Declarative Management in Microsoft SQL Server

Hongfei Guo Dan Jones Jennifer Beckmann Praveen Seshadri
Microsoft, One Microsoft Way, Redmond WA, USA
{hongfeig, dtjones, jennbeck, pravse}@microsoft.com

ABSTRACT

This paper describes the principles and practice of Declarative Management — a new approach to the management of database systems. The standard approach to database systems management involves a brittle coupling of interactive operations and procedural scripts. Such ad hoc approach results in incorrect administration, which leads to increased management costs. In the Declarative Management paradigm, a user specifies "what" the desired state is, and the system figures out "how" to get there and stay there. Declarative Management represents a fundamental step towards the goal of a self-managing database system. It also has the potential to significantly lower both administrative error and cost. An initial implementation of Declarative Management has been released with the Microsoft SQL Server 2008 database product, and the paper covers the implementation design as well.

1. INTRODUCTION

Database systems manage and process data efficiently, reliably, and at scale. For common database applications, many commercially available database products provide the necessary features at satisfactory performance. However the complexity and cost of *management* of the database systems themselves has increasingly become the primary differentiator among the different products. The focus of this paper is a new approach to database systems management that drastically lowers cost and complexity, while retaining flexibility. In fact, this approach is a promising step towards an eventual goal of self-management, which is particularly important in the context of the accelerating shift towards cloud-based server computing.

1.1 Problem Description

Database systems are managed by database administrators (DBAs). Typically, DBAs use a combination of five broad approaches to system management:

- Graphical interactive management tools that allow the DBA to browse metadata, examine configuration, and take operational actions.
- Operational command-line scripts that capture one or more DBA actions.
- Automation of scheduled (often recurring) or event-driven operational scripts for system maintenance.
- Monitoring via real-time and historical monitoring tools.
- Trouble-shooting diagnostics and tuning via "advisors".

The current management technologies and tools are both DBA-intensive as well as prone to DBA-error. These technologies are severely limiting in today's IT environments because of the increasing number of database deployments (hundreds per DBA), the complexity of the database systems (database systems are among the most complex of server products), and the dynamic changes that occur constantly and need to be reacted to (new data, new applications, new management requirements). While hardware costs have fallen, the relative cost of skilled DBA time and attention has gone up steeply. With greater multi-tasking demanded of the DBA, the probability of human-error increases and with it come significant costs (for example, if there is an error in the recurrence schedule of a backup script, important data may not be recoverable).

New technology is needed that drastically increases DBA productivity and correctness, while still retaining and exposing the rich management options and capabilities of the underlying database system. The other important requirement is that the management technology must be simple to understand and use. DBAs are typically not fans of complexity (for example, a hypothetical solution that involved having the DBA write Prolog programs would fail the real-world simplicity-test even if it theoretically solved the problem). Indeed, simplicity and transparency are primary design imperatives.

The other important observation is that we are not focused on tasks of great complexity (for example, performing deep security threat analysis on a database system). The objective is to increase productivity and reduce errors for tasks that are, in isolation, quite routine for the skilled DBA, yet have to be done repetitively and interactively, and with the potential for error.

Finally, the next decade will see a significant shift towards server computing "in the cloud" [3]. One of the most viable and currently popular approaches to cloud computing is to host existing software services (operating systems, databases, etc) on cloud-hosted infrastructure [2]. In such an environment, machines and software services are provisioned automatically and must be managed without human intervention. Managing database services in such an environment also requires a change in approach — the traditional DBA techniques clearly do not suffice.

1.2 Contribution

Declarative Management represents a uniquely database-inspired approach to systems management — it applies the declarative tradition of database query processing to management. Instead of specifying a procedural script of administrative actions for each task, the DBA specifies a desired outcome at an appropriate high level of abstraction. This is the DBA's declarative intent.

Since the intent is specified in terms of the outcome, and at a higher level of abstraction than the implementation details of the underlying system, management is less prone to human error. The Declarative Management technology interprets the intent, applies it in a scalable way to all systems being managed, and ensures that the intent is continuously monitored and enforced.

Below are some intent examples that a DBA may want to enforce. We refer to these examples throughout this paper:

Example 1: For all columns in all tables in the Payroll database, if a column contains SSN, then the column needs to be encrypted.

