

A Framework for Enforcing Application Policies in Database Systems

Lin Qiao¹, Basuki Soetarman², Gene Fuh², Adarsh Pannu², Baoqiu Cui³,
Thomas Beavin², William Kyu²

¹IBM Almaden Research Center, San Jose, CA

²IBM Silicon Valley Lab, San Jose, CA

³Yahoo! Inc., Sunnyvale, CA

{lsqiao,basuki,fuh,adarshrp,beavin,wkyu}@us.ibm.com, bcui@yahoo-inc.com

ABSTRACT

As database systems have grown in terms of scale and complexity, administration tasks have become increasingly difficult and time-consuming. A scarcity of skilled database professionals has meant that human costs have begun to dominate the total cost of ownership (TCO) of a database system. Database vendors are under immense pressure to provide solutions that make their products easy to administer in areas such as problem diagnostics, monitoring, query tuning, access control and system configuration.

To address this issue, we have built a framework that allows control over many administration operations via the use of policies. Users can uniformly define, manage and enforce policies to affect disparate aspects of the system.

In our framework, policies are declarative constructs that are comprised of type, scope, condition and action. Policy groups cover query monitoring and tuning, query prioritization, system configuration, access control, report generation, etc. Policy scope defines the domain over which policies apply. Policy actions are performed if certain conditions are true. This framework has been fully integrated into DB2 for z/OS V9. Using detailed system performance evaluations, we report that enforcement of policies is largely a function of data-collection granularity. Under the setting for normal monitoring with minimal report, the overhead on system performance is very low (0.1%).

Categories and Subject Descriptors

H.2 [Information Systems Database Management]: Miscellaneous

General Terms

Management

Keywords

Policy, Database System, Self-managing

1. INTRODUCTION

Over the years, database systems have grown substantially in complexity. There has been a marked increase in the functionality provided by DBMS engines along with the ability to support large hardware configurations (such as multi-way clusters), huge data volumes (often in terabytes) and extremely complex queries. As applications have scaled up to take advantage of these capabilities, administering such systems has become very difficult. Tasks such as problem diagnostics, physical schema tuning, query tuning and access control now take a significant amount of database administrators time. To make matters worse, database experts in these areas are insufficient and expensive. On the other hand, improper configuration and tuning often leave systems vulnerable to performance, availability and other problems. At a minimum, this sub-optimality can result in loss of valuable time and resources. At an extreme, the effect could be catastrophic for mission critical enterprise applications and can lead to serious financial implications.

These problems have motivated database vendors to look at ways to enable smooth product operation in environments even with limited access to skilled administration personnel. Several solutions are already available in the marketplace spanning a range of functionality and sophistication. Even so, database administration remains a difficult task:

- Existing solutions for areas such as monitoring, performance tracing, and tuning are non-uniform [11, 12, 10, 9]. Often times, administrators and tools developers are faced with a potpourri of piece-meal solutions, each handling a different aspect of the problem. For instance, most commercial databases continue to have different sets of interfaces for profiling, for specifying system configuration parameters, for controlling query behavior and performance, etc.
- When it comes to configuration and tuning, one size does not fit all. With the advent of server consolidation, it is not uncommon for a single database instance to support different classes of applications ranging from simple OLTP to mix-workload and decision-support queries. However, the configuration and tuning options afforded to applications are often coarse-grained affecting a class that is larger than necessary. It is extremely difficult to strike a balance in attaining global optimization that satisfies all applications. Tuning and optimization criteria for one application may not necessarily apply to other applications. At worst, the criteria might even be contradictory and cause a negative impact to the other applications.
- There is little support for autonomic problem detection. The database system is usually tasked with raw monitoring and performance data collection. However, subsequent analysis and actions still require human intervention.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD'07, June 12–14, 2007, Beijing, China.

Copyright 2007 ACM 978-1-59593-686-8/07/0006 ...\$5.00.

We introduce a framework that alleviates each of the aforementioned problems using the unifying notion of policies. A policy is a declarative means for expressing directives to be carried out by the underlying system, which, in this case, is a database engine. Policies make the system perform designated actions under pre-determined circumstances. Policies have several key benefits. Firstly, they have a broad range of applicability. They are useful not only for expressing monitoring or tuning related directives, but they also serve as the perfect medium for directives relating to aspects such as access control, system configuration, resource limits, auditing, etc.

Secondly, our framework provides for the specification of policies at various levels of granularity. At one end of the spectrum are coarse-grained policies that affect the behavior of the system as a whole. These allow the use of global settings. At another end are fine-grained policies only affects a group of statements or specific applications.

Lastly, policies are not merely limited to controlling database operations under normal circumstances. They have been extended to cover detection of complex events including exceptions, and this aids in early problem diagnosis. Detecting spikes in query performance is one example of such a circumstance.

Our framework is open, flexible and extensible. It provides a unified mechanism to manage and enforce policies related to different domains and areas of applicability in a consistent manner. Users are allowed to customize solutions to be applicable only for a specific scope, thus eliminating interference between solutions. The extensibility of the framework comes from the property that its components can be extended to cover additional domain or area of importance without disrupting the existing support. Finally this framework is generally applicable to other kinds of systems or applications, not only limited to database systems.

The rest of this paper is organized as follows. In Section 2, we explain how policies are declared in our system to govern database behavior. Section 3 describes our framework, each component in the framework, and interactions between these components and existing components in the DBMS engine. Section 4 shows the application scenarios of our policy framework. We discuss experimental results in Section 5. We review related work in Section 6 and conclude in Section 7.

2. POLICY DECLARATION

A policy is a declarative means for expressing directives to be carried-out by the underlying system, which in this case, is a database engine. Policies make the system perform specific actions under a pre-determined circumstances; they allow the focus of changes to affect only a specific statement, a set of statements, applications, or the database as a whole. Hence, a policy is identified by a quadruple $\langle type, scope, condition, action \rangle$. Policies have a domain or area of applicability, which they may be put in place to govern, which we refer to as *type*, such as

- *query monitoring*: policies to determine which queries to monitor and the kind of monitoring to perform;
- *query tuning or access path generation*: directives to guide the query access path generation to yield optimal performance;
- *prioritizing query execution*: policies to guide the execution priority for a query, a set of queries, or applications;
- *data collection*: policies for indicating the desirable level of granularity and the amount of data collection that should be performed and reported. This domain is used to complement other domains that collect data and produce reports, for example query monitoring;
- *access control and authorization*: policies to regulate access con-

trol to resources;

- *system configuration*: policies to determine the preferred system configuration for running a specific application or query;
- *resource limits*: policies to limits the resource usage and specifies the action to take when the threshold is reached;
- *auditing*: policies to govern what kind or part of database access require audit log.

