Continuous Queries over Append-Only Databases

Douglas Terry, David Goldberg, David Nichols, and Brian Oki

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Rd.
Palo Alto, CA 94304

Abstract: In a database to which data is continually added, users may wish to issue a permanent query and be notified whenever data matches the query. If such *continuous queries* examine only single records, this can be implemented by examining each record as it arrives. This is very efficient because only the incoming record needs to be scanned. This simple approach does not work for queries involving joins or time. The Tapestry system allows users to issue such queries over a database of mail and bulletin board messages. The user issues a static query, such as "show me all messages that have been replied to by Jones," as though the database were fixed and unchanging. Tapestry converts the query into an incremental query that efficiently finds new matches to the original query as new messages are added to the database. This paper describes the techniques used in Tapestry, which do not depend on triggers and thus be implemented on any commercial database that supports SQL. Although Tapestry is designed for filtering mail and news messages, its techniques are applicable to any append-only database.

1.0 INTRODUCTION

A new class of queries, *continuous queries*, are similar to conventional database queries, except that they are issued once and henceforth run "continually" over the database. As additions to the database result in new query matches, the new results are returned to the user or application that issued the query. This paper concentrates on the semantics and implementation of continuous queries.

Continuous queries were developed and incorporated into the Tapestry system for filtering streams of electronic documents, such as mail messages or news articles. The Tapestry system maintains information about a document, such as its author, date, keywords, and title, in a database. The database is append-only, that is, new documents are added to the database as they arrive and are never removed. Continuous queries are used to identify documents of interest to particular users. Although the concept of continuous queries was developed for Tapestry, it applies to any database that is append-only.

Tapestry users desire more elaborate filtering queries than those that use only the properties of the individual message, such as selecting all messages that were written by a given person or that contain a given keyword [16]. Tapestry's filter queries can also select a message based on its relationship to other messages, such as the fact that it is a reply to a particular message or that one or more messages are replies to it. They can select a message based on its age, such as the fact that it is two weeks old and nobody has replied to it. They can select a message based on annotations attached to the message by one or more users; for example, users might vote for messages they like and have such votes registered as annotations on the messages. To support these desires, the system cannot simply examine each message as it arrives, but needs to run arbitrary database queries continuously.

Writing a continuous query should be as easy as writing a conventional query for a relational database. In the Tapestry system, users write continuous queries in a special language TQL (Tapestry Query Language) that is similar to SQL. These queries are written as queries over a static database. This permits a user to try out a query by running it as an *ad hoc* query against the database, refine it, and then try it again. Once satisfied with the query, the user can install it in the Tapestry system as a continuous query.

For its storage, the Tapestry system uses a commercial relational database management system that supports SQL. A straightforward method of implementing a continuous query over such a database is to periodically execute the query, say once every hour. Figure 1 outlines the basic algorithm.

Figure 1 Periodic Query Execution

```
FOREVER DO
  Execute Query Q
  Return results to user
  Sleep for some period of time.
ENDLOOP
```

While simple to implement, this approach has three main deficiencies:

- *Nondeterministic results.* The records selected by a query depend on when that query is executed. A query that is executed every hour on the hour may produce a different composite set of results than the same query executed once per day or even every hour on the half hour. This means that two users with the exact same continuous query could be presented with a different set of results.
- *Duplicates.* Each time the query is executed the user will see all records selected by the query, old as well as new. Since the database is append-only, the set of records returned by a query will increase steadily over time. In practice, users are only interested in the records matching a continuous query that have not been previously returned.

1992 ACM SIGMOD - 6/92/CA, USA
© 1992 ACM 0-89791-522-4/92/0005/0321...$1.50

- *Inefficiency.* Executing the same query over and over again is overly expensive. Just as the size of the query's result set increases over time, so does the execution cost. Ideally, the cost of executing a continuous query should be a function of the amount of new data, and not dependent on the size of the whole database.

The problem of duplicates can be solved by having the system remember the complete set of records that has been returned to each user. The system would then take care to only return query results that are not in this set. While this approach strictly avoids duplicates, it still has efficiency problems. Much of the computation cost of the query is spent selecting records that are subsequently discarded.

Active databases, such as the Alert system [12], address the inefficiency problem by using triggers to execute queries over new data as it arrives. Continuous queries are similar to the active queries of the Alert system but can be implemented in standard SQL [2]. Tapestry transforms each user-provided query into an *incremental query* that is run periodically. These incremental queries execute efficiently and avoid duplicates, to a large extent, by limiting the query to the portion of the database that might newly match the query. This is similar to the approach taken by active databases but does not require a trigger mechanism. The result of running a sequence of incremental queries is the same as executing the original user query after every update to the database, but the computation cost is drastically reduced.

The following section explains in more detail why periodic execution can yield nondeterministic results and proposes a clean, time-independent semantics for continuous queries. Sections 4.0 and 5.0 detail the translation steps needed to convert a general SQL query into its associated incremental query. Section 6.0 discusses the implementation of incremental queries and their performance when run on a sample database. Section 7.0 discusses other approaches to supporting continuous queries and related work. Section 8.0 suggests applications of continuous queries and future work.

2.0 CONTINUOUS SEMANTICS

The most significant problem with simply executing a query periodically is that this can produce nondeterministic results. Consider the query: "select messages to which nobody has sent a reply." When a message is added to the database, it matches the query. However, once a reply message arrives, the message being replied to no longer matches the query. If a particular message were to arrive in the database at 8:15 and a reply to it arrived at 8:45, then the message would not be returned by a system that ran the algorithm in Figure 1 every hour on the hour, but would be returned by a system that ran it every hour on the half hour (since the message would match at 8:30).

This raises the general question: What are reasonable semantics for a query that executes "continuously?" In other words: What guarantees can be provided to users about the set of records returned by a continuous query?

Users should not need to understand the implementation of the system in order to know what results to expect as the result of a continuous query. The semantics should be independent of how the system operates internally and when it chooses to perform various operations such as executing queries. Two users with the same continuous query should see the same result data. This implies that the semantics of continuous queries should be time-independent.

We suggest that the semantics of a continuous query should be defined as follows:

> *Continuous semantics*: the results of a continuous query is the set of data that would be returned if the query were executed at *every* instant in time.

This says that the behavior of a continuous query is that it appears to be executed continuously by the system. That is, the system guarantees to show the user any record that would be selected by the query at any time. The system may implement this behavior in any number of ways, such as collecting results and presenting them to the user periodically, but the actual set of results eventually seen by the user is well-defined and time-independent.