Example 2: For all databases more than 20MB in size, statistics collection should be turned on.

Example 3: For all tables, views and stored procedures, their names should be at least 5 characters long.

Example 4: For all HR databases, they should have a data loss window of 15 minutes, and can be recovered within 1 business day.

Take Example 1 for instance (described in detail along with core concepts in Section 3), a DBA mandates that all columns with social security numbers (SSNs) should be encrypted. Under the Declarative Management paradigm, the intent is captured by a condition: *[If Column.Type = 'SSN', then Encrypted = 'true']*. This intent can be configured to apply to all or a subset of database services in the IT environment, and, within each server, to all or a subset of tables. Further, this intent can be configured to be automatically checked periodically or whenever any schema change occurs in the target databases. Having specified this intent, the DBA no longer needs to manually and interactively manage the system for this purpose — rather, the system is smart enough to understand the desired intent, and to apply it automatically across multiple tables in multiple databases.

We proposed a simple yet practical model for declarative management and designed a framework that supports this model. In the Microsoft SQL Server 2008 commercial database system, we implemented this framework and integrated it with the rest of the product.

The integration itself poses practical challenges — DBAs are typically unwilling to use a plethora of one-off management tools for different problems — rather they want an integrated management tool that includes a variety of related management solutions. Furthermore, since SQL Server itself is not a new product, DBAs adopting the new management technologies in SQL Server 2008 expect to leverage and extend many of the existing technologies they have already learned and adopted. These pragmatic constraints of the commercial database market, associated with the inherent technical hurdles of the problem space have made this work particularly challenging.

The paper is organized as follows. Section 2 sets up the background and discuss related work. Section 3 presents our model for declarative management. Section 4 describes the overall design, architecture and implementation. Section 5 addresses policy automation. Finally, section 6 concludes the paper and points out future directions.

2. BACKGROUND

Since the topic of database systems management has not received as much academic attention as other aspects of database systems (like transaction processing and query processing), this paper provides some introductory background and problem description.

2.1 Total Cost of Ownership

It has been four decades since the invention of relational database system, and today most commercial relational database systems have an adequate feature set for the needs of most customer applications. Relational database systems are at natural stage of technology maturation. Cost, not feature set, has become a primary deciding factor in the choice of one database system over another. The cost of adoption of a database system is measured not just in license cost, but also many other components such as hardware cost, the developer time taken to build a new application (time to solution) where the application is not purchased, and the cost of managing the application and system over its entire lifecycle. Total Cost of Ownership (TCO) is the aggregation of all these costs. The hardware and software license costs have traditionally been factored into database performance benchmarks (for example: the various TPC benchmarks have price/performance metrics) acknowledging the important role played by cost in technology adoption. However, hardware and software costs, while very tangible at initial adoption, are an increasingly small fraction of TCO [23]. The dominant factor is becoming the aggregated cost of management, for which we coin the phrase "Total Cost of Administration" (TCA). Techniques and technologies to reduce TCA are a relatively new but very commercially important area for research and innovation.

This cost is partially the human cost of the database administrator (DBA) who deploys and manages the applications and the underlying database system. A DBA is an expensive employee and a skilled one is difficult to hire and retain, and database systems have not yet provided the right technology to make them productive and efficient. A more significant cost is due to lack of management (because there are too many systems for the few DBAs to manage) or due to human error by an over-worked DBA. Large IT environments also see constant change — existing database applications grow in data size and distribution, are moved to new hardware, and are upgraded to new versions. New applications are often deployed. In this dynamic environment, the DBA is often playing catch-up rather than guaranteeing management correctness and consistency as desired.

Three recent trends have exacerbated these problems: (a) growth in system complexity, (b) increase in the number of databases each DBA is expected to manage, (c) increase in the heterogeneity of the environment being managed (different geographic locations, database system versions, applications, storage systems, etc).

