SQLCM: A Continuous Monitoring Framework for Relational Database Engines

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Arnd Christian König
Microsoft Research
chrisko@microsoft.com

Vivek Narasayya
Microsoft Research
viveknar@microsoft.com

Abstract

The ability to monitor a database server is crucial for effective database administration. Today's commercial database systems support two basic mechanisms for monitoring: (a) obtaining a snapshot of counters to capture current state, and (b) logging events in the server to a table/file to capture history. In this paper we show that for a large class of important database administration tasks the above mechanisms are inadequate in functionality or performance. We present an infrastructure called SQLCM that enables continuous monitoring inside the database server and that has the ability to automatically take actions based on monitoring. We describe the implementation of SQLCM in Microsoft SQL Server and show how several common and important monitoring tasks can be easily specified in SQLCM. Our experimental evaluation indicates that SQLCM imposes low overhead on normal server execution end enables monitoring tasks on a production server that would be too expensive using today's monitoring mechanisms.

1. Introduction

The ability to monitor a database server is a crucial aspect of database administration. There are a variety of scenarios where monitoring is essential to accomplish common administrative tasks. For example, monitoring execution time of SQL queries can be useful for detecting under-performing queries. Monitoring locks held and delays due to locking are important for identifying locking hotspots in the data. Similarly, monitoring server resource consumption (CPU, memory, number of connections, etc.) are necessary for tasks such as auditing, capacity planning and detecting security problems.

Database monitoring is the ability to observe the values of *system counters* that describe the system's state, e.g., execution time of a query, locks held, CPU/memory usage in the above examples. Today's database systems typically have the necessary instrumentation to expose such system counters. In this paper, we refer to *automated monitoring* of a DBMS as the ability to evaluate conditions over these counters and to take *actions* on them. Examples of actions are: persisting counters to a table/file, notifying the database administrator (DBA), aborting the execution of a query etc.). Such automated monitoring can reduce total cost of ownership and increase efficiency of a DBA.

In today's commercial database systems, system counters are exposed to clients using two basic mechanisms. The first mechanism allows obtaining a *snapshot* of these counters at any point in time by *polling* the server. The second mechanism is *event recording*, i.e., ability to write counters associated with a system event to a file/table. Events include SQL statement execution begin/end, lock acquire/release, user login/logout, etc. With each event several counters associated with it e.g., time, database id, application that causes the event, or duration of the event may be recorded. While these mechanisms are commonly available in today's systems, they are inadequate for several important monitoring tasks.

Consider the task of detecting "outlier" invocations of a stored procedure *P*, i.e., identifying invocations of *P* that are much slower to execute than other instances. This can be useful since a DBA can later analyze these outliers to determine the reasons for underperforming. Similarly, consider the task of detecting the total delay caused by blocking on a lock resource (grouped by the SQL statements in conflict and ordered by the total amount of time blocked). This task can be helpful in detecting locking problems due to poor application design or unanticipated interactions across applications. For both these tasks, if we use event recording, then a very large volume of monitored data needs to be written out by the server (all stored procedure completion events in the first example, and all blocking/release events in the second example), even though the amount of information the DBA needs to see is considerably smaller. On the other hand, if we use mechanism of repeatedly polling the server, we could compromise the accuracy of answers obtained if we do not poll frequently enough (e.g., miss outliers, underestimate total blocking delay for short, but frequent queries). If instead, we poll very frequently, we can once again incur significant load on the server (e.g., due to repeated traversal of the lock resource dependency graph in the second example).

Thus, the polling based approach has the drawback that if polling is performed infrequently, then the monitoring application can lose valuable information. On the other hand, if polling is very frequent, it can impose significant CPU overheads on the server. Similarly, event recording

(although not lossy), can incur significant overheads on the server since potentially a large number of events needs to be copied out to a file/table or sent over the network. Thus the existing solutions to these monitoring problems are unsatisfactory. We note that in both these examples, if the monitoring could have been done *inside the database server*, i.e., at the source of the monitored data, then both the performance overhead of event recording as well as loss in accuracy of polling could have been avoided.

In this paper we describe an alternative framework, called **SQLCM** (SQL Continuous Monitoring engine) that makes it easy to develop a large class of important monitoring based tasks such as the ones described above. An overview of the SQLCM architecture is shown in Figure 1 (the architecture is explained in more details in Section 2.3). SQLCM addresses the limitations of existing mechanisms discussed above as a consequence of the following design characteristics: (1) SQLCM is implemented entirely *inside* the database server, and (2) the monitored information can be automatically grouped and aggregated, with the grouping columns and aggregation functions being specified by the database administrator (DBA). As we show in this paper, this grouping and aggregation can be done very efficiently. Consequently, the volume of information that needs to be copied out of the server is small, thus dramatically reducing the overheads incurred on the server by the monitoring tasks. (3) Monitoring tasks can be specified to SQLCM in a declarative manner using Event-Condition-Action (ECA) [6] style *rules*. A rule implicitly defines *what* conditions need to be monitored (e.g., an instance of a stored procedure executes 5 times slower than the average instance, a statement blocks others for more than 10 seconds) and what actions need to be taken (e.g., report the instance of the stored procedure to a table, cancel execution of the statement). SQLCM only incurs monitoring overhead that is necessary to implement currently specified rules (i.e., DBA tasks). These rules are written against a predefined *schema* that specifies the valid events, conditions and actions supported by SQLCM.

We have implemented a prototype of SQLCM inside Microsoft SQL Server. We demonstrate how several important DBA tasks can be easily specified in the SQLCM framework. We have conducted experiments to measure the overhead of monitoring and rule evaluation in SQLCM; and we compare this with overheads for solutions using today's monitoring mechanisms. This work was done in the context of the AutoAdmin project [1] at Microsoft Research. The goal of the AutoAdmin project is to develop technology for making database systems more self-tuning and self-managing.

The rest of this paper is structured as follows. We present an overview of SQLCM in Section 2 and justify our design decisions. Section 3 provides several examples of interesting DBA tasks that are enabled by SQLCM (or are much more efficient to perform in SQLCM). Sections 4 and 5 respectively describe the two key components of SQLCM: the monitoring engine and the rule engine. Section 6 describes the implementation of SQLCM in Microsoft SQL Server and presents a careful experimental evaluation of its overheads. We discuss related work in Section 7 and conclude in Section 8.