To be precise, let $Q(t)$ be the set of records returned by the execution of query Q over the database that existed at time t. That is, $Q(t)$ is the result of running Q at time t. Now let $Q_M(t)$ denote the total set of data returned up until time t by executing query Q as a continuous query:

$$Q_M(t) = \bigcup_{s \le t} Q(s) \qquad \textbf{(EQ 1)}$$

When a query Q is executed with continuous semantics, it returns $Q_M(t)$, not $Q(t)$.

Continuous queries are qualitatively different from one-time queries. Consider the user who wants to see all the messages that do not receive replies. The obvious formulation: "select messages to which nobody has sent a reply," when executed as a continuous query, would return every message to the user, since every message has no replies when it first arrives. This is undoubtedly not what the user intended. The problem does not lie with continuous semantics, but rather with the user's imprecise specification of his continuous query. Finding the messages that *never* receive a reply would require waiting forever, but a short wait will find most messages that never receive a reply. Thus a more precise query would be something like: "select messages that are more than two weeks old and to which nobody has sent a reply." This illustrates the point that not all database queries are suitable as continuous queries. Nevertheless, continuous queries are a valuable concept. Throughout the remainder of this paper, continuous semantics are assumed to be the desired semantics for continuous queries.

One very important question remains: Can continuous semantics be realized in a practical system? Certainly, running a query at every time is not possible, and if it were possible, would not be practical. This paper discusses techniques for providing continuous semantics in an effective and efficient manner.

3.0 PROVIDING CONTINUOUS SEMANTICS

The key to providing efficient continuous queries is the following observation: If we have a query Q_M that can compute $Q_M(t)$ as defined above, then the simple technique of periodically executing Q_M and returning the new results yields continuous semantics. The frequency with which Q_M is executed simply affects the size of each batch of results, not the collective set of results. Figure 2 shows a modification to the algorithm in Figure 1 that obeys continuous semantics. The algorithm keeps track of the last time it ran, τ.

This algorithm works because Q_M is *monotone*, that is, $Q_M(t_1) \subseteq Q_M(t_2)$ whenever $t_1 < t_2$. Many interesting queries are not monotone and are converted to Q_M. We call Q_M the *minimum bounding monotone query* since it is the smallest monotone query that returns all the messages in Q.

Figure 2 Continuous Query Execution using Q_M.

```
Set τ = −∞
FOREVER DO
  set t := current time
  Execute queries Q_M(t) and Q_M(τ)
  Return Q_M(t) − Q_M(τ) to user
  set τ := t
  Sleep for some period of time
ENDLOOP
```

Tapestry's approach to implementing continuous queries is two-fold. First, a query, Q, is converted into the minimum bounding monotone query, Q_M. If the user's query is already monotone then Q_M is usually the same as Q, and in any event produces the same results. Section 4.0 gives the details of how to generate an efficient Q_M.

Second, the monotone query is converted into an *incremental query*, Q^I, that can quickly compute an approximation to $Q_M(t) - Q_M(\tau)$. The queries Q, Q_M, and Q^I can all be in expressed in SQL.

Incremental queries are introduced for performance reasons. An incremental query, Q^I, is parameterized by two times: the time that it was last executed τ, and the current time t. $Q^I(\tau, t)$ is intended to return the records that begin matching query Q_M in the time interval from τ to t. The incremental query works by restricting the portion of the database over which it runs to those objects that might match the query and have not been previously returned. This allows incremental queries to run much more efficiently than queries over the complete database.

An incremental query should obey the following two properties:

It returns enough: $Q_M(t) - Q_M(\tau) \subseteq Q^I(\tau, t)$. (a)
It doesn't return too much: $Q^I(\tau, t) \subseteq Q_M(t)$. (b)

Ideally, Q^I should return exactly the new results, but the current rewrite rules do not achieve this. Unlike Q_M, which is exactly the minimum bounding monotone query, Q^I is only an approximation of $Q_M(t) - Q_M(\tau)$. Q^I returns at least the new results, and occasionally returns a past result. Thus, to guarantee that previously returned results are not returned to a user again, the system must keep track of these results and explicitly filter out duplicates. In practice, if users are not bothered by occasional duplicates, then the results of the incremental queries can be returned directly to users.

As long as the minimum bounding monotone query for Q can be obtained and this query can be incrementalized so as to satisfy the two properties above, then the incremental query can be executed periodically and still guarantee continuous semantics. This is because the union of the results of all the incremental queries is exactly $Q_M(t)$:

$$Q_M(t_n) = Q^I(-\infty, t_1) \cup Q^I(t_1, t_2) \cup \ldots \cup Q^I(t_{n-1}, t_n) \quad \textbf{(EQ 2)}$$

which is true because (using (a))

$$\begin{aligned} Q_M(t_n) &= Q_M(t_1) - Q_M(-\infty) \cup \\ &\quad Q_M(t_2) - Q_M(t_1) \cup \ldots \cup Q_M(t_n) - Q_M(t_{n-1}) \\ &\subseteq Q^I(-\infty, t_1) \cup Q^I(t_1, t_2) \cup \ldots \cup Q^I(t_{n-1}, t_n) \end{aligned} \quad \textbf{(EQ 3)}$$

and (using (b))

$$\begin{aligned} &Q^I_M(-\infty, t_1) \cup Q^I_M(t_1, t_2) \cup \ldots \cup Q^I_M(t_{n-1}, t_n) \\ &\subseteq Q_M(t_1) \cup Q_M(t_2) \cup \ldots \cup Q_M(t_n) \\ &= Q_M(t_n) \end{aligned} \quad \textbf{(EQ 4)}$$

This indicates an effective strategy for executing a continuous query using a conventional relational database manager. The basic algorithm is presented in Figure 3. The system runs each incremental query, queues up the results for delivery to users, records the time at which each query was run, waits some period of time, and then repeats this process using the recorded times as parameters to the incremental queries

Figure 3 Continuous Query Execution

```
Set τ = −∞
FOREVER DO
  set t := current time
  Execute query Q^I(τ, t)
  Return result to user
  set τ := t
  Sleep for some period of time
ENDLOOP
```

As mentioned before, in the Tapestry system users write queries in a special language TQL (Tapestry Query Language). We have developed algorithms for taking a TQL query, transforming it to be monotone, incrementalizing that monotone query, and then converting it to SQL. Rather than introduce TQL, the following sections present versions of the algorithms that translate SQL queries. Because they were designed to work for TQL, the algorithms have some restrictions in the SQL environment. The major difference is that Tapestry queries always want duplicate suppression (DISTINCT) because they are always retrieving mail messages. While the algorithms below do not always suppress duplicates, they often do, and this means they do not support the use of aggregates (such as SUM or COUNT). Another area we have not addressed is outer joins. These are areas for future work.