There has been some research over the last two decades on the impact of human errors. Gray did early work on understanding system faults [15][16] and estimated that administrator mistakes account for 15% of system outages. Gray showed that highly available systems can mask some administrator error with redundancy or fault tolerance; however, Brown and Patterson [7] argue that system dependability continues to be a huge factor in system uptime because all mistakes cannot be prevented by high availability alone. DBA tasks and mistakes are categorized by Gil et al. [14], and that work was extended by Vieira and Madeira [30] to measuring the dependability of OLTP systems under faults by employing a benchmark that injects errors into a workload.

2.2 Autonomous and Auto-Admin Databases

One approach that has been explored in prior work is to have an opaque self-managed system (sometimes called "autonomous"). In the case of some focused server technologies, there is a trend towards self-managing appliances that have very few configuration settings available to the user (for example, "gizmo databases" [5][29]). While this is a viable approach for a specific server solution (for example, a print server), it isn't suitable to general-purpose database systems. Modern database systems are comprehensive application platforms. They offer users a veritable Swiss-army-knife of data-centric technologies. There are multiple physical storage options, access paths, configuration settings, query behaviors, recovery settings, etc. The same database system software is expected to support a variety of different applications in different resource environments with different performance requirements. Each DBA expects to be able to control the behavior of the database system in these different dimensions — flexibility of management intent is essential.

Brown et al. [6] propose goal-oriented buffer management, where users set workload response time goals and the system automatically adjusts the buffer memory allocations to achieve those goals. Such approach provides algorithms for declaratively managing specific system aspects, and can be readily plugged in the declarative management platform we describe.

There has been much work on auto-tuning, Chaudhuri and Narasayya [9] provide a good survey. Some research simplifies systems enough to enable automatic tuning [10][17][31]. And some techniques have made it into commercial systems by aiding DBAs with performance, such as index selection or physical partitioning [1][8][26][27][28]. This work is strictly complementary to Declarative Management. Auto-tuning advisors typically act as advisors to human beings, and are focused on complex management problems. Managing across a large number of servers isn't the primary focus of these efforts. In contrast, Declarative Management is focused on simpler daily management tasks, and the desire to make these correct and scalable across many servers being managed.

2.3 Systems Management Technologies

Since operator errors are so common in large system administration, there are a few approaches for helping administrators identify such errors before they are made on live production systems. Oliveira et al. [24] survey DBA test environments and propose validation techniques to help administrators identify errors before they affect a production system. Galanis et al. [13] introduce Oracle's Database Replay which allows DBAs to identify potential errors by allowing them to run and record planned changes on test system and then replay those changes on real production system workload.

In the broader domain of operational systems management, there are multiple products that provide monitoring and management capability for data centers and IT environments. Commercial monitoring application suites, such as OPNET Panorama [25], HP Business Service Management (formerly known as the OpenView suite) [18], Microsoft System Center [21], and IBM Tivoli [19], help administrators monitor key health indicators and alert them to unhealthy symptoms. That body of work is complementary to our approach: The goal of those products is the health state of the whole IT stack, not a particular layer. In contrast, our approach is tailored for the database layer; hence it fully leverages the domain knowledge and capacity of the database system.

In recent years, there has been a trend towards model-driven management [11][22]. Management models capture the topology of the environment and the metadata of the systems being managed. Management intent is specified against the model, and the management infrastructure utilizes that intent to better monitor and manage the systems. Recently, there have been efforts to standardize the definition of management models as well as management intent, such as SML [4]. The work in this paper reflects the same spirit as these initiatives, yet the fact that it is built on and for a database system enables natural and deep integration of declarative intent.

3. BASIC CONCEPTS OF DECLARATIVE MANAGEMENT

There are six core concepts related to Declarative Management. (a) Health conditions, (b) target binding, and (c) automation modes are utilized to build the basic unit of intent — (d) a management policy. All of these concepts utilize (e) management models, which together with (f) facets, encapsulate domain knowledge of the system to be managed.

In the rest of the section, we first describe management models, which lay foundation for declarative management, followed by health conditions, target bindings and automation mode. We will then summarize policies based on that. Finally we will discuss the more advanced concepts: facets and the OnChange – Impose automation mode.

3.1 Management Models

Management models are not a new concept, nor are they built only for Declarative Management. All modern databases define management models (typically represented via object models) as an abstraction to build management tools. However, since they act as a foundation for the concepts in this paper, we will describe them briefly.