A scope determines the extent and level of the policy impact, which is quite useful for isolating a problem or solution areas. In a database system, a scope may cover the whole database, a specific application, a set of statements, or an individual statement only. A set of statements in a scope can be selected based on various statement properties, for example, authorization ID, IP address from where the statement is submitted, application name, package name, etc. Here a package is a block that can contain only declarations, default statements, and procedure blocks. One application can have multiple packages. (DB2 uses name scope of application, plan, package. However, to simplify description in this paper, we illustrate only application, package scope.) The action or directive specifies the task to be taken when the policy is in effect. The actions are report generation, access path generation, set system-level configuration, cancel a query, etc. A condition further parameterizes when an action should be taken. A condition can be one of the followings: a query accesses certain objects; performance exceptions, such as $cpu\ time > threshold$, elapsed time of current run is extremely larger than average elapsed time; query optimizer estimation errors, such as cardinality estimation error, cpu time estimation error, IO operation estimation error, etc.

We enumerate some policies that can be defined in our system as follows. Wildcard '*' means ANY.

Example 1:

```
[type] query monitoring
[scope] application=ERP, package=Transaction
[condition] if this query execution time > 3 times
              average query execution time
[action] push out access path information and runtime
              statistics information
```

Example 2:

```
[type] query tuning
[scope] application=ERP, package=*
[condition] *
[action] run queries in parallel, with degree=5
```

Example 3:

```
[type] query tuning
[scope] application=ERP, package=*
[condition] if a query involves more than 10 tables
[action] enable star-join
```

Example 4:

```
[type] resource limit
[scope] application=ERP, package=Personnel
[condition] if queries run for more than 3 minutes
[action] cancel the query and generate a detailed report
```

Example 5:

```
[type] system configuration
[scope] application=*, package=*
[condition] *
[action] set sort heap size = 3MB
```

Example 6:

```
[type] access control and authorization
[scope] authID='first line manager', IP addr=1.99.60.5
[condition] if a query involves EMPLOYEE table
[action] only classified columns (Name,Rank,Salary) of
              EMPLOYEE table can be accessed
```

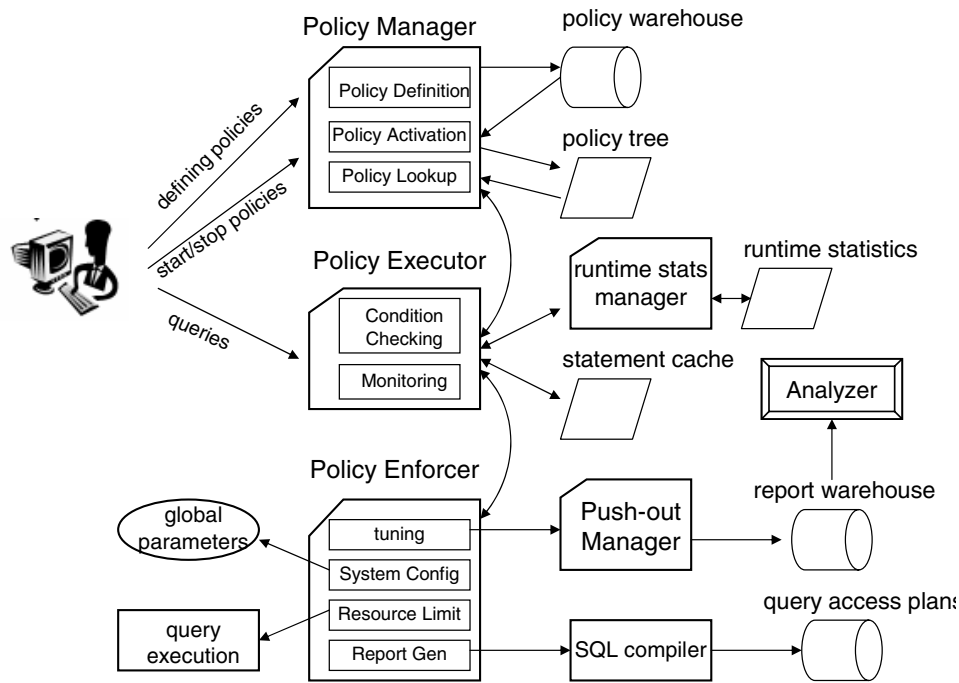


Figure 1: System Architecture

Example 7:

[type] auditing
 [scope] *
 [condition] if a statement attempts to access the SALARY column of EMPLOYEE table
 [action] Log the statement and relevant information

Example 8:

[type] query prioritizing
 [scope] authorID='employee', IP addr=1.99.60.5
 [condition] *
 [action] degrade query priority every 2 minutes

In addition to the general policies, our system is able to define and enforce special policies to regulate the execution of other policies. We call these kinds of special policies *meta-policy*. Meta-policies are mainly used to limit the impact of policy checking and execution on front-end applications.

Example 10:

[type] meta-policy
 [scope] application=ERP, package=Transaction
 [condition] occurrence of action in Example 1 > 30
 [action] do not push out access path information and runtime statistics information

For instance, as report generation can be I/O intensive, it is necessary to control the execution of such action. Otherwise, it will interfere with running queries. The policy in Example 8 stops report generation if there are too many actions taken when executing the policy in Example 1.

3. SYSTEM DESCRIPTION

3.1 Framework

The framework outlines the interface and mechanism for supporting policies. The high level architecture is illustrated in Figure 1. As the first step, users define policies that are required to be in effect for the optimal operation of the system. The defined

policies are managed by *Policy Manager*. *Policy Manager* consists of three components: *Policy Definition*, *Policy Activation*, and *Policy Lookup*. All defined policies are stored and organized in *policy warehouse*, which is a set of policy tables. Once policies are defined, the user might choose to activate the policy facility. The *Policy Activation* component reads policies from policy warehouse and then builds a policy tree in memory representing activated policies. It can also activate or deactivate policies on the fly. *Policy Activation* does some processing to ensure the policies are consistent and performs conflict resolutions as necessary. Finally, given a database request, *Policy Lookup* searches the policy tree for all applicable policies.

After policies are activated, all database request streams to the system are subjected to scrutiny to determine if a particular input is affected by any of the activated policies. In this process, the *Policy Executor* reads the input and calls the *Policy Manager* to perform a policy look-up. If a match is found, the *Policy Executor* may monitor query performance by maintaining the run-time statistics. Eventually, it checks to see if specified policy conditions are satisfied. If so, the policies need to be enforced. *Policy Executor* passes the flow to *Policy Enforcer*.

Note that the incremental cost for adding a new policy is low, but we may touch a lot of code. It can be made easier to have an abstract layer which encapsulates all the features of policies. Hence, we can add new policies by adding new code under this abstract layer, transparent to other components. This will be our future work, when our system grows with more and more policy types.

Policy Enforcer in turn selects the appropriate actions and initiates them. The actions are enumerated in Figure 1, but not limited to, *tuning*, *system configuration*, *resource limit*, and *report generation*. The tuning actions take optimization hints and pass them to SQL compiler. SQL compiler generates query access plan based on the hints and store the plan in system catalog for static SQL queries. The system config actions override any global parameters that may have been in effect. The resource limit module stops the execution

of a query if the resource usage exceeds the given threshold and triggers a report generation. The report generation actions further drive *Push-out Manager* and actually complement other actions by providing a report of the action and storing it in *report warehouse* in a configurable amount and granularity level of detail. These reports can be pulled by external *Analyzer* for problem diagnosis or auditing purposes.