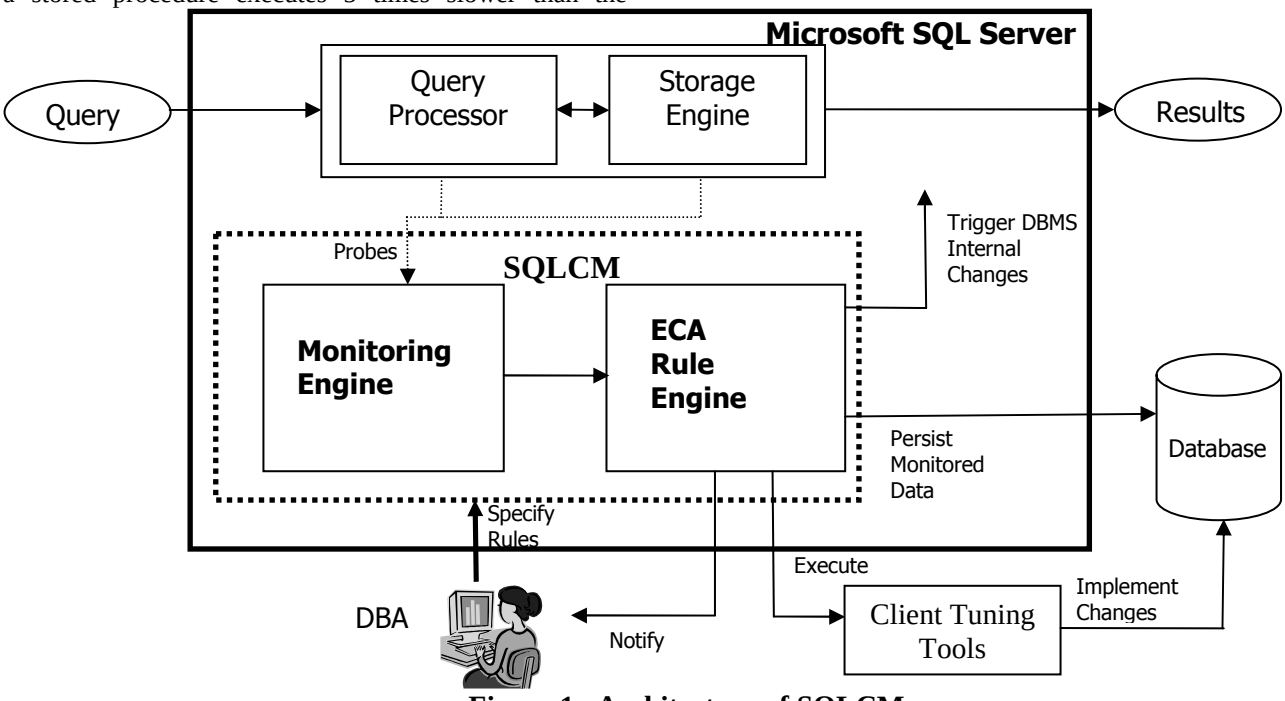


Figure 1. Architecture of SQLCM

2. Architecture of SQLCM

In this section we present the rationale for our design choices in SQLCM, and provide an overview of the key components of SQLCM. There are two major criteria that a framework for database monitoring applications should satisfy. First, it should flexible enough to easily specify a variety of monitoring tasks including the ability to take actions based on monitored data. Second, as highlighted by the two examples in the introduction, the performance overhead imposed on the database server should be small. Moreover, the developer of the monitoring task/application (e.g., a DBA) should be able to control the overheads imposed. This is particularly relevant for tasks where *accuracy* of monitoring is crucial (e.g., auditing system usage or monitoring for security violations).

2.1. Key Design Decisions in SQLCM

We make two key design decisions that determine the basic architecture of SQLCM: (1) We adopt a server-centric solution, i.e., SQLCM is implemented inside the database server. (2) The programming model for specifying monitoring tasks in SQLCM is Event-Condition-Action (ECA) rules. In our current prototype, the monitoring as well as rule evaluation is interleaved with query processing (see Section 6 for details on implementation). Below, we justify these two design decisions in light of the criteria discussed above.

Justification for Server-Centric Solution

There are a number of reasons for a server-centric solution to monitoring. First, several monitoring tasks require ability to observe a large number of events in the server. However, the actions to be taken as a consequence of monitoring in these tasks are often based on a much smaller volume of information (typically *filtered* and/or *aggregated*). For example, finding the top *k* most expensive queries during the day requires the ability to observe the execution time of each query that executes on the server but report only *k* queries. Since SQLCM provides the ability to apply filtering and aggregation over the monitored data *at its source* (i.e., inside the server), the volume of information that needs to be copied from the server to the client is typically small, thereby reducing the performance overhead significantly. Furthermore, this architecture allows the monitoring to be synchronous with the events in the system, resulting in accurate measurements and no missed events (as there would be in a polling-based architecture). Finally, a significant advantage of a server-centric solution (although not the focus of this paper) is that it enables the possibility of taking actions based on monitoring that can allow the server to dynamically adjust its behavior without DBA intervention (e.g. resource governing or automatically correcting database statistics). In Section 3 we provide examples of such tasks. While a server-centric monitoring solution can be advantageous, it also imposes the requirement that the overheads be very low.

Justification for ECA as the Programming Model

We use Event-Condition-Action (ECA) rules [6] as the programming model in SQLCM. ECA rules are simple and have been widely used in many systems including database systems for triggers and for notification services [14]. Moreover, as we demonstrate in this paper through examples (see Section 3), the ECA framework appears to be expressive enough to allow specification of a large class of important monitoring tasks. In automated monitoring applications, conditions on system counters need to be evaluated potentially multiple times per query invocation and should not have noticeable impact on performance, particularly when the system is under heavy load. Thus, (unlike regular SQL) the expressive power of the programming model is of secondary importance, whereas *low* and *controllable* overhead is crucial. For this reason, SQLCM does not use the existing database mechanisms such as triggers or notification mechanisms, but rather implements a lightweight ECA rule engine. Due to their simplicity, ECA rules are amenable to implementation with low CPU and memory overheads (which we validate in Section 6.2). Furthermore, (as described in Section 4) the aggregated monitored data upon which rules are evaluated are held in in-memory data structures, whose memory consumption can be controlled during specification.. Thus evaluating a rule incurs no disk accesses. Finally, we note that the monitoring overhead is itself limited to gathering system counters that are referenced in currently active rules, i.e., no monitoring is performed unless it is required by a rule.