A management model represents the types of the management entities in the system. For example, a relational database management model might define *Table, View, Stored Procedure, Trigger*, etc. as types in its management model. Each of these types in the model defines properties as well as relationships to other types[1]. The individual properties and relationships have a variety of attributes that indicate management behavior, for example, some properties are changeable whereas some are read-only.

Management targets are instantiations of management model entities in a real system being managed. For example, a particular table T1 is a management target of type *Table*. The values of model properties and relationship constitute the complete observable state of the management target, at least from the viewpoint of system management.

It is important to explain the role of the management model in the management software infrastructure. For the management model to be usable, it needs to provide two core capabilities:

[1] A complete management model also describes operations and their effects, but those details are unnecessary for the concepts in this paper.

- An instance of the model (a collection of management targets) can be populated from a database system being managed, reflecting the metadata and state of the system.
- Certain changes to the model (for example, a configuration change) can be propagated faithfully to the database system being managed. Properties that can be changed in this fashion are attributed appropriately.

The model itself provides a "logical" abstraction of the system for the DBA to manage. The developers of the database system (in our implementation, the SQL Server product engineers) provide the model implementation. It is important for the system engineers to define the model, rather than the DBA, because the development of models is complex and requires deep knowledge of the underlying system, and ideally there should be one reference model. The model implementation can also hide differences between different major and minor versions of the same database product (for example, between SQL Server 2000 and SQL Server 2005, there are significant differences in the representation of metadata in the system catalogs, but this is handled transparently by the management model).

One of the important tasks of a DBA is to be able to browse the metadata of the system. Systems have grown large in the number of artifacts being managed (tens of thousands of tables in an application are not uncommon, and a DBA might own hundreds of applications). Consequently, rich management model implementations also provide the ability to search and query the model.

3.2 Health Conditions

The DBAs interact with the database system by specifying their management intent via a Health Condition. It is declarative in nature and is a description of a healthy system. It specifies a Boolean constraint over the state of the management target.

The semantics of Declarative Management are straightforward: If the Boolean constraint is satisfied, the target is in conformance with the Health Condition. If not, the state of the target needs to be changed to bring it to conformance.

Consider the desire to ensure that automatic statistics collection is turned on for a particular database, as in Example 2, a common DBA task. This feature typically adds some overhead to normal execution, but enables significantly better query optimization. In some highly tuned database applications without ad hoc queries, the extra overhead might be undesirable and without value. This is one of a large number of available configuration settings exposed by a modern database system. Every large-scale IT department enforces standardization norms for the systems that they deploy. Requiring automatic statistics collection is one such norm.

The management model defines a *Database* entity type with many properties including one, *AutoStatistics*, which is either true or false. The Health Condition is specified by the Boolean expression: *Database.AutoStatistics = True*. Since the model populates values from the underlying system, Declarative Management checks the Health Condition for conformance of a specific database.

It is not so obvious that the same Health Condition can be used to "impose" the intent — i.e. change the database to conform to the intent. In this simple example, it is easy to see that if a database fails to conform to the Health Condition, it could be made to conform by altering the *AutoStatistics* property to a value of "true" and propagating the change from the model to the underlying system via whatever syntax is used in the underlying system – for example, for SQL Server 2008, this involves using the "SET" command of the Transact-SQL language. The model implementation encapsulates the details of command syntax, and the DBA does not need to be aware of it. We will discuss richer aspects of imposing intent in a later subsection.

3.3 Target Binding

As described above, a DBA can pick individual health conditions and check them against a specific management target. This is simple but does not scale, either for DBA-driven checks, or for automated checks. An important concept therefore is the binding of a health condition to a set of targets. In keeping with declarative design principles, the set of targets is defined by a query over the management model.

In Example 2, the health condition should only be applied to all databases that are larger than 20 MB in size. The target binding is captured by a query expressed in pseudo-syntax as: *Databases [Size > 20MB]*. Notice that this is simply a Boolean condition acting as a filtering predicate over all databases being managed. Indeed, the query expressive power needed for model queries is only the ability to filter all instances of a particular type. Richer query constructs (projection, aggregation, joins) are not essential for basic management. In a later section, we will discuss the issue of query richness and the advanced capabilities it enables.

