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.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
1992 ACM SIGMOD - 6/92/CA, USA
© 1992 ACM 0-89791-522-4/92/0005/0321...$1.50

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

- *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:

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

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.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

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 $Q_M(t)$:

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

which is true because (using (a))

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

and (using (b))

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

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
```


DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

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.

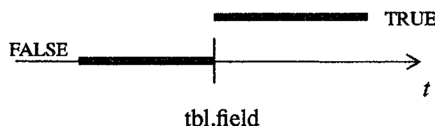
Tapestry queries may include the following constructs, which can lead to non-monotone queries:

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

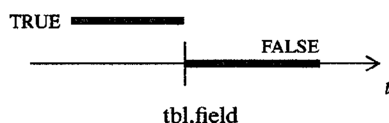
First consider time. While the original SQL standard does not include an explicit data type for storing dates, many versions of SQL, such as Sybase's Transact-SQL [15], as well as the proposed new ISO SQL standard [2], do support dates. These systems generally provide functions that read and return the 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:



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.



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 >, ≥, =, or ≠.

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.

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.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

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 $<, =, >, \leq, \geq, \neq$, 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, ...
$\text{AND}_{1 \leq i \leq k}\ P(x_i)$	$P(x_1)$ AND $P(x_2)$ AND ... AND $P(x_k)$
$\text{OR}_{1 \leq i \leq k}\ P(x_i)$	$P(x_1)$ 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:

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

(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

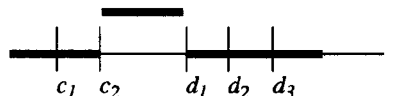
SELECT * FROM tbl
WHERE tbl.field = "joe" AND tbl.ts < t

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

$$\text{AND}_{\substack{1 \leq i \leq n \\ 1 \leq j \leq m}}(c_i < d_j) \text{ AND } \text{AND}_{1 \leq i \leq n}(c_i < t) \text{ AND } P$$

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,

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.