working with other commercial vendors on such extensions. Fourth, is to provide a SQL language construct that allows us to compose CQs. Right now our CQs are stand-alone objects whose results cannot be easily composed using SQL. A possibility is to apply FROM clause windows on CQs [ABB+03].

12. References

[ABB+03] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, J. Widom. "STREAM: The Stanford Stream Data Manager", *Proceedings of SIGMOD, 2003*.

[AS89] A. Swamy, "Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques,", *ACM SIGMOD, 1989*.

[RB+98] R. Bello, et all. "Materialized Views In Oracle", *Proceedings of VLDB, 1998*.

[BLT86] J. Blakeley, P. Larson, F. Tompa. "Efficiently Updating Materialized Views", *Proceedings of SIGMOD, 1986*.

[CC+02] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring Streams: A New Class of Data Management Applications", *Proceedings of VLDB, 2002*.

[CC+03] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, M. A. Shah, "TelegraphCQ: Continuous Dataflow Processing", *Proceedings of SIGMOD, 2003*.

[CD+00] J. Chen, D. J. DeWitt, F. Tian, Y. Wang, "NiagaraCQ: A Scalable Continuous Query System for Internet Databases", *Proceedings of SIGMOD, 2000*.

[CJSS03] C. Cranor, T. Johnson, O. Spataschek, V. Shkapenyuk, "Gigascope: A Stream Database for Network applications", *Proceedings of SIGMOD, 2003*.

[CS94] S. Chaudhuri, K. Shim, "Including Group-By in Query Optimization", *Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994*.

[CS96] S. Chaudhuri, K. Shim, "Optimizing Queries with Aggregate Views", *EDBT 1996*.

[G+92] C.A.Galindo-Legaria et al. "How to extend a conventional optimizer to handle one- and two- sided outerjoin", *IEEE ICDE 1992*.

[G+97] C.A. Galindo-Legaria et al., "Outerjoin Simplification and Reordering for Query Optimization", *TODS, 1997*.

[N+05] N. Folkert, et al "Optimizing Refresh Of a Set of Materialized Views". *Proceedings of VLDB, 2005*.

[G+04] A. Gupta, et al "Data Densification in Relational Database Systems". *Proceedings of SIGMOD, 2004*.

[GHQ95] A. Gupta, V. Harinarayan, D. Quass, "Aggregate-Query Processing in Data Warehousing Environments", *Proceedings of VLDB,1995*.

[GMS93] A. Gupta, I. Mumick, V. Subrahmanian, "Maintaining Views Incrementally", *Proceedings of SIGMOD, 1993*.

[JMS95] H. V. Jagadish, I. S. Mumick, A. Silberschatz, "View Maintenance Issues for the Chronicle Data Model", *Proceedings of PODS, 1995*.

[LPT99] L. Liu, C. Pu, W. Tang, "Continual Queries for Internet Scale Event-Driven Information Delivery", *TKDE, 1999*.

[MSHR02] S. Madden, M. A. Shah, J. M. Hellerstein, V. Raman, "Continuously Adaptive Continuous Queries over Streams", *Proceedings of SIGMOD, 2002*.

[ORA10GC] Oracle Database Concepts 10G Release 1. Part Numb er B 10743-01.

[PD99] N. Paton, O. Diaz, "Active Database Systems", ACM Computing Surveys, 31(1):63-103, Mar 1999.

[PHH92] H. Pirahesh, J.M. Hellerstein, W. Hasan. "Extensible/Rule Based Query Rewrite Optimizations in Starburst", *Proceedings of ACM SIGMOD 1992*.

[PSCP02] T. Palpanas, R. Sidle, R. Cochrane, H. Pirahesh, "Incremental Maintenance for Non-Distributive Aggregate Functions", *Proceedings of VLDB 2002*.

[SLR94] P. Seshadri, M. Livny, R. Ramakrishnan. "Sequence Query Processing", *Proceedings of SIGMOD, 1994*.

[SZZA+04] R. Sadri, C. Zaniolo, A. Zarkesh, J. Adibi. "Expressing and Optimizing Sequence Queries in Database Systems" *ACM Transactions on Database Systems, June 2004*

[Sul96] M. Sullivan, "Tribeca: A Stream Database Manager for Network Traffic Analysis", *Proceedings of VLDB, 1996*.

[Z99] "Rank, Moving and reporting functions for OLAP," *99/01/22 proposal for ANSI-NCITS.*