Consider policies in Example 2 and Example 3. These policies are submitted by the user or application and after passing consistency checking, stored as two rows in policy warehouse. After users explicitly activate policy manager by running “START POLICY” command, these two policies become effective. The scope of both policies is ERP application. These policies are for query tuning and are implemented during access path generation. In particular, the actions are to generate a parallel execution plan or to generate an execution plan with star join. Queries or statements belonging to other scopes are not affected. This effectively eliminates any surprising results that might occur if the policies are applied globally. When these policies are active, the stream of input queries received from users or applications are examined for the matching scope by *Policy Executor*, who further calls *Policy Manager* with a request for *Policy Look-up* into the existing entries in *policy tree*. When there is a match, *Policy Executor* processes the query as it normally does and checks whether the query involves more than 10 tables. Based on the checking result, it invokes the *Policy Enforcer* to carry out the appropriate action in the *tuning module*, which in this case would be passing optimization parameters to SQL compiler, to generate a parallel execution plan or to include star join in the generated plan.

This framework is open, flexible, and extensible. It provides a unified mechanism to define, manage, and enforce policies related to different domains and areas of applicability in a consistent manner. The declarative nature of a policy expression gives the user some degrees of flexibility in defining policies independent from the underlying database engine. Users are allowed to customize solutions to be applicable only for a specific scope, thus eliminating interference between solutions. In other words, policies are used to isolate a problem into a more specific context, upon which either general or customized solutions are derived. Policies also provide a flexible means to control the default behavior of the system toward achieving specific goals. The extensibility of the framework comes from the property that the *Policy Enforcer* can be extended to cover additional domain or area of importance. *Policy Manager* also allows more policy domains to be added without disrupting the existing support. In addition, *Policy Manager* and *Enforcer* are architected as separate parts of the underlying engine so they can be potentially delivered as plug-ins, when appropriate. Finally this framework is generally applicable to other kind of systems or applications, not only limited to database systems.

3.2 Policy Manager

3.2.1 Policy Tree

In general, policies can be organized as a policy tree, as shown in Figure 2. Notice that the policy scopes have hierarchical structure. Hence, the policy tree is constructed based on the policy scopes. After the policy engine is started, one policy tree per system will be created in memory. There are two types of nodes in the policy tree, one is scope node and the other is policy node. Subsequent scope nodes along the tree define the scope of the policies. A policy nodes consists of the group, condition and action elements of the policies. Each scope node can have two types of children, one is a subscope node and the other is a policy node applied to that scope. The root

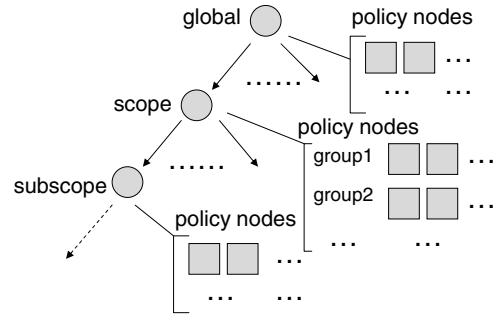


Figure 2: Policy Tree

of the tree refers to a *global* scope, meaning all the queries qualify that scope. Hence, the policies represented by the policy nodes of the global scope will apply to all the queries. A scope node can have 0, 1 or more child subscope nodes or child policy nodes.

There are three kinds of conflicts in a policy tree.

1. Identical scope. This can be resolved using timestamp.
2. Overlapping scope. This is resolved by precedence, e.g. Authorization_ID + IP address takes precedence over application+package.
3. Inherited scope. A scope node will inherit all the policy nodes from its parent, except those conflicting with its own policy nodes. This is resolved by that child scope takes precedence over parent scope.

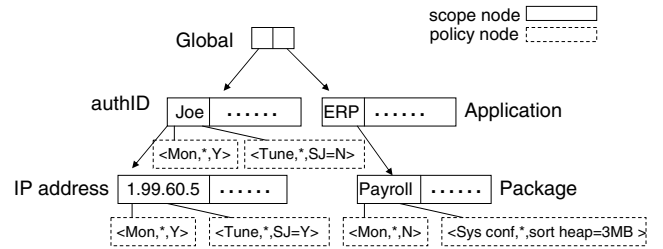


Figure 3: Policy Tree Example

As an example, we show one instance of the policy tree implemented in our system in Figure 3. There are two types of subsopes under the global scope, one is *Authorization ID* and the other is *Application*. For one subscope, there can be multiple entries. For instance, Authorization ID can be 'Joe', 'Tom', etc. Authorization ID has *IP Address* subscope, while Application has *Package* subscope.

3.2.2 Policy Definition

Policies are managed and stored into tables. We normalize the representation of policies into two tables, namely POLICY_SCOPE table and POLICY_CONDITION_ACTION table, as shown below in Table 1 and Table 2. Policy_ID is used to join a row in POLICY_SCOPE and a row in POLICY_CONDITION_ACTION, because there may be many (condition, action) pairs associated with one policy scope. In the POLICY_SCOPE table, a policy scope contains either a valid (Appl_name, Pkg_name) or a valid (Author_ID, IP_addr). Timestamp is recorded when the scope is created or is updated. It is mainly used for policy conflict resolution. Finally, a policy scope can be enabled or disabled. This feature is useful

Column name	Description
Policy_ID	The unique identifier for the policy
Appl_name	Name of application containing the queries
Pkg_name	Name of package containing the queries
Author_ID	Authorization ID who issues the queries
IP_addr	IP address from where the queries are issued
Timestamp	When the policy scope is inserted or updated
Enabled	If the policy scope is enabled or disabled

Table 1: POLICY_SCOPE Table

Column name	Description
Policy_ID	The unique identifier for the policy
Cond_literal1	Literal policy condition
Cond_literal2	Literal Policy condition
Cond_string	String policy condition
Action	Policy actions
Timestamp	When the policy (condition,action) pair is inserted or updated

Table 2: POLICY_CONDITION_ACTION Table

when the user want to turn on and off a policy scope on the fly without having to delete it and then insert it back. In the POLICY_CONDITION_ACTION table, Timestamp column is also used for conflict resolution and Enabled column is to turn on and off a policy (condition, action) pair on demand.

A policy tree is constructed from the result of POLICY_SCOPE $\bowtie$ POLICY_CONDITION_ACTION. Note that there can be conflicting policies defined in POLICY_SCOPE table and POLICY_CONDITION_ACTION table. Conflicts are first resolved by sorting the join results based on the Timestamp columns from both tables. So the latest defined policy takes effect. When processing a new policy, the policy manager first traverses the policy tree to locate the right scope. A new scope node is created if no such scope node exists. During the tree traversal, policy nodes from the parent scope node are passed down to the children. Policies nodes are created for both inherited-and-not-conflicting policies and new policies. Precedence is also defined between sibling scopes. For instance, the (Authorization ID, IP Address) scope can take precedence over (Application, Package) scope. In Figure 3, two policy nodes are created for authorization 'Joe' with IP address '1.99.60.5'. Node $\langle \text{Mon}, *, Y \rangle$ means policy type is monitoring, no condition, and monitoring is set to yes. This is a node inherited from its parent - authorization 'Joe'. Node $\langle \text{Tune}, *, SJ=Y \rangle$ means policy type is tuning, no condition, and star join should be considered if possible.