2.2. SQLCM Monitored Objects

In SQLCM, the internal state of the database server is modeled by a number of *monitored classes* (e.g. a *Query*), each of which expose one or more events (e.g. the *Query.Start* event is triggered when a query starts executing) and a number of attributes or system counters (e.g. *Query.Duration* representing the total time a query takes to execute). These attributes are referred to in SQLCM as *probes* since their values are obtained by probing (or extracting from) the database server at runtime. We note that for a particular probe or event to be available for monitoring, appropriate instrumentation needs to be done in the server code (e.g., inserting a timer around query execution code to obtain the *Query.Duration probe*). In general, the set of available

monitored classes, probes and events (that constitute the SQL *schema*) could vary from one DBMS to another. A subset of the schema that we use in our current implementation is shown in *Appendix A*. The current SQLCM implementation exposes 5 monitored classes: *Query*, *Transaction*, *Blocker*, *Blocked* (referring to pairs of blocked queries in a lock conflict) and *Timer* (used for asynchronous rule invocation, see Section 5.1). In general, this schema can be augmented to cover other relevant server objects to monitor (e.g., *Table*).

2.3. Overview of SQLCM components

As shown in the architectural overview (Figure 1) SQLCM is implemented inside the database server. There are two key components of SQLCM: the *monitoring engine* and the Event-Condition-Action (ECA) *rule engine*. The user, typically a DBA, interacts with SQLCM by specifying ECA rules (expressed against the SQLCM schema) for implementing monitoring tasks.

The purpose of the monitoring engine is to collect attributes of *objects* (i.e., instances of monitored classes) that are necessary to implement currently active rules. Objects are grouped, aggregates are computed for each group, and the resulting data is stored in in-memory data structures called *lightweight aggregation tables* (LATs). Grouping and aggregation is performed on attributes of the object type, i.e., on probes that are gathered synchronously during query execution. For example, we may group all queries by the application (or user) that issued them. For many tasks, only aggregate values of an attribute in a group may be necessary, e.g., the average and standard deviation of the *Duration* attribute of all queries in a group. SQLCM supports the ability for common aggregation functions such as COUNT, AVERAGE, etc. to be specified on any attribute.

The ability to specify at which level of granularity (i.e., grouping) to store aggregated monitored information at the server is critical, as different tasks may call for monitoring at the level of connections, applications, users, transactions etc. Any fixed interface is likely to either provide an insufficiently fine level of aggregation or to be too detailed, thereby requiring too much memory inside the server for state. SQLCM addresses this issue through the use of LATs as a form of flexible, in-server grouping and aggregation. The details of the monitoring engine are discussed in Section 4.

The goal of the rule engine is to monitor events and take the necessary actions when a rule fires (i.e., the condition in the rule becomes true). The expressiveness of ECA rules, and other design details of the rule engine are the topic of Section 5. As an illustration, the following simple rule appends to a specified table the probe values for any query that takes more than 100 seconds to execute (when the query commits).

Event: Query.Commit.
Condition: Query.Duration > 100
Action: Query.Persist (*TableName*).

In addition to **Persist** (write to a table), other actions such as **SendMail** (sends an email), **RunExternal** (launches a specified program) can be useful for a variety of tasks. In Section 5.3, we describe the actions supported by our current prototype. Finally, as mentioned earlier, a unique and powerful feature of SQLCM is that by virtue of being inside the server, it allows actions that can adjust server behavior dynamically (without DBA intervention).

3. Applications of SQLCM

In this section we describe several examples of monitoring tasks. These examples highlight how SQLCM as a framework makes it easy to specify these tasks, thereby either reducing the complexity of implementing these tasks as compared to today's database systems or reducing performance overheads, or in many cases both. We have implemented the first three tasks below using our prototype implementation of SQLCM on Microsoft SQL Server.

Example 1: Detecting Outlier Instances of a Stored Procedure: The task of detecting invocations of a stored procedure *P* that are outliers, i.e., much slower to execute than other instances, is an important and common task for DBAs (in this example, we could define "much slower" as any instance that runs 5 times slower than the average instance, or any other appropriate statistical measure that can be expressed using SQLCM aggregate functions). This type of outlier detection is often valuable for DBAs to identify problematic combinations of parameters for the stored procedure. We will describe the implementation of this example application in detail in Sections 4 and 5.

Example 2: Detecting Poor Blocking Behavior: Another common problem faced by DBAs is detecting which update statements are responsible for the largest blocking delays in the system. In other words for each statement, we need to track the total time for which it blocked other statements. Such a task can potentially help in identifying poor design in the SQL application, e.g. a hot spot in the data (or metadata). This task would be specified in the SQLCM framework as a simple ECA rule triggered by any statement *S* releasing a lock resource other statements are waiting on. For each of the blocked statements, the time it has been waiting on the lock resource is then added to the total waiting time for *S* (see Appendix A for *Blocker* and *Blocked* objects in the schema that would be used in this rule). Each statement, along with the total

blocking delay caused by that statement is stored in a lightweight aggregation table (LAT).

Example 3: Identifying Top k most expensive queries: One methodology used by DBAs for identifying performance bottlenecks is to find the few most expensive queries over a period of time. Even for this relatively simple task, the overheads of using today's monitoring solutions can be high. In contrast, this task is not only easy to specify in SQLCM but the overheads are very low (see Section 6.2). This task would be specified in the SQLCM framework using a LAT storing the queries, and an ECA rule that inserts every query after it commits into the LAT. The LAT is specified in such a way that it only stores *k* entries ordered by *Query*.*Duration*, thus maintaining the top k queries by duration at all times (for a detailed description of LAT specification, see Section 4.3).

Example 4: Auditing/Summarizing System Usage: DBAs often require the ability to audit or summarize usage of system resources. This may be necessary for a variety of reasons, e.g., (a) enforcing service level agreements (b) detecting potentially unauthorized access attempts, e.g., number of login failures for each user, (c) summarizing query/update "templates" (see Section 4.2) for a particular application, their associated frequencies and average/max duration for each template etc. over a 24 hour period. Note that this may require the ability to collect summaries synchronously with query execution (in order to compute aggregate values), and in addition have rules that persist these asynchronously (e.g. every 24 hours). The latter type of invocations are handled using a special Timer class, described in Section 5.1.