The following sections examine various constructs that can be used in SQL and discuss how to generate minimal bounding monotone and incremental queries for continuous queries that use these constructs. The rules for producing monotone and incremental queries make two principal assumptions about the database: (1) the database is append-only, namely, records are added to the database but no data is deleted or modified, and (2) each table contains a timestamp column, called "ts", that indicates when the record was added to the database.

4.0 MONOTONE QUERIES

4.1 The Class of Non-monotone queries

Without the database being append-only, no query is monotone since it is always possible to delete a record that has been previously returned as the result of a query, thereby reducing the query's result set. For an append-only database, many common SQL queries are monotone. For example, SQL queries that are simply boolean predicates over the column values of a single table are monotone in nature. Such queries can include the comparison operators (=, <, >, ...) and boolean operators (AND, OR, and NOT). The following is an example of a simple query:

```
SELECT * FROM tbl
WHERE
  tbl.field1 = "Foo" AND NOT tbl.field2 < tbl.field3
```

This query is monotone because once a record is added to the database, it either satisfies the query or not, and that satisfaction doesn't change over time (since the database is append-only).

Queries involving joins are also monotone in nature. Again, this is because the database is append-only. Conceptually, a query with a join is the same as a query over a single table formed by taking the cross product of the joined tables. This single "join" table is append-only as long as the base tables are append-only.

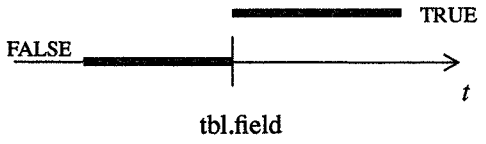
Tapestry queries may include the following constructs, which can lead to non-monotone queries:

- functions that read the current time
- subqueries prefaced by "NOT EXISTS"

First consider time. While the original SQL standard does not include an explicit data type for storing dates, many versions of SQL, such as Sybase's Transact-SQL [15], as well as the proposed new ISO SQL standard [2], do support dates. These systems generally provide functions that read and return the current date and time. In Transact-SQL, for example, GetDate() is such a function, and the ISO SQL standard uses the variable CURRENT_TIMESTAMP. Queries involving calls on such functions are often non-monotone. The simplest example of a query involving time is

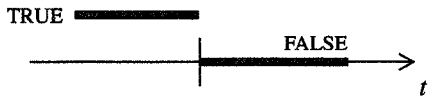
SELECT * FROM tbl WHERE tbl.field *op* GetDate()

When *op* is <, this query can be illustrated as follows:



tbl.field

The horizontal axis represents time t, (the value returned by GetDate()), and the graph represents the boolean value of the query tbl.field < GetDate() for a fixed message. It is false when time t is less than tbl.field (evaluated for that one fixed message), true when t is greater than tbl.field. The fact that this graph is monotone increasing translates directly into the query being monotone. When op is >, the graph is decreasing, and the query is no longer monotone.



tbl.field

An example of a non-monotone query is "select messages that have not expired", or

SELECT * FROM m WHERE m.expires > GetDate()

This is not monotone because any message that satisfies the query will eventually cease to satisfy it (after it expires).

The graph argument just given suggests that a query is monotone if its only reference to time is in subexpressions of the form

$E <$ GetDate()

or

$E \leq$ GetDate()

and likely to be non-monotone if it the comparison operator is >, $\geq$, =, or $\neq$.

Here E is some date-valued expression, possibly involving fields of one or more tables and other built-in functions. The next section will show that queries involving the AND and OR of terms of the first form are indeed monotone. However, boolean combinations of terms of the second form are not always non-monotone. For example,

```
SELECT * FROM tbl
WHERE
  (tbl.field > GetDate() AND tbl.string = "base") OR
  (tbl.field ≤ GetDate() AND
    tbl.string LIKE "%base")
```

is monotone, because it can be rewritten as

```
SELECT * FROM tbl
WHERE tbl.string = "base" OR
  (tbl.field ≤ GetDate() AND
    tbl.string LIKE "%base")
```

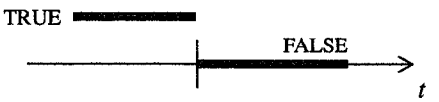
assuming that the two calls to GetDate() in the original query return the same value.

This example illustrates that a monotone rewriting rule is not the same as a test for monotonicity. Although the rewriting rules of the next section will rewrite the first form of the query into the second, it requires knowledge about the semantics of LIKE to conclude that the two queries are the same, and thus that the original query was monotone.

Only the failure of monotonicity due to time has been discussed so far. A second cause of nonmonotonicity is the use of NOT EXISTS. The simple query "select messages that have no reply", might be written in SQL as

```
SELECT * FROM msgs m
WHERE NOT EXISTS (
  SELECT * FROM msgs m1
  WHERE m1.inreplyto = m.msgid)
```

This is non-monotone because a message may satisfy the query for a while, but then fail because of the arrival of a reply. No explicit occurrence of time in the query is involved. Assuming that each append-only table has a column named "ts" that contains a timestamp of when the row was added to the table, the following figure illustrates the non-monotonicity.



m1.ts

A more realistic non-monotone query is "select messages that are more than two weeks old and to which nobody has sent a reply." Although it involves time, it uses the monotone construction E < GetDate(), but is still non-monotone because of NOT EXISTS.

4.2 The Basic Rewriting Rules

This section will show how to compute the minimal bounding monotone query for any SQL query in *standard form* (see below). Throughout the next two sections, several shorthands are used in expressing SQL queries. Table 1 lists these shorthands and their SQL equivalents. In particular, the term t used in a query refers to the current time. All instances of t in a query should obtain the same value. To ensure this, t could be a parameter to the query that is set by calling GetDate() exactly once. $Q_M(t)$ occasionally refers to both the query Q_M and the set of records returned by Q_M when evaluated at time t. The meaning should be clear from context.

Standard form queries have the form

SELECT ... FROM tbl1, tbl2, ...
WHERE (E_{11} AND E_{12} AND .. AND E_{1k_1}) OR
(E_{21} AND E_{22} AND ... AND E_{2k_2}) OR
...
(E_{n1} AND E_{n2} AND ... AND E_{nk_n})

where each E_{ij} is either of the form NOT EXISTS(q), or a boolean expression without subqueries. Furthermore if E_{ij} involves time, then it must be of the form t *op* c, where *op* is one of <, =, >, ≤, ≥, ≠, and c is an arithmetic expression that does not involve t. The subqueries q of NOT EXISTS(q) must also be in standard form. The technical report gives examples to show how most common SQL queries can be rewritten to this standard form [17].