3.2.3 Policy lookup

Policy lookup is a mapping function $F^2(F^1(\text{scope}), \text{group}) \rightarrow (\text{condition}, \text{action})$. First, F^1 maps a query scope to a set of policy nodes in policy tree. To speed up lookup, one hash table is created for each scope node. Given a set of policy nodes and a policy type, F^2 further produces a set of (condition,action) that should be considered for a query. In Figure 3, consider a query that has authorization 'Joe' and is issued from IP address '1.99.60.5' with application name 'ERP' and package name 'Payroll'. Here are the example lookups for that query.

$F^2(F^1(\langle \text{'Joe'}, 1.99.60.5 \rangle), \langle \text{'ERP'}, \text{'Payroll'} \rangle), \text{'Mon'})$
 $\rightarrow \langle *, Y \rangle$
 $F^2(F^1(\langle \text{'Joe'}, 1.99.60.5 \rangle), \langle \text{'ERP'}, \text{'Payroll'} \rangle), \text{'Tune'})$

$\rightarrow \langle *, SJ=N \rangle$
 $F^2(F^1(\langle \text{'Joe'}, 1.99.60.5 \rangle), \langle \text{'ERP'}, \text{'Payroll'} \rangle), \text{'Tune'})$
 $\rightarrow \langle *, SJ=Y \rangle$
 $F^2(F^1(\langle \text{'ERP'}, \text{'Payroll'} \rangle), \langle \text{'ERP'}, \text{'Payroll'} \rangle), \text{'Sys Conf'})$
 $\rightarrow \langle *, \text{sort heap}=3\text{MB} \rangle$

3.2.4 Policy Activation

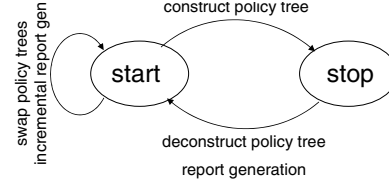


Figure 4: Policy Activation/Deactivation

After a user starts a policy engine, policies are read from the policy tables and an in memory policy tree is constructed, as shown in Figure 4. The policies included in the policy tree after conflict resolution are written to a POLICY_HISTORY table. This is called policy activation. If a user stops the policy engine, the policy tree is de-constructed and incoming queries are not checked. This is called policy deactivation. Note that, one type of action needs to be taken during policy deactivation. That is report generation. Some report generation actions are triggered when a query leaves the statement cache. However, if such reports have not been generated yet because the queries are still in cache, while the policy engine is deactivated, the report generation actions should take effect immediately. This way, when the queries are purged from cache later, when there is no policy tree to be checked against, the needed reports will not be lost.

A user can also restart a policy engine, by issuing a start command while the engine is still active. A new policy tree T_{new} will be constructed by reading the join results from policy tables. Although the new tree T_{new} and old tree T_{old} co-exists, T_{new} takes effect immediately and T_{old} becomes a secondary structure. T_{old} is then compared with T_{new} to enforce policies that are no longer covered by T_{new} . For the policies in $T_{old} - T_{new}$, if they are to generate reports, reports are generated immediately for the matched queries due to the reason explained before. T_{old} will be deleted afterward. For the policies in $T_{new} \cap T_{old}$, if the matched queries are still in the cache, the policies remain effective and no further action due to restart should be taken. In addition, newly incoming queries will be checked against that part of the tree. For the policies in $T_{new} - T_{old}$, these are the newly effective policies, and only the newly coming queries will be checked against that part of the tree.

3.3 Policy Executor

Policy executor is the component driving the execution of policies. For a given query, it first calls the lookup interface provided by policy manager to check if there are matched policies. If matched policies are found, it dispatches to an appropriate component in policy enforcer. For each query, the policy executor is invoked twice, once at query bind time and once at query run time.

During query bind time, policy executor looks up the policy tree for the policy with the same scope as the query scope. In addition, it searches for the policy type = 'query tuning' or 'system config'. If matched policies are found, corresponding $\langle \text{condition}, \text{action} \rangle$ pairs are retrieved from the policy tree.

During query run time, policy executor only looks up the policy tree once when the query is first cached into the statement cache. During the query lifetime in the statement cache, no further lookup

is needed. For query runtime policy lookup, policy executor invokes lookup calls with a policy scope and policy type = 'query monitoring' or 'resource limit'. If policies with group 'monitoring' are found, policy executor needs to construct an internal structure for the monitored statistics as described by the policies. (The details about performance statistics will be described in the next section.) There are two types of monitoring policies. One is normal monitoring, meaning the policy condition is ANY, and query report is generated when the query is purged from statement cache. Normal monitoring policies are used for health checking purpose. The other is exception monitoring. In this case, the policy condition is usually about exceeding a certain threshold, and query report is generated whenever the condition is satisfied. Such threshold describes both performance exception and optimizer estimation exception. Exception monitoring policies are used to capture and diagnose performance problems.

After the lookup is done, policy executor dispatches the flow to the corresponding component in policy enforcer. It passes `< condition, action >` to policy enforcer as the input.

Note that, for the "monitoring" policy, the policy executor needs to maintain a set of run-time performance statistics as well as counters. The counters are used to keep track of the number of reports generated per exception, per query, and per statement. They are used to enforce meta-policies, which we will describe in detail in Section 3.5.

3.3.1 Run-time Performance Statistics

There are two kinds of performance statistics. One is the *resource statistics*, which is about resource consumption, such as query elapsed time, cpu time, IO time, number of IOs, number of sorts, number of parallel groups, etc. These statistics can be used to detect and capture queries with resource usage beyond normal expectation. The other is the *operator statistics* about the results from each query operator, such as cardinality, filtering factor, etc. These statistics are compared with the estimated statistics by optimizer which is stored in catalog. A big difference may indicate out of date statistics in catalog tables.

The resource statistics are maintained at per query and per statement levels. Query statistics are for each execution of a single query, while statement statistics are for multiple executions of a query that is the same statement cached in the statement cache. Per query statistics are valid during that query execution, while per statement statistics are maintained throughout the life span of the statement in the statement cache. At the end of query execution, per query statistics are collected and per statement statistics are accumulated. For instance, per query statistics can be query elapsed time. Per statement statistics can be the average query elapsed time. For each query, the per query statistics are compared with the per statement statistics. If, say, per query statistics are 300% more than per statement statistics, this query execution is marked as an exception, and certain action should be taken. Per query statistics are discarded afterward.

The operator statistics are collected after the execution of each query operator, such as table scan, index scan, join, etc. Estimated statistics are stored in memory, along with the query access plan, and are compared with their run-time counterpart for each operator in the query access plan at the end of this query execution. If, say, they are 500% smaller or bigger than their run-time counterpart, this query execution is marked as an exception and a detailed report will be generated. This feature is very useful to compare run-time results with estimated results side by side if mismatch happens. The output can further feed into tools, such as statistics advisor, to suggest recollecting statistics or building additional statistics. For in-

stance, a wrongly estimated filtering factor from join operator may suggest creating and collecting multi-column statistics on the join columns. The output can also feed into index advisor to suggest creating appropriate indexes.