Example 5: Resource Governing: The ability to limit resource consumption of queries in a flexible way can be very useful in a variety of scenarios: (a) Stopping a runaway query (i.e., a query that has exceeded a certain budget on system resources). (b) Enforcing limit on concurrent query execution for a user (e.g., User *X* cannot have more than *K* queries executing at any point in time). (c) Adjusting the multi-programming level (MPL) dynamically based on the monitored resource consumption.

Finally, we note that since SQLCM allows rules to be added and removed dynamically. Thus, monitoring applications can take advantage of this to allow more flexible and customized monitoring (e.g., turning off/on rules based on time of day, adjusting thresholds in rules to capture more/less information etc.).

4. Monitoring Engine

We now describe the key components of the monitoring engine – the *probes* inside the database server (Section 4.2), including *signatures*, a special type of probe useful for identifying templates of parameterized queries (Section 4.2), and *light-weight aggregation tables* (Section 4.3).

4.1. Probes

The monitoring engine of SQLCM uses probes inside the query processor and storage engine to collect the attributes of the monitored objects probes, which are assembled into monitored objects on demand (i.e., at the time of rule-evaluation). As most of the probes are collected at various points of the server code already, this typically adds negligible overhead to normal query execution (see Section 6.2).

SQLCM offers a generic interface to integrate new monitored objects, events and probes into the schema. This means that probes can be implemented with no knowledge of SQLCM internals. SQLCM offers methods to register monitored objects and probes internally, when they are available to the system (e.g. the corresponding *Blocker* is only available during the period that a query is blocked on a lock resource, a query's *physical plan signature* (see Section 4.2) is available only after query optimization is complete). Internally, probe values are cast to SQL Server types, enabling the use of all aggregation functions provided by the database server for LAT aggregation as well.

4.2. Signatures

Consider a SQL application that executes "templatized" queries (not stored procedures) repeatedly, e.g.., different instances of the same query with different constants in the selection conditions. In this case, it is natural for the DBA to track the performance of the template, rather than each individual query. Consider another example of a SQL stored procedure that is structured as follows: IF *Condition* THEN A ELSE B, where A and B are SQL statements. Some instances of the stored procedure will execute A, while others will execute B. The performance characteristics of the stored procedure could be different in each case. Thus for some tasks, e.g. outlier detection (Example 1 in Section 3), it is meaningful for the DBA to monitor performance of these two different paths separately.

The simplest method of matching the query-text may be sufficient to differentiate different parameter-less stored procedures, but any purely query-text based grouping is undesirable due to its sensitivity to formatting and its inability to group different instances of the same query with different parameters. To support monitoring

applications such as the ones described above, SQLCM exposes the notion of query *signatures*. A signature is a probe value that is exposed as an attribute of the Query object. If two queries have the same signature, they share the same internal structure (depending on the exact signature type used, see below); otherwise their structure differs. Note that since aggregation tables support grouping on all possible combinations of probes, it is possible to group queries on signatures.

Below, we describe the four kinds of signatures exposed in SQLCM (see schema in Appendix A), and examples where each signature can be useful. In Section 6.2, we measure the overhead of signature computation.

(1) *Logical Query Signature* – As described in the example above, it is sometimes necessary to monitor the execution of re-occurring query templates which have a number of implicit parameter values that vary from one instance to another. To facilitate such a grouping, SQLCM uses the internal logical query tree generated during query optimization to compute a linearized representation of the structure of a query and its predicates (the techniques used are similar to the query/view representation of [9]; we omit details here for brevity). In cases where we can differentiate between the different parameters $P_1,\ldots,P_n$, of a query template internally (e.g. such as when the query is executed as part of a stored procedure with n parameters) we replace each occurrence of a parameter P_i with the symbol that matches only other occurrences of P_i. If we are unable to identify parameters (e.g. for ad-hoc queries) we substitute a wildcard symbol for any constant expression we encounter in the query's predicates. Two queries are then assigned the same signature value if their internal representations match (i.e. are identical with the exception of matching wildcards and predicate ordering). The logical query signature is computed during query optimization and stored as part of the query plan; thus, if a query plan is cached, so is its signature, thereby avoiding the need to recompute it often.

(2) *Physical Plan Signature* – The logical query signature allows us to track counters across multiple instances of the same query template; however, while this is sufficient e.g., for tracking the number of times a template is executed, for applications that monitor the running time of templates (e.g. see Example 1 in Section 3), this is not sufficient, as logical query plans may result in vastly different execution plans, requiring an additional signature on the execution plan. The physical plan signature is computed similarly to the logical one, with the linearized representation being constructed over the query's *execution* plan tree.

(3) *Logical Transaction Signature* – The logical transaction addresses the problem of grouping different code paths inside a stored procedure. It is defined through the sequence of logical query signatures inside a transaction, with the transaction boundaries being defined through the outermost *begin* and *commit* brackets.

(4) *Physical Transaction Signature* – The physical transaction signature is defined analogous to the logical one, except over the sequence of physical plan signatures.

4.3. Light-Weight Aggregation Tables (LAT)

Several monitoring tasks require the ability to filter on dynamically maintained *aggregate* values of probes. Thus, it is necessary to keep some state (i.e., history of the collected probes), which can be referenced inside the conditions of ECA rules. In Example 1, it is necessary to maintain the *average* duration for each stored procedure. This ability is crucial for any monitoring application that seeks to detect outliers, changes in workload characteristics or requires any other condition that correlates current performance with the past.

LAT Functionality: In SQLCM, this functionality is provided through *light-weight aggregation tables*, which offer a mechanism for storing aggregate information over collections of probes of a single monitored class in memory. An aggregation table is defined through (a) a set of *grouping columns* and (b) a set of *aggregation functions*, both of which are defined over the attributes of the monitored class. Which monitored objects are inserted into the aggregation table is governed by rules (see Section 5). The semantics correspond to the SQL projection and aggregation operators applied to the inserted objects: the objects are grouped on the grouping attributes, and the aggregation functions evaluated over each group (as in a traditional GROUP BY SQL query). In addition to the standard aggregation functions COUNT, SUM, and AVG, SQLCM also supports a number of additional aggregation functions such as STDEV (computes the standard deviation) and FIRST and LAST, which retain the value assigned to the attribute by the first or last object inserted into the container, respectively. The latter type of aggregation can be important e.g., when using the LAT to store a representative *Query.Text* attribute (i.e., the query string) for each group in the LAT.