An interesting aspect of this design is that there is no fundamental difference between Health Conditions and the filters used in the target binding queries. For example, *Databases [AutoStatistics = True]* could represent a set of target databases that have auto statistics set to true, or a Health Condition specifying that a healthy database as one that has auto statistics set to true.

The query-based approach to target binding is crucial for reacting to change, especially when declarative management intent is applied via automation. Database systems are dynamic — in the example above, a database that is tiny one day may grow in size past 20MB or a new database might be deployed after the management intent is defined. By specifying a query-based target set, the set of targets is chosen dynamically depending on the state of the targets at the time of evaluation, rather than the state of targets at the time of definition.

3.4 Automation Modes

If a system is to be self-managing, the application of management intent needs to be automated. All mature database systems provide management automation capabilities. Described simply, these allow procedural tasks (scripts or programs) specified by the DBA to be run when particular events occur. The events could be driven by a recurring schedule or could be ad-hoc driven by changes in the system being managed. Management automation tasks typically run as background activities without interfering with the actual application workload of the database system.

Before we discuss automation modes for intent, we require a brief description of metadata event support in database systems. Database systems expose the notion of synchronous change events – events that fire in the context of a transaction making a change. Users of the SQL language are familiar with the concept of triggers, which execute transactional logic when data is changed. Likewise, advanced database systems support metadata triggers that activate when metadata definitions change (for example, if a table is renamed, or if a key is added). Advanced database systems also support a number of asynchronous events corresponding to operational changes that may not be transactional (for example, events raised when a database is restored, or when a new user session connects).

Assuming this support for change eventing, there are three aspects of the automation mode for management intent:

- When is it evaluated? It could be on a recurring schedule, or it could be when specific change events occur. In particular, the events of interest are change events on management targets defined in the target set.
- In what execution context is it evaluated? There are therefore two interesting execution contexts for management intent: Synchronous (when the underlying system does provide such events), or Asynchronous.
- What happens on failure to conform[2]? Interesting options on failure are (a) Impose intent (described in Section 3.8), (b) Rollback change, and (c) Other. In a practical system implementation, other options must exist, including logging failure at different levels, sending notifications to system operators, running procedural scripts, etc. However, they do not alter the essential declarative concepts, and we will capture the entire class by a single failure option in (c).

Not all combinations are sensible. A Rollback failure action makes sense only for OnChange events with a synchronous context (in all other cases, the transaction has committed before the policy is executed, and while we could develop a whole model of compensating actions to rollback committed transactions, this is not compelling and violates the simplicity design imperative). Conversely, for policies evaluated OnChange in Synchronous execution context, most of the failure reactions in (c) are not applicable because typically such actions (for example, sending an email to a system operator) are not supported within transaction boundary. This leads to the grid of combinations below:

Event Model	**Execution Context**	**Failure Reaction**
OnSchedule	Asynchronous	Impose / Other
OnChange	Asynchronous	Impose / Other
OnChange	Synchronous	Rollback
OnChange	Synchronous	Impose

Extending Example 2, a DBA can specify that every 24 hours, the Health Condition requiring automated statistics should be checked against all databases whose size is larger than 20 MB. Note that the schedule is also specified in a declarative fashion. Instead of specifying a particular time to evaluate it (for example, at 2 AM every night), it only specifies the recurrence duration. This allows the very same intent to be applied to two different systems, perhaps in two different time zones, and perhaps with two different load schedules. The maintenance window for one might be between 1 AM and 4 AM, and the maintenance window for the other might be between 10 PM and 2 AM. A declarative schedule allows the choice to be made locally.

[2] Note that the presumption is that successful conformance means that the system is healthy and there is not much else to do other than log this information for diagnostic purposes.

3.5 Policy

To summarize, management models define management entity types. Health conditions define desired system state by expressing Boolean conditions over the properties of a management entity type. A target binding is a query that defines a set of management targets.

A policy represents the discrete unit of declarative management intent, including what to check (the health condition), when to check (the automation mode) and where to check (the target binding).