Table 1 SQL shorthands.

t	CURRENT_TIMESTAMP *or* GetDate()
c + 2 weeks	c + INTERVAL "14" DAY *or* DateAdd(week, 2, c)
MAX(c_1, c_2, ..., c_k) < t	$c_1 < t$ AND $c_2 < t$ AND ... AND $c_k < t$
t < MIN(d_1, d_2, ..., d_k)	$t < d_1$ AND $t < d_2$ AND ... AND $t < d_k$
$P(x, y, ...)$	Some expression involving x, y, ...
$\underset{1 \le i \le k}{\text{AND}}\ P(x_i)$	$P(x_1)$ AND $P(x_2)$ AND ... AND $P(x_k)$
$\underset{1 \le i \le k}{\text{OR}}\ P(x_i)$	$P(x_1)$ OR $P(x_2)$ OR ... OR $P(x_k)$

The rewrite rules need only consider each of the AND subexpressions, since the minimal bounding monotone query Q_M of a query Q in the form P OR R is P_M OR R_M. Here is a proof:

$$\begin{aligned} Q_M(t) &= \bigcup_{s \le t} Q(s) \\ &= \bigcup_{s \le t} (P(s) \text{ OR } R(s)) \\ &= \bigcup_{s \le t} (P(s) \cup R(s)) \\ &= (\bigcup_{s \le t} P(s) \cup \bigcup_{s \le t} R(s)) \\ &= (P_M(t) \text{ OR } R_M(t)) \end{aligned}$$

(EQ 5)

This section assumes there are no NOT EXISTS terms. Then each AND subexpression has the form (E_1 AND E_2 AND ... AND E_k). If a term E_i doesn't test the current time (the GetDate() function), then its truth value cannot change as time passes. Since the truth values of these terms are unchanging with respect to time, they don't affect the monotonicity of the query. Each of the remaining terms are of the form c *op* t, with *op* a simple relational test.

A device that will simplify the algorithms is to add a term tbl.ts < t for each table tbl (recall that each table has a 'ts' column with the time the row was added). Thus the query

SELECT * FROM tbl WHERE tbl.field = "joe"

becomes

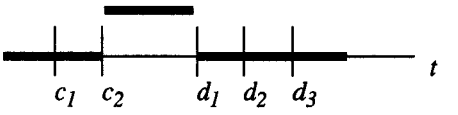
SELECT * FROM tbl
WHERE tbl.field = "joe" AND tbl.ts < t

After the manipulations are over, any remaining tbl.ts < t terms are redundant and can be removed since a record will not appear in the table before time tbl.ts. An example follows shortly.

To avoid multiple cases, first assume that *op* is < or > (the minor changes needed for the other relational operators are indicated at the end of this section). Then each AND subexpression is of the form $c_1 < t$ AND ... AND $c_n < t$ AND $t < d_1$ AND ... AND $t < d_m$ AND P, where P is the conjunction of all the terms that don't involve t. The tbl.ts < t terms mentioned above simply add to the list of c_i's. The expression $c_1 < t$ AND $c_2 < t$ AND ... AND $c_n < t$ is equivalent to MAX(c_1, c_2, ..., c_n) < t, and the expression $t < d_1$ AND $t < d_2$ AND ... AND $t < d_m$ is equivalent to t < MIN(d_1, d_2, ..., d_m). Thus, the AND subexpression can be rewritten as

MAX(c_1, c_2, ..., c_n) < t AND t < MIN(d_1, d_2, ..., d_m) AND P

where P is the conjunction of all the terms that don't involve t. If MAX(c_1, c_2, ..., c_n) < MIN(d_1, d_2, ..., d_m), then the AND subexpression is true between those times, as in the figure below.



If MAX(c_1, c_2, ..., c_n) > MIN(d_1, d_2, ..., d_m), then the AND subexpression can never be true. Combining these cases yields

MAX(c_1, c_2, ..., c_n) < MIN(d_1, d_2, ..., d_m) AND MAX(c_1, c_2, ..., c_n) < t AND P.

Since SQL does not have MAX and MIN functions as such, this must be rewritten as

$$\underset{\substack{1 \le i \le n \\ 1 \le j \le m}}{\textbf{AND}}(c_i < d_j) \text{ AND } \underset{1 \le i \le n}{\textbf{AND}}(c_i < t) \text{ AND } P$$

Here are two examples. First, consider the query "select messages whose date field is in the future." This can be written as

SELECT m.msgid FROM m WHERE t < m.date

After adding the m.ts < t subexpression, c_1 = m.ts and d_1 = m.date, so the monotone query is

SELECT m.msgid FROM m
WHERE m.ts < m.date AND m.ts < t

The redundant m.ts < t can be removed for a final answer of

SELECT * FROM m WHERE m.ts < m.date

Note how the introduction of the m.ts < t term is reflected in the final answer. For a second example, consider the query "select messages that are between 2 and 3 weeks old", which can be written in SQL as

SELECT * FROM m
WHERE m.ts + 2 weeks < t AND t < m.ts + 3 weeks

There is no need to add an m.ts < t subexpression, since it would be redundant. Then c_1 = m.ts + 2 weeks, d_1 = m.ts + 3 weeks, so the monotone query is

SELECT * FROM m
WHERE m.ts + 2 weeks < m.ts + 3 weeks AND
m.ts + 2 weeks < t

Or simplifying,

```
SELECT * FROM m
WHERE m.ts + 2 weeks < t
```

The operators ≤ and ≥ can be dealt with in the same spirit. For example, if there is a subexpression $c' \leq t$, then add c' in with the c_i's, but change $c' < t$ to $c' \leq t$. If there is also a subexpression $t \leq d'$, then add d' in with the d_j's, but in the first AND operation, change $c' < d'$ to $c' \leq d'$. Finally, convert terms of the form $e_1 = t$ into $e_1 \leq t$ AND $t \leq e_1$ and $e_2 \neq t$ into $e_2 < t$ OR $e_2 > t$.

4.3 Handling NOT EXISTS terms

The previous section required that there were no NOT EXISTS terms in the query. This restriction is now removed by allowing terms of the form

```
NOT EXISTS(
  SELECT * FROM tbl1, tbl2, ..., tbln WHERE R).
```

These new terms can be thought of as "for every" terms, since $\neg\exists x P(x)$ is the same as $\forall x \neg P(x)$.

Complications arise if the subexpression R involves calls to GetDate(). Since calling GetDate() returns the current value of time t, R is written as $R(t)$ to indicate that it depends on the current time. First, assume that the function $R(t)$ is already monotone (this includes the case when R does not involve t, that is, does not contain any GetDate() calls).