LATs also support an *aging* version of each aggregation function. Aging is typically important for tasks related to performance monitoring, where the baseline performance may change over time (e.g., because of increased size of tables). Thus there is a need to ignore (or give less importance) to older probe values than more recent ones. The basic idea is that at each point in time the aggregate value does not reflect any value older than a threshold *t (i.e.,* a moving window*)*. However, aging out individual values as time progresses would require to store every single value as well; instead, SQLCM groups values

into blocks that span an interval of size Δ, which are then used as the unit of aging. Note that the aging version of an aggregate requires up to 2t/Δ more storage than the non-aging version (we omit details due to lack of space).

Aggregation tables are in-memory objects at the server; but it is possible to persist them to tables (see description of the *Persist*() action in Section 5.3). For this, an aggregation table is associated with a disk-resident table with schema identical to the aggregation table, plus one additional column storing a timestamp of when the rule writing a row was triggered. The ability to persist LATs allows more complex SQL post-processing on the LAT data that may not supported by the rule engine. Furthermore, it is possible to maintain LAT data over multiple restarts of the database server, by uploading the contents of a table to a specific LAT at database startup time.

LAT specification: To illustrate what is required to logically specify an aggregation, we define a LAT for Example 1 from Section 3 as follows below. In this paper, we do not focus on the specific syntax, but rather on what elements constitute the specification of a LAT. In practice, this functionality could be exposed e.g., by appropriate system stored procedures or by introducing appropriate SQL syntax.

LAT Name: *Duration_LAT*
Grouping Columns:
Query.Logical_Signature AS Sig
Aggregation Columns:
AVG(Query.Duration) AS Avg_Duration
Ordering Columns:
Avg_Duration DESC
Maximum Size:
100 Rows

Managing LAT memory overhead: Because LATs are memory-resident, they compete for memory with operator workspace memory and buffer pool space. Therefore, we allow the ability to specify limits on the maximum size (specified either in terms of the number of rows stored or the overall row size) for an aggregation table, together with a subset of LAT columns specifying the ordering of LAT rows (and if the ordering is ascending or descending). If a LAT insertion violates the size constraint, SQLCM automatically discards the row(s) in the LAT that is "least important", i.e., having smallest value of the ordering columns of the LAT, until the size constraint is satisfied. Each evicted row is exposed as a monitored object, making it is possible to specify additional rules that e.g. persist the evicted row to a table (we omit details due to lack of space).

5. Rule Engine

The second key component of SQL is a *rule engine* that evaluates Event-Condition-Action (ECA) rules. Rules are specified as an event E, a condition C and an action A. The Action A is executed whenever the event E occurs and C is evaluated as *true*.

In order to keep the overhead of SQLCM low, the expressiveness of the rule language is limited to a relatively small set of common operations required by typical monitoring applications. We expect that any more complex logic required by a monitoring application can be achieved by post-processing the data persisted to tables from monitoring (see Section 4.3). Before describing each part of the rule engine in more detail, we briefly discuss to salient issues related to the rule engine.

Rule evaluation order: All rules are executed in a fixed order and no new rule can be triggered before the current rule has been evaluated. Furthermore, for any given event, all applicable rules are triggered before any later event is processed. This means that any action, that as a side-effect may trigger further events, is not executed synchronously. For example, if a rule triggered by a *Query.Start* event cancels this query as an action, the action only sends the cancel signal to the thread(s) currently executing the query. All other rules (if any) triggered by the same event are processed and only then does the control flow return from SQLCM to its current execution path.

Managing rule evaluation overhead: An important concern for any monitoring application is to be able to control the overhead of evaluating the ECA rules. We have observed that the overhead for rule evaluation is mainly a function of the number of rules (in case of rules that iterate over large numbers of objects, each different object-combination has to be thought of as a separate invocation of a single rule)., but does not vary significantly between rules of different complexity (see Section 6.2 for an experimental verification of this claim). Thus the user of SQLCM can control the overhead of rule evaluation through the number of rules in the system.

5.1. Events

SQLCM supports a number of different events to be used in the E-clause of an ECA-rule. Events are used to indicate when a condition is to be evaluated. The events supported in our current prototype indicate either transition points in the execution of a query (such as a query committing or aborting) or interruption of such the execution (such as an operator being blocked). For example, in our prototype, the *Query* type has a number of Events associated with it that indicate various transition

points in query execution, among them *Query.Commit* (occurring when query execution completes), *Query*.Start, *Query.Compile*, *Query.Cancel*, *Query.Rollback*, *Query.Blocked* (occurring when a operator of the query is blocked on a lock resource), and *Query.Block_Released* (occurring when the query is granted a lock on a lock resource it had been waiting on). In cases where the condition evaluation cannot be tied to a system event (for example to detect queries that are blocked for more than a given amount of time), the *Timer* object described in Appendix A can be used to instrument a background thread that periodically evaluates such rules. A *Timer* object generates a *Timer.Alert* event after a certain amount of time has passed. Finally, we note that, in principle, SQLCM can use a much wider variety of events, including events connected to connection management, database maintenance or the operating system events.

5.2. Conditions

The rule engine evaluates conditions defined over the object attributes defined in the schema (see Appendix A), the logical operators {=, !=, <, >, <=, >=} and the mathematical operators {+,-,*,/}. The order of evaluation can be specified using brackets and multiple conditions can be combined using the logical *AND*, *OR* and NOT operators. We illustrate the semantics of condition evaluation using the example of outlier detection (Section 3, Example 1):

```
Event: Query.Commit
Condition:Query.Duration
        > 5* Duration_LAT.Avg_Duration
Action: Query.Persist(TableName, Query_Text)
```

Consider the rule shown above, where the Event references an event related to the *Query* type. In cases where the monitored object in the Condition occurs in the Event clause, the scope of the rule is the object triggering the event (in the example, the query that just committed). When the Event clause doesn't reference an object in the condition (e.g., timer based events), the scope of the rule is over *all* objects of the type referenced in the Condition clause. For example, if the Event clause is empty and the Condition clause is: *Query.Time_Blocked* > 10, then the engine iterates over all query objects currently in the system. Note that if the rule references more than one object type in the Condition clause, the rule engine (logically) iterates over *all combinations* of objects of the given types currently registered with SQLCM, with the rule being evaluated once for each combination. We refer to the (combination of) object(s) that is evaluated at a given point as *in context*.

In addition to monitored object attributes, it is possible to refer to the columns of a LAT object (as *Tablename.Columnname*). In this case, the rule is evaluated for each row in the LAT for which the values of the grouping columns are identical to the corresponding probe values for the object (of the correct type) in context.