3.4 Policy Enforcer

Policy enforcer is responsible for checking policy conditions and taking action if needed. As mentioned before, policy enforcer is invoked both during bind time and during run time. During bind time, for the 'tuning' policies, the actions are usually enforcing query optimization hints, such as degree of parallelism, parallelism mode, join method, join sequence, table access method, etc. Policy enforcer evaluates the conditions, and decides if it should apply the optimization hints during query access plan generation. In the 'system config' policies, usually a set of bind parameters are specified. So the policy enforcer will pass to SQL compiler these bind parameters which take precedence over the global bind parameters. For instance, the global bind parameter specifies 'StarJoin=YES', while a 'system config' policy says 'StarJoin=NO'. During the lifespan of this policy, star join will be turned off during query optimization for the queries with the same scope as that policy. 'System config' component can tune other parameters such as sort heap size, etc.

In runtime, two, but not limited to, actions can be taken, namely 'report gen' and 'resource limit'. 'Report gen' is triggered by the monitoring activity and statement activity. If it is normal monitoring, resource stats are casted out when the statement leaves the cache. If it is exception monitoring, resource stats and operator stats are casted out when there is an exception observed. 'Report gen' component interacts with 'pushout manager' to actually generate a report. We will describe the pushout manager in more details in the next section.

'Resource limit' component is supposed to restrict the resource usage by each query, such as CPU time, etc. If such constraint is exceeded, the query execution can be terminated or paused.

3.5 Pushout Manager

Pushout manager generates detailed reports by populating three categories of tables: 1) a table captures resource statistics accumulated at statement level as well as at query level (STATEMENT_RUNTIME_INFO table), 2) a table contains operator statistics for the operators in a plan (OBJECT_RUNTIME_INFO table), and 3) several explain tables contain the detailed information of estimated statistics by optimizer, such as PLAN_TABLE, etc.

STATEMENT_RUNTIME_INFO table records the resource consumption monitored during query runtime as shown in Table 3. In addition, it also records the accumulated runtime statistics during the life span of the statement in the statement cache. We can classify its columns into four categories: 1) statement metadata, such as STMT_ID, the primary key of this table, statement text, authID, IP address, etc; 2) the accumulated runtime resource stats, including statement execution time, number of getpage(s), etc. The columns start with "STAT_" for statement; 3) the runtime resource stats at query level. They are the counterpart of columns in (2) and start with "QUERY_"; 4) exception information.

OBJECT_RUNTIME_INFO table is used to record the operator statistics as shown in Table 4. Here we only record the operators on table (including workfile for sort) and index, namely tablescan and indexscan, as these are the useful information to feed into statistics advisor and index advisor later. It has a set of estimated statistics for the row scanned, processed, the cardinality, filtering factor, etc. It also records the observed counterparts.

Explain tables contain detailed information for about access path

selection for a statement. There are PLAN_TABLE, DSN_STATEMENT_TABLE, DSN_FUNCTION_TABLE, etc. Please refer to [6] for more details about these tables. Each row in a PLAN_TABLE describes a step in the execution of a query or subquery in an explainable statement. The column values for the row identify, among other things, the query or subquery, the tables involved, and the method used to carry out the step.

Column name	Description
STMT_ID	Primary key
BIND_QUALIFIER	Bind qualifier
BIND_DEGREE	Parallelism degree for bind
BIND_ISOLATION	Isolation level for bind
STAT_ENUM	The number times this statement has been executed
STAT_GPAG	The accumulated number of getpage() operations
STAT_SYNR	The accumulated number of synchronous buffer reads
STAT_WRIT	The accumulated number of buffer writes
STAT_ROW	The accumulated number of rows processed
STAT_SORT	The accumulated number of sorts
STAT_INDXX	The accumulated number of index scans
STAT_TSCN	The accumulated number of table scans
STAT_PGRP	The accumulated number of parallel groups
STAT_ELAPSE	The accumulated elapsed time
STAT_CPU	The accumulated CPU time
STAT_SUS_SYNIO	The accumulated wait time for synchronous IO operations
STAT_SUS_LOCK	The accumulated wait time for lock and latch requests
QUERY_GPAG	The number of getpage() operations in this query
QUERY_SYNR	The number of synchronous buffer reads in this query
QUERY_WRIT	The number of buffer writes in this query
QUERY_ROW	The number of rows processed in this query
QUERY_SORT	The number of sorts in this query
QUERY_INDXX	The number of index scans in this query
QUERY_TSCN	The number of table scans in this query
QUERY_PGRP	The number of parallel groups in this query
QUERY_ELAPSE	The elapsed time of this query
QUERY_CPU	The CPU time of this query
QUERY_SUS_SYNIO	The wait time for synchronous IO operations of this query
QUERY_SUS_LOCK	The wait time for lock and latch requests of this query
EXCEPTION_COUNT	The number of exceptions occurred during the monitored period
STMT_TEXT	The text of the SQL statement
Timestamp	When this row is inserted

Table 3: STATEMENT_RUNTIME_INFO Table

In the POLICY_CONDITION_ACTION table, user can specify what kind of statistics information to cast out. For each of the above three categories of tables, there are low, medium and high granu-

Column name	Description
QUERY_ID	Primary key
OPERATOR_NO	Identifier of this operator
PLAN_NO	Identifier of this plan
SROW_EST	Estimated number of rows scanned by data manager
QROW_EST	Estimated number of rows qualified after applied predicates
COMPCD_EST	Estimated compound cardinality
IDX_FF_EST	Estimated filtering factor for index object
TBL_SROW	Number of rows scanned by data manager for table object
TBL_QROW	Number of rows qualified after applied predicates for table object
COMPCD	Observed compound cardinality
IDX_SROWS	Number of rows scanned for index object
IDX_QROWS	Number of rows qualified after applied predicates for index object
Timestamp	Insertion timestamp
...	...

Table 4: OBJECT_RUNTIME_INFO Table

larities that a user can specify during policy definition phase. That is only a subset of columns will be populated. For instance, for RUNTIME_INFO table,

- Low:
 - Cast out statement metadata, such as statement text, authID, IP address, etc.
 - Cast out number of executions.
- Medium:
 - Cast out columns in Low granularity
 - Cast out elapsed time, CPU time and IO statistics
- High:
 - Cast out all columns

As mentioned before, pushout manager is invoked once when the statement is purged from statement cache for normal monitoring. It is invoked each time when there is an exception detected by the system. Hence, there are two performance issues. First, if there are lots of exceptions occurring during execution of a query, lots of reports will be generated. Or if there are lots of queries with different statement texts running, lots of report will be generated for normal monitoring. Under these circumstances, it not only consumes too much CPU and IO resource, and may stall query execution as well. Hence, we design the pushout manager in the following way to address these problems. First, meta-policy is used to constrain the amount of records being populated in report tables for a given query. Second, the pushout process is asynchronously. In particular, we use producer/consumer model to transfer data from DB2 internal structure to a pushout queue, then to external tables. The details of these two approaches are described in the following paragraphs.