The NOT EXISTS subexpression shown above is true for the original query $Q(t)$ if no records exist at time t that satisfy the $R(t)$. As before, the rewrite rules need to add tbl.ts < t subexpressions to encode the fact that messages are not in the database until time t = tbl.ts:

```
NOT EXISTS (
  SELECT * FROM tbl1, tbl2, ..., tblk
  WHERE R(t) AND
    tbl1.ts < t AND tbl2.ts < t AND  ... AND tblk.ts < t).
```

This is equivalent to

```
NOT EXISTS (
  SELECT * FROM tbl1, tbl2, ..., tblk
  WHERE
    R(t)  AND MAX(tbl1.ts, tbl2.ts, ..., tblk.ts) < t).
```

As explained in the previous section, each disjunct can be handled separately, and each disjunct containing a NOT EXISTS subexpression will be in the form

```
MAX(c1, c2, ..., cn) < t AND
MIN(d1, d2, ..., dm) > t AND P AND
NOT EXISTS (
  SELECT * FROM tbl1, tbl2, ..., tblk
  WHERE
    R(t)  AND MAX(tbl1.ts, tbl2.ts, ..., tblk.ts) < t)
```

Because $R(t)$ is monotone, any row that satisfies it can never cease to satisfy it as time goes on. Therefore, the NOT EXISTS subexpression can never go from FALSE to TRUE. Thus, it suffices to test R at the earliest time that the rest of the disjunct could become true, namely $C = \text{MAX}(c_1, c_2, ..., c_n)$. The subexpression becomes

```
C < t AND MIN(d1, d2, ..., dm) > t AND P AND
NOT EXISTS (
  SELECT * FROM tbl1, tbl2, ..., tblk
  WHERE
    R(C) AND MAX(tbl1.ts, tbl2.ts, ..., tblk.ts) < C)
```

Here is an example. The query "select messages more than 2 weeks old that no one has replied to" might be written in SQL as

```
SELECT * FROM m
WHERE m.ts + 2 weeks < t AND
  NOT EXISTS(
    SELECT * FROM m m1
    WHERE m1.inreplyto = m.msgid)
```

adding the MAX subexpression gives

```
SELECT * FROM m
WHERE m.ts + 2 weeks < t AND
  NOT EXISTS(
    SELECT * FROM m m1
    WHERE m1.inreplyto = m.msgid AND m1.ts < t)
```

C = m.ts + 2 weeks, so the monotone query is

```
SELECT * FROM m
WHERE m.ts + 2 weeks < t AND
  NOT EXISTS(
    SELECT * FROM m m1
    WHERE
      m1.inreplyto = m.msgid AND
      m1.ts < m.ts + 2 weeks)
```

This says "select messages more than 2 weeks old that did not receive a reply in their first two weeks", which is exactly the smallest monotone query returning at least the messages of the original query.

Next, consider the most general case, where $R(t)$ is not necessarily monotone. Unfortunately, R(t) cannot simply be converted to the minimal bounding monotone query and use the rules from above, as this would change the meaning of the query. For example, consider the query "select messages that have not received a reply for the last two weeks." This is

```
SELECT m.msgid FROM m
WHERE NOT EXISTS(
  SELECT * FROM m m1
  WHERE
    m1.inreplyto = m.msgid AND t < m1.ts + 2 weeks).
```

which means "select messages for which there does not exist a reply less than two weeks old." Making the subquery in the NOT EXISTS monotone produces a query that means "select messages for which there does not exist a reply of any age," which is not the same query.

Monotonizing a NOT EXISTS with arbitrary $R(t)$, requires finding a way to quantify over time, turning NOT EXISTS(...t...) into $\exists\, t$ NOT EXISTS(...t....). It is not possible to quantify over t in SQL, but it is possible to quantify over tbl.ts. For more details, see the technical report [17].

5.0 INCREMENTAL REWRITE RULES

This section now explains how to compute $Q^I(\tau, t)$ from $Q_M(t)$. The hard work was done in the previous section, which gave the algorithms for computing Q_M from Q.

Simply setting $Q^I(\tau, t) = Q_M(t)$ AND (NOT $Q_M(\tau)$) would be very inefficient since typically both $Q_M(t)$ and $Q_M(\tau)$ will involve searching the whole database, and will both return large sets of messages that are mostly identical. To get a more efficient form for $Q^I(\tau, t)$ consider a very simple query such as "select messages from joe". For this query $Q^I(\tau, t)$ is obvious: just check the records that arrived in the interval (τ, t), namely

SELECT * FROM m WHERE
m.from = "joe" AND $\tau \le$ m.ts $< t$

The same idea works in general. Note that the rewriting rules of the previous section always result in a Q_M of the form $OR_i(P_i$ AND $c_i < t)$, where c_i is a MAX of constants. In the case that Q_M has NOT EXISTS subexpressions, there may be NOT EXISTS inside the P_i, but the monotone rewrite will make it independent of t. Choosing $Q^I(\tau, t) = OR_i(P_i$ AND $\tau \le c_i < t)$ clearly satisfies $Q^I(\tau, t) \subseteq Q_M(t)$ since Q^I is just Q_M with an extra restriction. To show $Q_M(t) - Q_M(\tau) \subseteq Q^I(\tau, t)$, compute

$$
\begin{aligned}
&Q_M(t) - Q_M(\tau) \\
&= OR_i(P_i \text{ AND } c_i < t) \text{ AND NOT}(OR_j(P_j \text{ AND } c_j < \tau)) \\
&= OR_i(P_i \text{ AND } c_i < t) \text{ AND } AND_j(\text{NOT } P_j \text{ OR } c_j \ge \tau) \\
&= OR_i[(P_i \text{ AND } c_i < t) \text{ AND } AND_j(\text{NOT } P_j \text{ OR } c_j \ge \tau)] \\
&= OR_i[(P_i \text{ AND } \tau \le c_i < t) \text{ AND} \\
&\quad AND_{j \ne i} (\text{NOT } P_i \text{ OR } c \ge \tau)]
\end{aligned}
$$

This shows that $Q_M(t) - Q_M(\tau) \subseteq OR_i(P_i$ AND $\tau \le c_i < t) = Q^I(\tau, t)$. This is not the only choice for $Q^I(\tau, t)$, but it is a simple one that should run efficiently over the database (because of the $\tau \le c_i < t$ guard), and for most queries be reasonably close to the "ideal" of $Q_M(t) - Q_M(\tau)$.