In the example rule shown above, to evaluate the condition, the row in the LAT Duration_LAT (see Section 4.3 for its definition) that matches the current *Query* object in the aggregation table's grouping column, i.e. the attribute Logical_Signature, is selected. All references to aggregation table rows are implicitly ∃-quantified; if a matching row doesn't exist, the condition is evaluated to *false*.

Note that for the applications we have outlined in Section 3, rules typically do not iterate over more than one object or LAT row. However, this functionality can be critical for other applications (e.g., any asynchronous reporting, such as a rule triggered periodically through the *timer* object, that reports queries that have been inactive or blocked for longer than a threshold value).

5.3. Actions

Our prototype implementation of SQLCM supports two basic categories of actions: actions that are attached to monitored objects (which are specified as *Object.Action(Parameters))*, and actions that do not attached to an object (e.g., *SendMail* and *Execute)*. The Action clause of the ECA rule may consist of a sequence of actions, which are executed in order. Below we give examples of Actions and indicate tasks for which these actions can be useful.

Insert (LATName) – Inserts or updates a row in the specified LAT with information about the monitored object. The appropriate row to update in the LAT is found by matching on the values of the grouping columns of the LAT. All aggregation columns in the LAT are updated as a result of this action. This action is essential for any monitoring task that uses a LAT.

Reset (LATName) – Clears the content of a LAT and frees up memory.

Persist (Tablename, Attr1, Att2, …) – Writes attributes of a *monitored object or LAT* to a table. When applied to a monitored object, this action inserts a single row into the table. When applied to a LAT, it inserts all rows in the LAT into the table. In either case, the schema of the table must match the schema of the inserted row.

SendMail (Text, Address) – Sends an email message with the given text as message body to be sent to given address. Attribute values from monitored objects and LATs can be substituted into the text string. The SendMail functionality can be used e.g., to generate an alert for the database administrator in case of a performance problem.

RunExternal (Command) – launches an external application. Similar to *SendMail*, attribute values of monitored objects and LATs can be substituted into the Command string. This action can be used in a number of ways, e.g. automatically invoking post-processing over a table into which a LAT has been persisted earlier using the Persist() action.
Cancel () – can only be applied to a *Query, Blocker or Blocked* object and has the effect of canceling the query.
Set (Time, number_alarms) – this action can only be used with a *Timer* object and governs the length of the interval after which a *Timer.Alarm* event is triggered. The second parameter governs the number of times this timer waits and generates the event; 0 disables the timer, a negative number sets up an infinite loop.

6. Implementation and Experimental Evaluation

6.1. Implementation of SQLCM

We have implemented SQLCM inside the Microsoft SQL Server database system. In this section we describe some of the key implementation issues.

Implementation of probes: The attributes of the exposed objects are gathered through probes inside the relevant execution paths. In most cases this adds negligible overhead to the execution time, as most of the attribute values (e.g., Duration of a Query object) are recorded inside the server already. An exception to this is the computation of pairs of a blocking and a blocked query. This requires the traversal on the lock-resource graph; if the rule that references these objects is triggered by a related event (such as *Query.Blocked*), the code triggering rule evaluation is simply piggybacked on the regular lock-conflict detection. Otherwise (e.g., if the rule is triggered by a Timer.Alarm event), our code traverses the lock-resource graph itself. If during this traversal we find a query waiting on a lock resource held by another query (and the requested and the held lock are incompatible), this pair is exposed as the Blocker and Blocked objects. In some cases it is ambiguous which query constitutes the blocker (e.g. when multiple queries share a resource another query is waiting on). In this case we designate one of the queries holding the resource as the Blocker.

Rule evaluation and execution: Rule evaluation is triggered in the code path of the event in the rule's Event clause, branching into the SQLCM code and then resuming execution afterwards (this is necessary for SQLCM to be of value in scenarios requiring immediate action, e.g., resource governing). Thus no context switching is required between SQLCM and query execution code.

LAT data structure: Aggregation table objects which maintain an ordering for eviction are stored using a heap structure on the ordering columns and a hash array on the grouping columns for fast row lookup. When the row width of a LAT is fixed, evicted leafs can be re-used for the newly inserted value, thereby keeping memory fragmentation low. As all rule evaluation and LAT updates occur in the same thread which triggers the event and thus potentially multiple threads can be accessing a LAT concurrently, each LAT row as well as the ordering heap as a whole and each entry in the hash table are protected through latches to avoid any conflicts. Initial experiments with large number of short queries executing concurrently on the database indicate that this latching does not introduce a new hotspot even under severe stress, as the latches are held for very short times.

6.2. Experimental Evaluation

While integrating additional monitoring functionality into the database system itself allows for many new applications, it also introduces additional overhead on the server. In this section, we demonstrate via experimental evaluation that:

- The overhead that SQLCM places on the SQL engine due to signature computation and rule evaluation is small, and scales well with the number of rules and complexity of the conditions.
- For a specific monitoring task (identifying top *k* most expensive queries), SQLCM outperforms the alternative approach of logging all events and performing post-processing. For the same task, SQLCM also provides much better accuracy compared to polling based approaches.

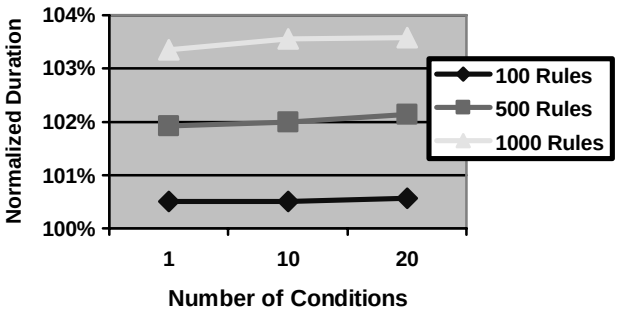
6.2.1. Evaluating the overhead SQLCM. As discussed before, the instrumentation of probes inside the server execution path only contributes little overhead to the DBMS. The areas where SQLCM adds measurable overhead are the computation of signatures, the evaluation of large numbers of rules and the maintenance of a large number of aggregation tables. In the following, we experimentally evaluate the overhead of these components.