3.6 Management Facet

It may be helpful to consider an analogy with programming languages, with the context that we are designing a "declarative programming" environment for DBAs to define management intent "programs". In programming languages, type systems enable polymorphism and abstraction via interfaces. By analogy, management models provide a type system for management intent, and they enable polymorphism and abstraction via management facets. As shown in Figure 1, system developers encapsulate system complexity into facets, and DBAs define conditions at higher level on top of facets.

A management facet is a group of (related) management model properties that represent a "facet" of management. Each of the properties is attributed to indicate if it is read-only or if it is read-write. Management entity types are default facets. In addition, in the management model, "abstract" facets can be defined as interfaces, which then can be chosen to be implemented be the entity types. Health Conditions are authored against management facets.

Facets enable four different capabilities essential for lower TCA: (a) simplification, (b) abstraction of system complexity, (c) abstraction of change complexity, (d) polymorphism.

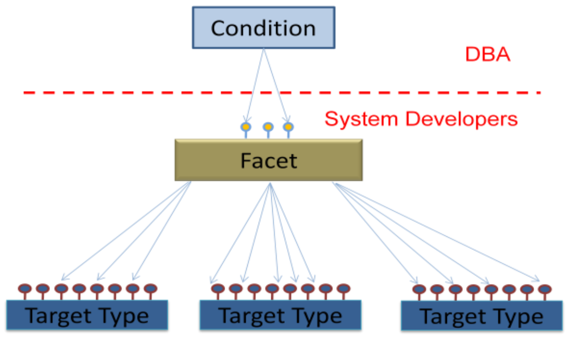

Figure 1: Facet Abstraction

3.6.1 *Eliminating Unnecessary Complexity*

Some management entity types have a large number of properties. For example, there are over 50 configuration properties of a Database type in SQL Server 2008, some of which define language semantics, some of which define storage preferences, some of which define performance hints, etc. When defining intent about language semantics, the rest of the properties are confusing and irrelevant. Organizing just those properties into a single facet simplifies the definition of intent and reduces errors.

3.6.2 Abstracting System Complexity

In today's commercial database systems, DBA's management intents are buried in the complex settings, making it impossible to reason. Facets provide a mechanism to abstract the complexity, allowing DBAs to specify the intents at higher level.

For example, one of the top routine maintenance tasks DBAs spend most time on is the backup task. There are many aspects in planning a backup strategy, recovery model, backup type, backup frequency, backup verification, backup devices, backup retention, and the list goes on. Often DBAs would setup a certain backup strategy. However, when disaster happens, there is no guarantee their system can recover satisfactorily.

Using facet mechanism, a data recoverability facet might expose properties in terms of recoverability goals and constraints, for example: what is the data loss window? Whether point of failure recovery is needed? Whether point of time recovery is needed? What is the acceptable down time? What is the optimization goal (minimize backup space, minimize restoration, or minimize performance penalty, etc.). Now DBAs can specify their intent along those dimensions, and the system will use heuristics to generate the backup strategy and will be able to automatically monitor it and verify it.

3.6.3 Abstracting Change Complexity

Upon policy violation, a desirable reaction is to change the system to conform (for details and examples see Section 3.7). Facets not only encapsulate the logic for property "check", but also encapsulate the logic for "change". For example, in order to allow remote access to a SQL Server instance, there is a long list of steps [20]. With the facet mechanism, we expose a Boolean configuration property "AllowRemoteAccess" to DBA while hiding the tedious steps inside of the facet implementation.

Further, characteristics of the properties w.r.t. change are also captured in facets. Some facet properties are read-only, for example, database size. Some properties are read/write, however, only certain change is deterministic, for example, there is a deterministic way to set *SupportsANSI99Semantics* True (see Section 3.7), however, there is no deterministic way to set it False. Some properties can always be changed deterministically.

3.6.4 Polymorphic Management Intent

Multiple management entity types can support the same management facet, enabling polymorphism, as shown in Figure 1. A direct benefit is that DBAs can define one policy, and apply it to multiple entity types. Naming convention is a typical usage. As in Example 3, a company requires that the names of all tables, views and stored procedures in the databases created by certain app contain more than 5 characters, i.e., [length (Name) > 5]. To support that, we can have a facet which contains the Name property, and this facet is implemented by all target types that support name. Now DBA only needs to define one policy with target binding only to views, tables and stored procedures.