A meta-policy can be defined this way: “if the number of CPU time exceptions > 5, do not pushout report further”, or “if the total number of reports generated during this query life time > 10, do not pushout report further”, or “if the total number of reports generated during this statement life time > 1000, do not pushout report further”. So basically, users can cap the amount of reports generated per exception policy and/or per query and/or per statement. Internally, there are counters recording the total amount of reports per exceptions and/or per query and/or per statement. These counters are checked against the thresholds defined in the meta-policies

by policy executor. It further decides whether to invoke pushout manager or not.

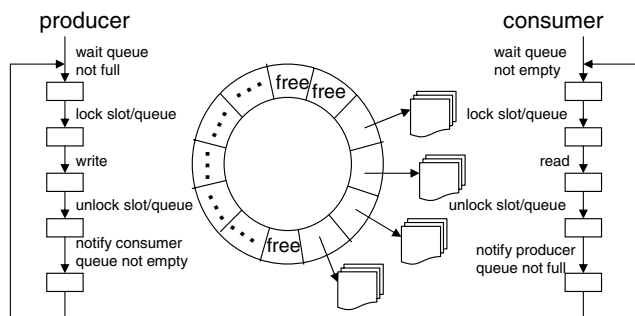


Figure 5: Pushout Model

Standard producer/consumer model is used to implement pushout manager report generation logic. A producer must wait until the buffer is not full, write report, and then notify the consumers that the queue is not empty. A consumer, on the other hand, must wait until the buffer is not empty, read a report item, and then notify the producers that the buffer is not full. Note that both producer and consumer can have multiple threads. Locks on a slot or queue is used to protect data in racing condition. So it guarantees mutex exclusion when the slots in the queue are accessed. The policy executor copies report data into a report pool, then the query can exit. During the pushout process, all the information get copied between producer and consumer are the pointers to the data in the report pool. Hence, it reduces a lot of overhead compared with copying the data from place to place. If producer generates reports much faster than consumer can consume, eventually the queue will be full. We cannot afford to have query wait until the queue has free slots. So if this case happens, the reports cannot get into the queue are simply discarded. However, a summary report will be generated to describe how many reports have been discarded due to queue full situation. It may indicate more threads are needed from consumer side, or IO may be the bottle neck.

4. APPLICATION SCENARIO

This section presents two main application scenarios where this policy-based framework is used. The examples illustrate the usage scenario at customer sites for query diagnostic and tuning. This framework is implemented as part of a toolset called Optimization Service Center for DB2 z/OS, or OSC for short. The term OSC will be used to refer to this framework throughout the rest of the paper.

4.1 Critical Applications

Problem detection and resolution in a mission critical application is vital for the business. For example, in a banking application, when query responses increase from 3 seconds to 5 minutes on a busy day, it would cause bank customers to pile up in the queue waiting for service. For on-line banking customers, the transaction time increases, resulting in decrease of the number of transaction serviced, until, ultimately, no-one can submit a request anymore. The direct result is a degradation of the quality of service, loss of revenue, and loss of business due to unhappy customers.

These kinds of problems take time to diagnose manually. In an actual case, it took 3 days to identify the problem query; clearly the impact to quality of service and business is more severe than what the customer can afford. The difficulty stems from the fact that in a typical enterprise application setting there are many applications running in the database system. Usually, it is difficult and

not clear how to start investigating this problem. In most cases, existing monitoring functions are not specific enough, not adaptive, or not efficient enough to locate the problem query. Most databases can not afford to leave the monitoring function active all the time for all applications due to its overhead.

The solution proposed in OSC provides a very affordable mechanism for capturing performance exceptions and collecting critical diagnostic information. The policy facility can speed up problem detection and resolution by several orders of magnitude. By defining a policy which effectively means:

"All queries for this application should not run longer than 10 seconds",

the database administrator can monitor queries under a given application for an exceptional performance deviation. In OSC, the function to specify is 'MONITOR CPU TIME', with the threshold set to 10 seconds, or 200% allowable increase.

Based on defined policies, OSC detects any imminent problem, generates an alert, and produces a report. By specifying an appropriate scope, the impact of this policy can be set to cover only a specific application, a group of queries, or a more limited scope. An example of a scope could be 'MyBanking' application, or any of the scope satisfies the description in Section 2. Due to its minimal overhead, the policy can be left active either continuously or most of the time as a safeguard mechanism. An exception is triggered when a particular query within the defined scope runs longer than the specified threshold, then a report containing relevant statistics is generated to aid the diagnostic process. This information can be feed-in directly to various analytic tools, like query advisor, index advisor, statistics advisor, and others analytic tools, which will suggest recommendations for resolving the problem. The database administrator analyzes these recommendations, evaluates the best course of action to take, and applies it after sufficient testing.

In this example case, the problem query runs with undetected sub-optimal performance of 3 seconds before the outage. Outage occurred when the execution time increased to 5 minutes. After tuning, the performance is improved to 0.1 seconds, a degree of magnitude better than before the outage. The process for identifying the problem query is reduced to almost instantaneously using OSC as opposed to 3 days with manual approach.

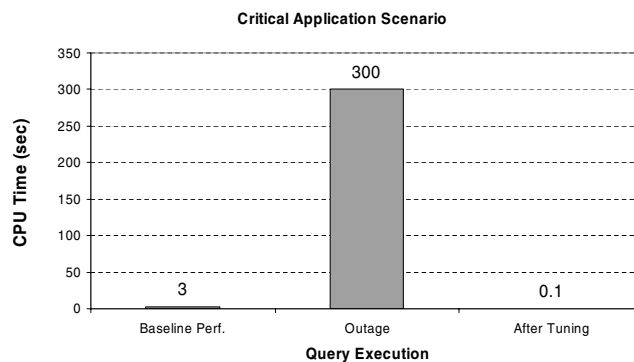


Figure 6: Importance of Exception Monitoring

Autonomic computing paradigm promises a self-repair concept, which closes the loop by implementing the recommended solution directly without human intervention. However, at this state, the technology is not mature enough to earn the trust for handling mission critical applications [17]. There is a fair possibility that the recommendations generated by the query advisor may yield worse performance; if so, it would be a disaster. In general, mission

critical applications usually have procedures in place for managing changes. That is, any proposed change must be analyzed and approved before it can be put in place. No changes will be put in place without sufficient testing and experimentation. This process is likely to remain until the technology matures.

4.2 Growth Tuning

This scenario applies to a database system which does not have any known outstanding performance problem. Such a system still needs a periodic health check to determine if the query portfolio has some performance degradation due to updates and other changes, or if it can be optimized further. Overtime, the performance of some queries in the system might have regressed into sub-optimal level undetected; therefore the chance for application outage has increased.

In this scenario, the task to be done is to capture the query workload and critical tuning information for each query to be used as a baseline for comparison. In some cases, existing general monitoring tools capture very limited tuning information, mostly only the statement text and the application identification information at the thread or the system level. The information is saved into persistent store at a fixed time interval. Tuning information, such as access path and runtime statistics are usually missing from the report, therefore they need to be obtained via a separate process whenever possible. This task demands a lot of time and effort, and thus would benefit from the application of an automated tool.

The solution provided by OSC automates the collection of tuning information and the baseline performance of application queries. The policies to set in place are for monitoring queries within the company's portfolio to detect conditions that are amenable for improvements. The information gathering is configurable selectively by specifying the relevant monitoring scope for queries under investigation. In this case, the policy to specify is to monitor queries in the workload, such as:

"Monitor queries in MyWorkload and report the baseline performance."