Setup: The experiments for overhead of signature computation were run on an x86 1680 MHz processor with 256MB of memory running Windows 2000 Server. The rest of the experiments were run on an x86 900 MHz processor with 384MB of memory running Windows 2000 Server. In the experiments in Section 6.2.2, we used a workload on the TPC-H schema [15] (with 6 million rows in the *lineitem* table) consisting of 20,000 short

single-row selections from the *lineitem* and *orders* table interleaved with 100 selections of 1000-2000 rows from a join between *lineitem*, *orders* and *parts*. In all experiments we executed the exact same queries (i.e., identical constant parameters) in order. The machines were rebooted before each experimental run, disconnected from the network and had all background services disabled. Repetitions of individual experiments showed very little variance in the measured performance.

Overhead of signature computation. The overhead for computing the linearized representation of the query depends on the complexity of the query itself. In the SQLCM prototype, the query signature is computed once during optimization and then cached with the query plan. Therefore, we measured the overhead of the signature computation relative to the total time used for *optimization*. We ran these experiments for a variety of synthetic and real workloads and discovered that the relative time decreases with the complexity of the queries. The extreme points in our measurements were 0.5% (for single-line selection queries without conditions) and 0.011% (for complex TPC-H queries), demonstrating that the overhead of signature computation is indeed very low.

Figure 2. Rule evalulation overhead



Overhead of rule evaluation and LAT maintenance. In this experiment we measure the overhead for rule evaluation through by means of stress test using queries of very short duration and a large number of rules. As a baseline, we measured the overhead for executing 10,000 single-row select statements on a 6 million row *lineitem* table (using the TPC-H schema) that use a clustered index. Then we executed the same workload again and measured the additional overhead caused by a varying number of rules (varying between 100 and 1000, all of which were evaluated for every single query) of varying complexity (the number of atomic conditions varied between 1 and 20). In addition, each rule stored a summary of the observed workload in a different in-memory aggregation table of fixed size, storing all attributes (incl. query text) of the last 10 queries seen, indexed by the signature id.

Figure 2 depicts the additional overhead caused by rule evaluation and LAT maintenance. Most importantly, even under the tested extreme conditions the additional overhead caused by SQLCM is less than 4%, and negligible for more realistic scenarios. In addition, one can see that the complexity of rules has very little impact on the additional overhead; rather, the overhead due to LAT maintenance (i.e., the overhead of inserting and evicting rows from the LAT and the memory consumed by LATs) is the biggest factor. By controlling the number and size of LATs, effective management of the SQLCM overhead is possible (see Section 4.3).

6.2.2. Comparison with Alternative Monitoring Solutions. In this section we contrast the efficiency of SQLCM with other possible design choices for database monitoring. In this experiment, we examine the cost of a simple monitoring task – determining the 10 most expensive queries during a given workload through monitoring – for different solutions. This type of query is often used in practice to alert administrators to instances problematic combinations of query parameters. We study the following solutions to this task:

(a) *Logging all queries*: In this approach, we write out all information on each committed query to a reporting table (we used a single rule writing the statistics using the *Report()* action). As monitoring and reporting is not integrated in this scenario, we force synchronous writes. The final result (top 10) is then obtained by running a SQL query on the reporting table. This solution corresponds to a solution incorporating push without filtering inside the server (similar to event logging). We will refer to this approach as **Query_logging**.

(b) *Polling the current state*: Here a client monitoring application repeatedly polls from the database a snapshot of the currently active queries and their execution time and computes the most expensive ones externally. This corresponds to a pull based solution, where the necessary filtering is performed on the client. Note that this approach may not identify the correct queries, with the error dependent on the frequency of polling. We will refer to this approach as **PULL**.

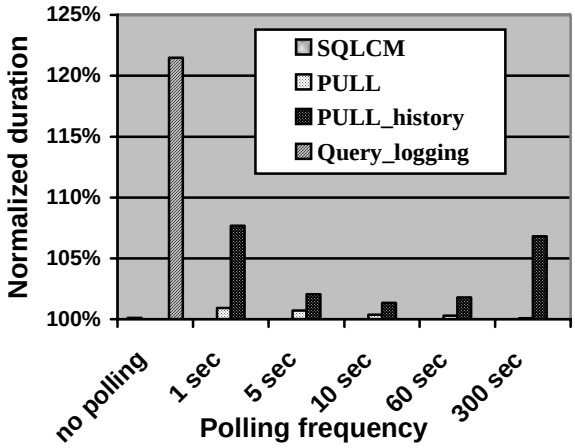
(c) *Polling the historic state*: This approach is identical to case (b), except for the fact that the server keeps a history of all queries and their execution times, which is only erased when being 'picked up' by the outside monitoring application. While this is not a realistic solution in practice, we use it to model a solution without push or filtering, but keeping history. We will refer to this approach as **PULL_history**.

(d) Using the SQLCM framework: Here, we use a LAT which stores the 10 most expensive queries seen (storing the query text and duration). At the end of the workload,

the LATs contents are written to a table using the *Persist*() action. We refer to this approach as **SQLCM**.

To compare these different approaches, we measured the execution times of the original workload without any monitoring enabled as well as the times for the different monitoring approaches. In case of the two polling based approaches **PULL** and **PULL_history**, we also varied the polling rate between 1/sec to 1/5minutes. The results are shown in Figure 3. As far as the impact of the monitoring is concerned, **SQLCM** (requiring less than 0.1% additional overhead and thus almost imperceptible in the figure) causes the least degradation in performance, followed by **PULL**. However, for none of the different polling frequencies did **PULL** result in the correct answer; rather, when polling every second, 5 of the 10 most expensive queries were not part of the **PULL** result set (when of polling every 5 seconds, 7 queries were missed, and when polling every 10 sec or more infrequent, 9 queries were missed). Not surprisingly, the additional overhead of this approach increased with an increased polling frequency.

Figure 3. Efficiency of different approaches



The PULL_exact approach did yield the correct results, but caused significantly more overhead than SQLCM. In addition, as shown in Figure 3, finding the right polling rate is a tuning problem – if polling is too frequent, the polling itself causes significant overhead, if it is too infrequent, storing the historical state requires significant memory, in turn degrading the server's ability to cache pages. Not surprisingly, the Query_logging approach resulted in the biggest degradation in performance (>20%).

We also executed the same set of experiments on a real (customer) workload used within Microsoft, resulting in similar trends, which are not reported for lack of space. Finally, we note that differences between SQLCM and the other techniques will add up when multiple monitoring tasks are executed in parallel.

7. Related Work

Today's commercial database systems have support for *event logging* as well as for obtaining a *snapshot* of system counters by polling the server. However, as explained in the introduction and demonstrated experimentally, these mechanisms may be inadequate (or significantly less efficient) for tasks that require monitoring a large number of events due to either: (a) overhead incurred on server by event logging or frequent polling, or (b) loss in accuracy when not polling frequently enough.