3.7 Imposing Conformance via a Policy

A policy can check conformance as well as impose conformance — i.e. to change or configure the system to conform to the policy. However as explained below, this is not always possible — it depends on the policy, on the management model, and on the state of the system.

Consider a Health Condition expressed against the properties of a management facet on a target entity type. For an arbitrary Boolean expression containing disjunctions or inequalities, there are multiple ways of changing the property values so as to satisfy the expression. Finding a combination of property values that satisfies an arbitrary Boolean expression is also not a tractable problem. In a management system, transparency and determinism are important. We need to avoid non-deterministic outcomes. This requires restrictions on the form of the Boolean expressions that can be used to impose intent. Further, there are correlations between behaviors of a complex system like SQL Server 2008. Changing one property may have the unexpected effect of altering another property because of interactions beyond the knowledge of the management model (for example, a customer could implement logic on the server that changes one configuration parameter based on the value of another). These are the practical complexities of a real system, yet the value of imposing policy conformance is significant enough that we need to find workable approaches to common-case scenarios.

The problem can be simplified by considering the Boolean expression in conjunctive form (subexpression1 AND subexpression2 AND ….) and considering it with the context of a specific evaluation (against a specific system). If the expression evaluation succeeds, there is no need to impose the policy – it is already in conformance. If the expression evaluation fails, certain subexpressions will fail, but not necessarily all of them. We follow the principle of least surprise and only impose changes corresponding to subexpressions that fail.

For the moment, let us consider a simple sub-expression of the form: *[Property1 = value] (*this kind of expression is actually very common in configuration management). If the management model allows *Property1* to be changed, then imposing the policy appears simple — set *Property1* to *value* and commit the changed model. However, this does not necessarily always succeed, for the following reasons:

- There may be a validation failure while setting the property value.
- The facet implementation may disallow setting certain values. This is particularly true for complex facets. For example, a Boolean configuration property for language semantics may be SupportsANSI99Semantics. In the underlying system, there may be two different configuration parameters, one to enable ANSINullSemantics and one to enable ANSIDefaultSemantics, both of which need to be enabled for the composite configuration property to be true. In other words, there is no deterministic way to make the expression false, but there is one for making it true. There may be a runtime error when trying to commit the changes.
- The changes may have side-effects that invalidate other subexpressions in the Boolean expression. The simplest case of cause is if the same property is used directly in another subexpression, but in complex cases, there may be interactions in the underlying system that could cause unanticipated correlation between properties.

For all of these reasons, the process of imposing conformance must have three distinct steps:

- The condition must be evaluated initially to identify subexpressions that fail.
- Changes to the model must be committed within a transaction.

- After the change, in the context of the same transaction, the condition should be reevaluated to ensure conformance, and if not the transaction should be aborted reverting all change.

So far, we have only discussed imposing a condition. Since a policy also specifies a target set, the semantics of imposing a policy is equivalent to imposing the condition on each of the targets in the target set. Because of the possibility of failure though, a particular system implementation needs to decide on the atomicity semantics of such a collective operation.