Using OSC, the function to specify is 'MONITOR', associated with the desired scope, like 'MyWorkload', or 'AUTHID=Proxy1' and 'IPADDR=9.30.22.122'. After the baseline information is established, new information collected from a subsequent monitoring measurement is compared with the baseline to discover any significant performance deviation. Queries with significant performance regression are subjected to tuning analysis with the help of query advisor, index advisor, statistics advisor, and other analytic tools. Other queries could also be investigated for potential performance improvement. Such an improvement could be, for example, to create a new index for queries with too many table scans, to eliminate unused index, to update stale statistics, and other conditions suitable for optimization. The resulting new improved performance is used as the new baseline for the next growth tuning or periodic health check.

This case study was experienced during a customers migration effort to a new release of DB2 for z/OS, which was an extreme example of tuning for growth. Due to lack of time and resources at the customer's site, queries were not tuned properly before the migration. After migration, performance regressions were detected in daily, weekly, and quarterly workloads involving hundreds of queries; each of them was having an acceptable performance prior to migration. This kind of problem would likely take about 5 person weeks of effort for tuning all workloads manually. Using OSC, the tuning effort was cut to only a few days. In addition, the performance results for both regressed and non-regressed queries were improved tremendously better than before the migration.

5. PERFORMANCE EVALUATION

The policy facility of OSC is required to be turned on explicitly to take effect. When it is turned off, there is no apparent overhead incurred as compared to the system without the policy facility.

The performance is evaluated using real customers query mixture on the following aspects: the overhead of policy facility turned on with no matching policy to enforce, performance of policy lookup, performance of normal monitoring versus exception monitoring, and the impact of the degrees of data collection and reporting.

5.1 Policy Enforcer Overhead

This case evaluates the impact of enabling the policy executor without having a query matching any of the defined policy entry. It gives the baseline overhead of having policy executor turned on without any matching policy to enforce.

The experiment results show that the overhead of policy executor is insignificant, < 0.1% CPU time overhead. The tests were conducted with 10, 50, and 250 threads running a mixture of 60 customers queries. Two different settings are used, No Match(1) is a setup with only one policy entry defined and no match, and No Match(1000) is a setup with 1000 entries with no matching policy to enforce. The results for both executions show negligible percentage of CPU time increase, barely affected by the number of defined policy entries.

5.2 Effectiveness of Policy Lookup

Policy lookup is done for each query prior to execution to see if the query matches any defined policy. Therefore it is important to ensure the optimal performance of this process. This step is executed as part of policy executor, which is already measured above with negligible percentage of CPU time increase. Consequently, the same conclusion as above can be derived here. That is, the overhead of policy lookup is negligible. The lookup is barely affected by the number of policy defined in the system. Figure 7 shows that the increase is somehow linear and almost flat, which suggests that the lookup would scale well as the number of policy grows.

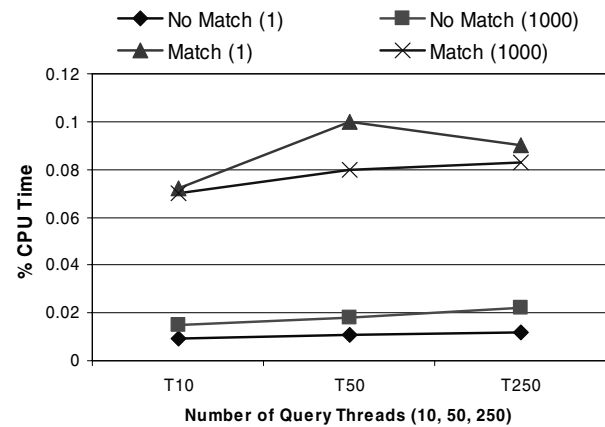


Figure 7: Effectiveness of Policy Lookup

5.3 Normal versus Exception Monitoring

This evaluation helps to understand the comparative overhead of normal versus exception monitoring, for example, CPU time or SPIKE exception. Monitoring normal query execution incurs 0.1% CPU time overhead with minimal report, whereas monitoring queries for CPU time exceptions show very low overhead of 0.5% with minimum report. All these numbers include the over-

head incurred by policy executor and enforcer, which are quite low. For comparison, the performance of comprehensive statement trace on the base code is shown in the graph below (Figure 8), which is about 5%.

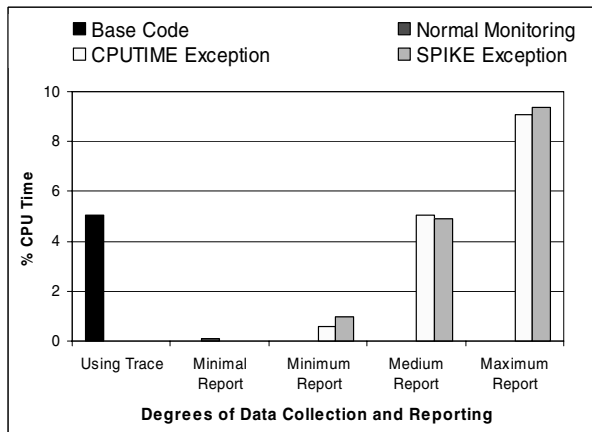


Figure 8: Monitoring Overhead

5.4 Degrees of Data Collection and Reporting

The amount of data collection and reporting has significant impacts on the performance. The graph above (Figure 8) shows that minimal data collection cost only 0.1% of CPU time. This overhead increases linearly when more data are collected and reported, that is, with minimum reporting < 1%, medium < 5%, and maximum data collection and reporting at around 9%. These results are consistent for CPU time and SPIKE exceptions.

5.5 Scalability

The scalability the implementation is mainly influenced by the performance of policy lookup and policy enforcer. Policy validation and conflict resolution are performed once when the policy facility is started, therefore they do not incur additional overhead during statement execution phase. Policy lookup is based on a quite efficient hashing function with stable performance as observed in Section 5.2. Lastly, policy enforcer dispatches the appropriate action via a bitmap with very minimal overhead. This analysis suggests that the performance would scale well beyond thousand policies.

Considering all the above evaluations, it can be concluded that overall performance overhead with minimal and minimum reporting is very low and is tolerable such that it would be affordable to turn the policy facility on all the time for diagnosing, tuning, and health check purposes. The overhead of medium and maximum reporting are relatively higher than the others, so they should be used sparingly. Potentially, these performance numbers can be improved further by optimizing and tuning the algorithm and by employing a specialized information processor engine, z9 Integrated Information Processor or zIIP [7].

6. RELATED WORK

Policy-based approach has been used for resolving and managing problems in various areas, both technical and non-technical ones. It should be noted, however, that the way policy is defined and interpreted varies among applications. Some examples of the area of usage includes commerce [16, 18], collaborations [15], automated provisioning [1], autonomic systems [4, 2], distributed sys-

tems [19], and other areas. Policies could be realized as a complex system, for example, policies for managing access control [14]. Beigi et al [3] studied the policy-based approach itself, as opposed to a specific application area on which policies were applied. The study describes how policies can be divided into different levels of abstraction and how to transform a higher level policy into a more concrete lower level one amenable for execution.