As an example of the above, consider the query "messages that received a reply." This might be written in SQL as

```
SELECT m1.msgid FROM m m1, m m2
WHERE m2.inreplyto = m1.msgid
```

Adding MAX(m1.ts, m2.ts) $< t$ and applying the above gives $Q^I(\tau, t)$ as

```
SELECT m1.msgid FROM m m1, m m2
WHERE m2.inreplyto = m1.msgid AND
  τ ≤ MAX(m1.ts, m2.ts) < t
```

which expands out to

```
SELECT m1.msgid FROM m m1, m m2
WHERE m2.inreplyto = m1.msgid AND
  m1.ts < t AND m2.ts < t AND
  (τ ≤ m1.ts OR τ ≤ m2.ts)
```

6.0 IMPLEMENTATION, EXPERIENCE, AND PERFORMANCE

6.1 The Tapestry service

The techniques for converting a query into a minimal bounding monotone query and then to an incremental query are used in the Tapestry service. The Tapestry service gathers articles from NetNews and indexes them in a Sybase relational database. Tapestry users can install continuous queries to select interesting articles with specific characteristics. Articles matching any of a user's continuous queries are sent to the user via electronic mail.

When a Tapestry user adds a new continuous query, it is first converted into its minimal bounding monotone query if the query is not already monotone. This query is then converted into an incremental query. The incremental query is added to the database as a stored procedure that takes two date parameters, one indicating the last time that this procedure was executed (τ) and one indicating the current time. Installing each incremental query as a stored procedure allows Sybase to parse the query, generate a query plan, and then compile the query so that it runs efficiently. Once a query is installed in this manner, it never needs to be recompiled, unless the user decides to modify the query.

The Tapestry query engine simply executes the algorithm outlined in Figure 3. That is, it wakes up periodically and calls each of the stored incremental queries with the appropriate parameters. Information about when each query was last executed as well as lists of which articles have been selected by which queries is maintained in the database along with the queries and the indexed information extracted from the news articles. The articles themselves are not stored in the database.

The Tapestry service has been in operation for over 12 months and has been used daily by its developers (a group of 5). A second version of the Tapestry service will be released shortly for more general use by researchers at Xerox PARC.

6.2 Performance measurements

Incremental queries were introduced as a cost-effective implementation of continuous queries. A key conjecture has been that the incremental queries generated by the rewrite rules described in section 5.0 can be run efficiently by a commercial relational database manager. The question is: can a reasonably intelligent query optimizer produce suitable query plans for such queries? In order to answer this question, a number of experiments and measurements were conducted.

Figure 4 Test queries.

Q1. *"Select messages from the comp.databases newsgroup"*

```
SELECT msgid FROM msgs
WHERE newsgroup = "comp.databases"
```

Q2. "Select messages from all of the comp.sys newsgroups"

```
SELECT msgid FROM msgs
WHERE newsgroup LIKE "comp.sys.%"
```

Q3. "Select messages that have a reply sent to comp.databases"

```
SELECT m.msgid FROM msgs m, msgs m1
WHERE m1.inreplyto = m.msgid
  AND m1.newsgroup = "comp.databases"
```

Q4. "Select messages more than 4 weeks old that have not been replied to"

```
SELECT m.msgid FROM msgs m
WHERE m.ts < dateadd(week, -4, getdate())
  AND NOT EXISTS (
    SELECT * FROM msgs m1
    WHERE m1.inreplyto = m.msgid)
```

Q5. "Select first messages in conversation chains greater than two in length"

```
SELECT m.msgid FROM msgs m, msgs m1, msgs m2
WHERE m.inreplyto = " "
  AND m1.inreplyto = m.msgid
  AND m2.inreplyto = m1.msgid
```

6.2.1 Methodology

All of the performance experiments discussed in this section consist of running queries over a trimmed-down version of the Tapestry database, containing data extracted from 380,000 news articles (or messages). This information is stored in a table, called "msgs", which has one record per article. The "msgs" table contains the following fields: msgid, from, subject, date, newsgroup, inreplyto, and ts. Most of these fields are character strings whose meanings should be evident. The "inreplyto" field contains the unique identifier (msgid) of the message to which the given message is a

response; this field is blank if the message is not a reply. The "ts" field is the timestamp indicating when the message data was added to the database. The data is stored as a clustered index on the "ts" field. Non-clustered indices are maintained on the msgid, from, date, newsgroup, and inreplyto fields.

Figure 4 lists the set of queries measured. (The queries are written in Sybase's Transact-SQL.) These are representative of typical continuous queries submitted by Tapestry users. To measure the cost of executing a query, each query was executed once to "warm start" the buffer cache, and then executed 10-20 times in succession. Each measurement reported in the next section is the average elapsed time to execute a query. All times are in seconds. For most queries, all of the data fits in the buffer cache, so the measurements do not include disk IOs. We would have preferred to start with an empty buffer cache, but convincing Sybase to flush its cache is not easy.

6.2.2 Results

The first experiment measured the time to execute a non-incrementalized query compared to the time to execute the incremental version of the same query for various values of τ. Values of τ were chosen so that the range from τ to the current time included the complete database, 10% of the database, and 1% of the database. The notation $Q^I(n\%)$ is used to indicate that the incremental version of query Q was run with a time range covering n% of the database. Figure 5 presents the measured execution times.

Figure 5 The cost of incremental queries vs. regular queries (seconds).

Query	Q	$Q^I(100\%)$	$Q^I(10\%)$	$Q^I(1\%)$
Q1	**2.8**	**123.8**	**4.2**	**0.066**
Q2	**138.0**	**139.3**	**5.7**	**0.18**
Q3	**10.6**	**10.2**	**8.0**	**8.5**
Q4	**9405.2**	**13105.4**	**2118.7**	**1373.5**
Q5	**819.0**	**1276.6**	**1210.6**	**1230.8**

Ideally, the cost of executing $Q^I(100\%)$ should be comparable to the cost of executing the original query. In some cases this is true and in some, notably for query #1, it is not true. For the original query #1, the query optimizer uses the index on the newsgroup field. For the incrementalized query #1, the optimizer decides to use the clustered index on the "ts" field instead. This turns out to be a bad choice. Using the clustered index is preferable only when τ is recent enough that the incremental query considers a small percentage of the database. In practice, this is always the case.

The most disturbing measurements in Figure 5 are those showing that the incremental versions of queries #3 and #5 run no faster than the full queries. This is not a fundamental problem with the incremental queries, but merely a limitation in the Sybase query optimizer. A smarter query optimizer that can utilize multiple indices for join queries should yield better results [9].

To understand this, consider incremental query #3 (in which "@tau" is the parameter containing the value of τ):

```
SELECT m.msgid FROM msgs m, msgs m1
WHERE m1.inreplyto = m.msgid
  AND m1.newsgroup = "comp.databases"
  AND (m.ts > @tau OR m1.ts > @tau)
```

Sybase cannot deal effectively with the timestamp restrictions in the query since they are ORed together. Thus, it chooses to use the index on the newsgroup field to get the candidate set of "m1"s and then use the index on the msgid field to get the "m"s. This is the same plan chosen for the original query, so it is not surprising that the two queries have similar execution costs.