IBM DB2 Health Center [7] is a tool that continuously monitors "health" of the database system and alerts DBA by email/pager or by logging the problem. An alert is raised when the value of the counter being monitored crosses a threshold. Similar to Health Center, there are several third party monitoring tools [2,16,13] for today's commercial database systems with similar functionality. In contrast to these tools, which are client applications, SQLCM can be viewed as new server-side infrastructure for enabling a broader class of monitoring applications that are either more efficient or enable functionality that is not possible with today's client side monitoring tools.

There have been several recent papers on query processing over streaming data e.g., [3,11,5]. In principle, a lightweight aggregation table (LAT) in SQLCM (see Section 4.3) could be viewed as a standing aggregation query over streaming data (generated by the DBMS). In comparison to the state of the art on streaming data we expose much simpler techniques to deal with memory constrictions at the server that do not allow keeping the entire state. Some of these techniques would typically not be applicable to streaming data scenarios, but scale for our specific monitoring applications. Conversely, a number of techniques discussed in the context of streams could potentially be valuable for SQLCM.

Rule-Production systems in DBMS have been extensively studied in the context of discriminatory networks such as TREAT [12] or RETE [4]. However, the scope of this work is the problem of efficiently matching a large collection of patterns to a large collection of objects, which is more general and expensive than required in the monitoring context. Similarly, the inbuilt support for triggers in database systems and the general purpose notification services e.g., [14], could be considered alternatives to the ECA rule engine used in SQLCM. However, both these mechanisms are more general and heavyweight than necessary for SQLCM, where one of the major requirements is to keep the runtime overhead of rule evaluation (see Section 6.2) as low as possible.

There is a large body of work on automatically tuning database systems [10, 17] by observing and dynamically adapting to system usage. Although not the focus of this paper, as discussed in Section 3, one of the important uses of the SQLCM framework can be to facilitate self-tuning and corrective actions. Some of these tuning tasks such as resource governing or admission control are enabled by virtue of the fact that SQLCM is implemented inside the database server.

8. Conclusion

In this paper we present SQLCM, a server side infrastructure that enables many common database monitoring tasks to be accomplished efficiently. The power of SQLCM is a result of the ability to support flexible, in-memory filtering and aggregation inside the database server combined with the ability to take actions based on the monitored data. In the future, we will continue to explore the use of SQLCM for a variety of other monitoring tasks including its application for internal database system tuning.

9. References

[1] The AutoAdmin Project at Microsoft Research. *http://research.microsoft.com/dmx/AutoAdmin/*

[2] BMC PATROL® Database Knowledge Modules, *http://www.bmc.com/*

[3] Carney D., Çetintemel U., Cherniack M., Convey C., Lee S., Seidman G., Stonebraker M., Tatbul N., Zdonik S. Monitoring Streams – A new class of Data Management Application., *Proc. of the VLDB 2002.*

[4] C. Forgy. RETE: A fast algorithm for the many patterns/many objects problem. *Artificial Intelligence. 19(1): 17-37, 1982.*

[5] Dobra A., Garofalakis M., Gehrke J., Rastogi R. Processing Complex Aggregate Queries over Data Streams. *Proc. of ACM SIGMOD 2002.*

[6] Chakravarthy S., Blaustein B., Buchman A.P., Carey M., Dayal U., Goldhirsch D., Hsu M., Jauhari R., Ladin R., Livney M., McCarthy D., McKee R., Rosenthal A. HIPAC – A Research Project in active, time-constrained Database Management. *Tech. Report XAIT-89-02, Xerox Advanced Information Technology, 1989.*

[7] IBM DB2 Universal Database: *http://www-3.ibm.com/software/data/db2/udb/v8/.*

[8] IBM DB2 Universal Database: System Monitor Guide and Reference. *ftp://ftp.software.ibm.com/ps/products/db2/info/vr8/pdf/letter/db2f0e80.pdf.*

[9] Goldstein J., Larson P. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. *Proc. of ACM SIGMOD 2001.*

[10] Lohman G., Lightstone S. SMART: Making DB2 (More) Autonomic. *Proc. of VLDB 2002.*

[11] Madden S., Shah M., Hellerstein J., Raman V. Continously Adaptive Continuous Queries over Streams. *Proc. of ACM SIGMOD 2002.*

[12] D.P. Miranker. TREAT: A Better Match Algorithm for AI Production Systems. *Proceedings of the National Conference on AI, 42-47, August 1987.*

[13] NetIQ Diagnostic Manager for SQL Server, *http://www.netiq.com/products/sdm/default.asp.*

[14] Praveen Seshadri: Building Notification Services with Microsoft SQLServer. *Proc. of ACM SIGMOD 2003.*

[15] Transaction Processing Performance Council. The TPCH Benchmark. *http://www.tpc.org/tpch/default.asp.*

[16] Veritas Indepth for SQL Server, *http://www.precisesoft.com/*

[17] Weikum G., Moenkeberg A., Hasse C., Zabback P. Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. *Proc. of VLDB 2002.*

Appendix A. SQLCM Schema

The current SQLCM implementation exposes 5 *monitored object* classes: *Query, Transaction, Blocker, Blocked* and *Timer*.

The ***Query Class*** contains the following attributes:

Attribute Name	**Type**	**Comment**
ID	Integer	
Query_Text	String	Query Text String
Logical_Signature	BLOB	
Physical_Signature	BLOB	
Start_Time	Datetime	
Duration	Float	
Estimated_Cost	Float	
Time_Blocked	Float	
Times_Blocked	Integer	
Queries_Blocked	Integer	#of queries blocked
Number_of_instances	Integer	
Query Type	atomic	Type∈ {UPDATE, SELECT,INSERT, DELETE}

The **Transaction** class has identical attributes to the *query* object, except for the plan signatures, which are exposed as a list of integers.

The **Blocker** and **Blocked** classes represent combinations of queries where the *Blocker* query owns a lock on a resource incompatible with the lock the *Blocked* query is waiting on (on the same resource). They have the same schema as the Query object.

The **Timer** class is provided to facilitate periodic invocation of rules that cannot be tied to a specific event. The system exposes a set number of Timer objects. These timers can be set (using the Set() action) to a specific wait period, after which the create a *Timer.Alarm* event. A Timer object also exposes the current time as an attribute.