This paper proposes the innovative idea of using policy-based framework for managing database applications, with a particular focus on the query monitoring and tuning as the specific exploration area on which policies are applied. It demonstrates how a policy based approach can be used to resolve customers problems not only in query monitoring and tuning, but also in managing database applications in general. The policy engine implemented in this work is not only easy to use and affordable but also powerful enough to handle real customer problems. The concept of policy scope introduced in this work has been very useful for tailoring the problem solution either to be applicable globally or to designate it as a customized solution applicable only for a specific context.

In regards to query monitoring and tuning area, it is well known that most database vendors have tools for monitoring and tuning query performance. Query Monitor for DB2 z/OS [13, 11] provides a facility for monitoring queries with some filtering mechanisms to provide selectivity for the queries to monitor. Query Patroller tool in DB2 for Linux, Unix, and Workstation [12] has the capability to capture queries, to monitor them, and to set the execution priorities of the queries. Oracles SQL Tuning Advisor [5, 10] is described to provide an automated query tuning and monitoring facility. Microsofts SQL Profiler [8, 9] is documented to have a capability to monitor and record one or many users executing queries and collects configurable amount of information.

These performance tools more or less have comparable functionality; however, the main difference from the above tools is that this work uses a policy-based architecture implemented as a framework for hosting the features. Besides providing a clear architectural foundation, this framework gives additional advantages:

- *Openness*: Users can interrogate what policies are defined and active at any point of time. They can refine these policies or disable them completely.
- *Flexibility*: New policies can be added; existing policies can be refined or removed without adding or modifying the existing code.
- *Extensibility*: The framework is extensible; policies for other areas of applicability can be added later on without disrupting the existing ones.
- *Naturalness*: Declarative policies reflect the way people work in organization or as individual; therefore it would be easy to explain, to understand, and to use. By the nature of his work, a DBA may have some conventions or operational directives (or policies) dictated by the enterprise for administering and managing the database. The policy-based framework provides a mechanism for encoding those policies into their executable form.
- *Abstractions*: Policies can be expressed declaratively in a high level of abstraction, at the level of human communication, without describing too much detail. These policies can then be transformed, if necessary, into their low level equivalent with enough details to facilitate a programmatic execution [3].

The above list is not necessarily complete but it is suffice to illustrate the benefit and potential of the profile-based framework.

7. CONCLUSION AND FUTURE WORK

This paper presents a new approach for managing database application performance on DB2 for z/OS platform using a comprehensive tool called Optimization Service Center or OSC. The main contribution of this work is in the conception of a policy-based framework for resolving a multitude of performance problems faced by database administrators. The framework serves as the unifying approach for defining groups of policies, in which each policy group is applicable for a specific area; for example, query monitoring, tuning, system configuration, query prioritization, access control, and others. Specific descriptions and examples are given on a group of policies for query monitoring and tuning.

The framework is implemented as extension of the database engine, lightweight, and practical, with very minimal overhead, thus can be activated all the time for capturing performance exceptions, collecting baseline performance and critical diagnostic information. OSC is shown to provide the flexibility to define a new policy, to modify, disable, or remove an existing policy without requiring code changes.

The two scenarios described in this paper highlight our success stories. OSC is demonstrated to be very effective and practical for solving real customer problems. So far, the experience and feedback gathered from customers on OSC has been very positive. These encouraging results provide us with ample motivation to do more work with customers to explore and expand the use of policy-based approach for optimizing other areas of database operations with the goal of optimizing performance and reducing total cost of ownership.

8. ACKNOWLEDGEMENT

We thank the rest of DB2 for z/OS OSC Engine development team for their contributions in the implementation of this work. They are Ahmed Bashir, Chuck Bonner, Frank Bower, Chih Jieh Chang, Hsiuying Cheng, Yonghua Ding, Illong Huang, Dan Kozin, Fung Lee, Chan-hua Liu, Adrian Lobo, Bruce McAlister, Xiazhen Wang, Dannie Weis, Lori Williams, Li Xia, Fang Xing and Kou-Hong Yang. We also thank the anonymous reviewers for their comments and suggestions for improving this paper.

9. REFERENCES

- [1] K. Appleby, S. B. Calo, J. Giles, and K.-W. Lee. Policy-based automated provisioning. *IBM Systems Journal*, 43(1):121–135, 2004.
- [2] N. Badr, A. Taleb-Bendiab, and D. Reilly. Policy-based autonomic control service. In *POLICY'04: IEEE 5th International Workshop on Policies for Distributed Systems and Networks*, New York, USA, 7th - 9th June, 2004.
- [3] M. Beigi, S. B. Calo, and D. C. Verma. Policy Transformation Techniques in Policy-based Systems Management. In *POLICY'04: Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 13–22, 2004.
- [4] H. Boughton, P. Martin, W. Powley, and R. Horman. Workload Class Importance Policy in Autonomic Database Management Systems. In *POLICY '06: Proceedings of the Seventh IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 13–22, Washington, DC, USA, 2006.
- [5] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic SQL Tuning in Oracle 10g. In *VLDB'04: Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 1098–1109, 2004.
- [6] DB2 UDB for z/OS V8 Application Programming and SQL Guide, Feb 2006.
- [7] IBM Corp., IBM System z9 Integrated Information Processor (IBM zIIP), <http://www-03.ibm.com/systems/z/ziip>.
- [8] Microsoft Corp., Microsoft SQL Server 7.0 Performance Tuning Guide, MSDN Library.
- [9] Microsoft Corp., SQL Server Profiler, MSDN Library, <http://msdn2.microsoft.com/en-us/library/ms173757.aspx>.
- [10] Oracle Corp., The Self-Managing Database: Guided Application & SQL Tuning with Oracle Database 10g Release 2, An Oracle White Paper, June 2006.
- [11] IBM Corp. DB2 Query Monitor for z/OS: Users Guide, Dec 2005.
- [12] IBM Corp. DB2 Query Patroller Guide: Installation, Administration, and Usage, Dec 2005.
- [13] Bruni, P., et al. DB2 for z/OS and OS/390: Tools for Performance Management. IBM RedBook, Nov 2001.
- [14] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26(2):214–260, 2001.
- [15] M. Lakhidissi, H. Harroud, A. Karmouch, and C. Grossner. A policy management system for mobile agent-based services. In *MATA'01: Proceedings of Management of Multimedia on the Internet : 4th IFIP/IEEE International Conference on Management of Multimedia Networks and Services*, pages 114–123, 2001.
- [16] S. Lamparter, A. Ankolekar, R. Studer, D. Oberle, and C. Weinhardt. A policy framework for trading configurable goods and services in open electronic markets. In *ICEC '06: Proceedings of the 8th international conference on Electronic commerce*, pages 162–173, 2006.
- [17] V. Markl, G. M. Lohman, and V. Raman. LEO: An autonomic query optimizer for DB2. *IBM Syst. J.*, 42(1):98–106, 2003.
- [18] N. H. Minsky and V. Ungureanu. A mechanism for establishing policies for electronic commerce. In *International Conference on Distributed Computing Systems*, pages 322–331, 1998.
- [19] M. Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2:333–360, 1994.