The Tapestry system can take advantage of domain-specific knowledge to get a more efficient incremental query. In particular, since reply messages are known to come after messages to which they reply, the incrementalized version of query #3 can be more simply written as:

```
SELECT m.msgid FROM msgs m, msgs m1
WHERE m1.inreplyto = m.msgid
  AND m1.newsgroup = "comp.databases"
  AND m1.ts > @tau
```

Query #5 can be treated similarly. This revised query, when handed to Sybase, does perform much better than the original query for small values of τ. The new execution times are presented in Figure 6. Q3′ and Q5′ are the revised queries.

Figure 6 The cost of incremental queries using a single timestamp comparison.

Query	Q	$Q^I(100\%)$	$Q^I(10\%)$	$Q^I(1\%)$
Q3	**10.6**	**10.2**	**8.0**	**8.5**
Q3′	**10.6**	**619.0**	**9.1**	**0.60**
Q5	**819.0**	**1276.6**	**1210.6**	**1230.8**
Q5′	**819.0**	**1288.7**	**158.3**	**9.6**

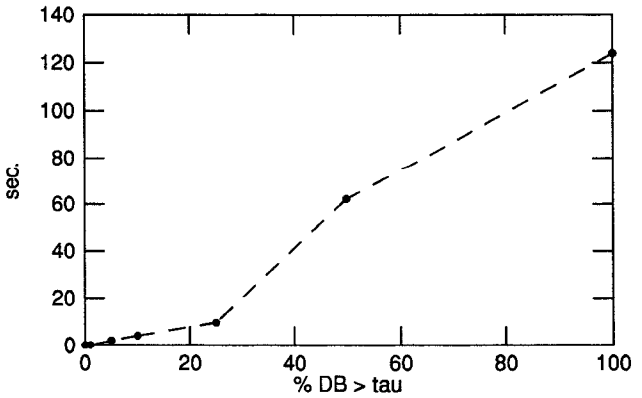
A more general technique for efficiently running the incrementalized version of queries involving joins of two (or more) tables, such as Query #3, is to break them up into two (or more) queries, each of which can be more readily optimized. For example, query #3 becomes:

```
SELECT m.msgid FROM msgs m, msgs m1
WHERE m1.inreplyto = m.msgid
  AND m1.newsgroup = "comp.databases"
  AND m1.ts > @tau AND m1.ts ≥ m.ts

SELECT m.msgid FROM msgs m, msgs m1
WHERE m1.inreplyto = m.msgid
  AND m1.newsgroup = "comp.databases"
  AND m.ts > @tau AND m.ts ≥ m1.ts
```

Running these two queries and summing their execution times yields an improvement over the original cost of $Q^I(1\%)$.

Figure 7 The cost of incremental queries as a function of τ.

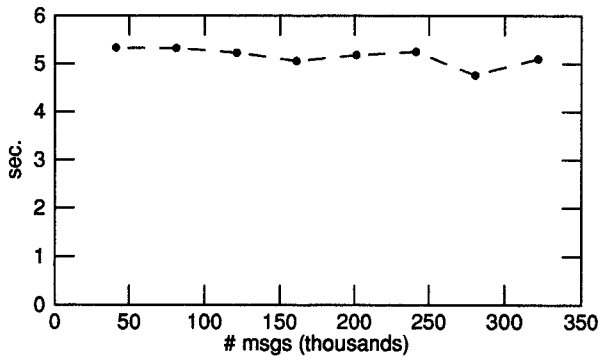


The next set of measurements demonstrates that the cost of running an incremental query is proportional to the number of messages that have been added to the database since the query last ran

and not proportional to the overall size of the database. Figure 7 plots the results of running incrementalized Query #1 for various values of τ. The value of τ was varied so that the proportion of the database considered by the incremental query ranged from the whole database to none of it. As expected. the cost of executing the query decreases as τ increases.

Figure 8 plots the cost of executing incrementalized Query #3 (with a single τ comparison) for various database sizes. In this experiment, the number of messages in the range from τ to the current time was kept constant at about 10% of the maximum database size, or about 38,000 messages. The execution cost remains basically constant as the database size grows.

Figure 8 The cost of incremental queries as a function of database size.



All of the experiments reported thus far were run over a database with simple indices, that is, indices built on the values of single fields. A better indexing scheme is to provide composite indices on indexed fields. These composite indices are built by appending the timestamp value to the field value. They provide a big improvement for incremental queries that include equivalence comparisons and do not hinder other comparisons. For example, replacing the index on the newsgroup field with a composite index on the concatenation of the newsgroup and timestamp fields substantially benefits the incremental versions of both query #1 and #3. This is shown in Figure 9. The composite index relieves the query optimizer from having to choose between using the newsgroup and timestamp index

Figure 9 The cost of incremental queries with a composite index on (newsgroup, ts).

Query	Q	$Q^I(100\%)$	$Q^I(10\%)$	$Q^I(1\%)$
Q1	**2.4**	**2.7**	**0.022**	**0.002**
Q3	**6.1**	**5.3**	**5.3**	**5.3**
Q3'	**5.2**	**5.3**	**0.187**	**0.011**

7.0 RELATED WORK

Recently, much work has been done on *active databases* [8][11][14]. Active databases allow users to specify actions to be executed whenever changes are made to a table. These systems give the actions access to the new data being added or deleted from the database. For any given continuous query, it would be possible to write a trigger that would look at the new data provided and use it to calculate the new query results. This would still not handle time queries.

The Alert system [12] provides *active queries*, which are very similar to continuous queries. Like continuous queries, active queries provide continuing results for queries over append-only databases. However, Alert cannot actively compute queries such as "all message that are two weeks old."

Alert implements active queries using the Starburst database system by checking for new query matches whenever data is added to the append-only tables. The batching inherent in our style of incremental queries should allow more efficiency when many small chunks of data are added to a table. In addition, our techniques require no changes to the underlying database system.

Differential update algorithms for materialized views are related to incremental queries [1]. These algorithms determine the insertions needed to update a view given updates to the base tables. The set of inserted records are essentially incremental results to the query that specifies the view. Tapestry uses similar algorithms, but rather than relying on "delta" relations provided by the database system, it rewrites continuous queries into incremental queries that run over the existing tables of the append-only database.

Some message-processing systems perform filtering of continuous data streams based on per-user profiles [4][6][7][10]. That is, these systems allow users to provide a collection of predicates over the contents of a message, document, or other form of record. When a new message arrives, it is compared against each of a user's predicates, and if any of them "match" the message, then it is returned to the user. These predicates are a form of continuous queries. However, the type of queries supported is quite limited. All of existing filtering systems restrict a predicate to be a simple query over the fields and contents of a single object. These systems do not support continuous queries involving joins or time.

Temporal (or historical) databases [3][5][13] record every change made to a table, allowing a query to ask about the state of the database at some time in the past. A typical implementation technique is to associate with each record a time range indicating over what time interval the record is valid. Queries must be rewritten to ensure consistency; for instance, the records participating in a join must both have been valid at the same time.

The append-only tables with timestamps for new records used in historical databases match nicely the requirements for continuous queries. It should be possible to add our techniques to a historical database. The resulting system would allow continuous queries over tables while relaxing the append-only restriction. This is an area for future work.

8.0 CONCLUSIONS

Continuous queries run "continuously" over a growing database. These queries can be arbitrarily complex, including joins of multiple tables, calls to read the current time, and existential subqueries. With continuous semantics, the results of such a query are well-specified and time-independent. Specifically, a user is guaranteed to see any record that ever matched the query.

A variety of applications ranging from information filtering to data monitoring to active reminders, can make use of continuous queries. For example, continuous queries have proven to be valuable in the Tapestry system, a service that finds interesting messages based on per-user interest profiles. Such profiles are represented as a set of continuous queries. Many existing databases, such as those storing payroll data, billing information, stock market reports, reservations, or reminders, share the append-only nature of the Tapestry database, and hence, are suitable targets for continuous queries.

Continuous queries are no more difficult to write than traditional database queries. In fact, the Tapestry system allows users to write continuous queries and *ad hoc* queries in the same language. This

permits users to experiment with and refine a query before installing it as a continuous query.

Tapestry's implementation of continuous queries has three significant features:

- *Use of conventional relational databases.* Techniques that rewrite a query first into a minimal bounding monotone query and then into an incremental query enable continuous queries to be implemented on top of a commercial database manager. Moreover, performance experiments indicate that incremental queries execute efficiently given appropriate indices for the database. A clustered index on the timestamp field of each record is strongly recommended.
- *Support for time-oriented queries.* To illustrate the difference between current active databases and Tapestry, consider the following example. Suppose a database contains reminders, where each reminder includes a "whenToRemind" field. Then the following continuous query can serve to issue the reminders: "SELECT * FROM reminders WHERE whenToRemind = *t*". A trigger-based system cannot handle this since the reminders should be delivered at certain times instead of in response to database changes.
- *Flexible scheduling.* An incremental query, derived from the minimal bounding monotone query, produces the same overall set of results independent of its frequency of execution. This permits the system to vary this frequency in response to server load or user requirements. For instance, in the Tapestry system, some users want to see messages selected by their queries as soon as possible while others only read such messages once per day or once per week. Tapestry can set the "timeliness" of query execution on a user-by-user basis. This flexibility in scheduling incremental queries is not possible in systems based on triggers.

The results of a continuous query may be returned to a user by several means, including sending them in electronic messages (as is done in Tapestry), writing them to a file, updating a per-user database, or making them available via a database cursor (ala Alert).

This paper presents a consistent semantics for continuous queries and a model for efficient execution of such queries. However, there are some classes of queries for which efficient incremental versions are not known. This includes queries involving aggregates, such as "select messages that have been voted on by more than 5 people" or "select messages that are among the top 10 vote getters." Supporting queries of this sort, as well as relaxing the append-only restriction on databases, is a fruitful area for future work.

9.0 ACKNOWLEDGEMENTS

The design of the Tapestry system, including the development of continuous queries, benefited from discussions with a great many colleagues at Xerox PARC. We are grateful to them and also to Margaret Butler, Vincent Delacour, Dan Greene, Carl Hauser, Matthew Morgenstern, Dan Swinehart, and Marvin Theimer, who provided constructive comments on earlier drafts of this paper.

10.0 REFERENCES

1] J. A. Blakely, P. Larson, and F. W. Tompa. Efficiently updating materialized views. *Proceedings ACM SIGMOD Symposium on the Management of Data*, Washington, D.C., May 1986, pages 61-71.

2] DIS 9075:199x(E). Database Language SQL. ANSI, April 1991.

3] D. Gabbay and P. McBrien. Temporal logic & historical databases. *Proceedings 17th International Conference on Very Large Data Bases (VLDB)*, Barcelona, Spain, 1991, pages 423-430.

4] D. K. Gifford, R. W. Baldwin, S. T. Berlin, and J. M. Lucassen. An architecture for large scale information systems. *Proceedings Tenth Symposium on Operating Systems Principles*, Orcas Island, Washington, December 1985, pages 161-170.

5] T. Y. C. Leung and R. R. Muntz. Query processing for temporal databases. *Proceedings Data Engineering Symposium*, 1990, pages 200-208.

6] E. Lutz, H. v. Kleist-Retzow, and K. Hoernig. MAFIA - An active mail-filter-agent for an intelligent document processing support. *Multi-User Interfaces and Applications*, S. Gibbs and A. A. Verrijn-Stuart (editors), North Holland, 1990, pages 16-32.

7] T. W. Malone, K. R. Grant, F. A. Turbak, S. A. Brobst, and M. D. Cohen. Intelligent information sharing systems. *Communications of the ACM* 30(5):390-402, May 1987.

8] D. R. McCarthy and U. Dayal. The architecture of an active data base management system. *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, 1989, pages 215-224.

9] C. Mohan, D. Haderle, Y. Wang, and J. Cheng. Single table access using multiple indexes: Optimization, execution, and concurrency control techniques. *Proceedings International Conference on Extending Database Technology (EDBT)*, Venice, Italy, March 1990, pages 29-43.

10] S. Pollock. A rule-based message filtering system. *ACM Transactions on Office Information Systems* 6(3):232-254, July 1988.

11] A. Rosenthal, S. Chakravarthy, B. Blaustein, and J. Blakely. Situation monitoring for active databases. *Proceedings 15th International Conference on Very Large Data Bases (VLDB)*, Amsterdam, Holland, 1989, pages 455-467.

12] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An architecture for transforming a passive DBMS into an active DBMS. *Proceedings 17th International Conference on Very Large Data Bases (VLDB)*, Barcelona, Spain, 1991, pages 469-478.

13] R. Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems* 12(2):247-298, June 1987.

14] M. Stonebreaker, A. Jhingran, J. Goh and S. Potamianos. On rules, procedures, caching and views in data base systems. *Proceedings of the ACM-SIGMOD Symposium on the Management of Data*, Atlantic City, May 1990, pages 281-290

15] Sybase. Transact-SQL user's guide. Sybase, Inc., October 1989.

16] D. B. Terry. 7 steps to a better mail system. *Message Handling Systems and Application Layer Communication Protocols*, P. Schicker and E. Stefferud (editors), North Holland, 1991, pages 23-33.

17] D. B. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. Technical report CSL-92-5, Xerox Palo Alto Research Center, May 1992.