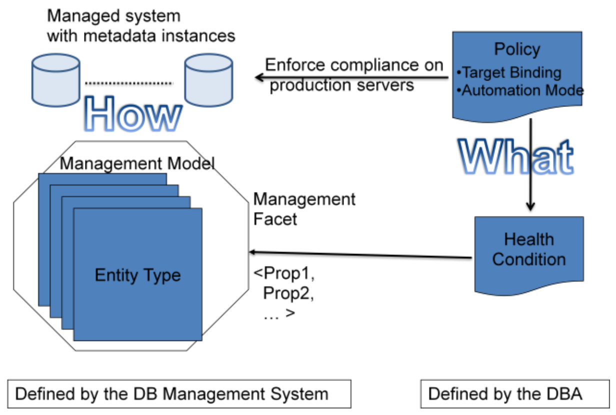

Figure 2: Basic Concepts

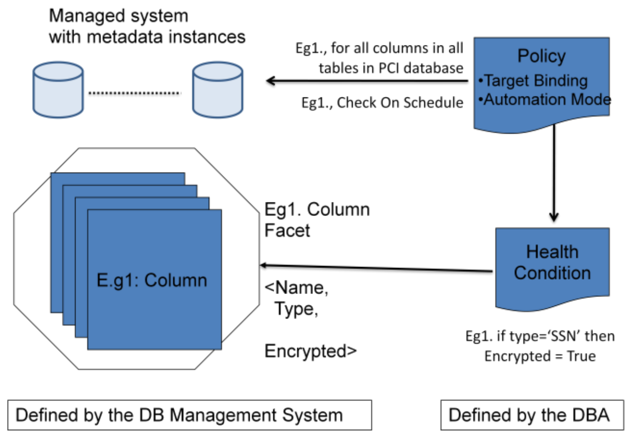

Figure 3: Example 1 Mapped to Basic Concepts

3.8 Summary

Figure 2 shows the relationship of the concepts. A policy is the discrete unit of intent, defining a "what" outcome. It combines a health condition, a target binding and an automation mode. DBAs interact with the database system by specifying policies. The system is automatically managed and monitored accordingly.

The management model and facets encapsulate the domain knowledge of the system to be managed (that is, "how" to make it so), raising the level of abstraction for DBAs. Figure 3 shows how Example 1 mapped to the basic concepts.

Declarative management is a key to addressing the challenges described in Section 2.1. Management models and facets encapsulate system complexity, allowing DBA to declare intent at higher level. This leads to simplified DBA tasks hence fewer errors. Once DBAs specify policies, the system can be monitored automatically according to the policies. Some policies can be used to prevent violating changes, reducing human error; others can be used to detect violation automatically. Such automation capacity enables DBAs to manage by exception, hence scale the number of databases one DBA can manage. Further, polices can be specified at a central location and pushed out to manage all servers. This capacity allows management at scale as well as management of heterogeneity in the environment.

4. DESIGN AND IMPLEMENTATION

This section describes the design principles, user interaction model and high level architecture, followed by implementation of the concepts described in the previous section.

4.1 Overview and Principles

Microsoft SQL Server 2008 has an initial version of Declarative Management under the name "Policy-Based Management" (PBM). As mentioned in the introduction, it is non-trivial to introduce a new management paradigm into a complex product with (a) a large existing customer base and (b) existing management practices. Our goal was to have positive impact on TCA with a large number of existing customers, and this required us to embrace and extend existing familiar technologies wherever possible.

While Declarative Management can apply to all aspects of database management, this initial implementation was scoped to focus on schema management and configuration. The new capabilities are exposed in two fundamentally different settings:

- Ad-hoc policy evaluation within a management tool (described in this section).
- Automated policies (described in Section 5).

In order to round-out a commercial database product feature, the declarative management capabilities were integrated in the management tools and solutions. This includes designer tools to author policies, serialization formats, libraries of pre-defined policies for easy customer adoption, etc. In the interest of brevity in this paper, we ignore those important aspects of the product and focus instead on the key technical underpinnings.

4.2 User Interaction Model

Figure 4 represent the user interaction model under the declarative management paradigm. DBAs specify intent by authoring policies. They can then evaluate system health state ad-hoc, or automate the policies. Policy evaluation results are logged. System Health report is generated based on the evaluation history. Once a DBA specifies the automation of the policies, they manage the system by exception. That is, DBAs only need to attend the system when policy violation happens. DBAs can leverage the ad-hoc evaluation for testing policies or for diagonosis purposes.

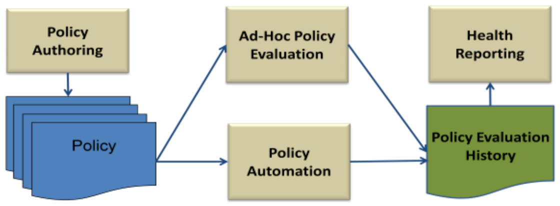

Figure 4: Declarative Management Paradigm

4.3 System Architecture

The system architecture of PBM is shown in Figure 5. Policies and metadata are stored in a management database (called *msdb*) (see Section 5). Outside of the SQL Server Engine, SMO (SQL Management Objects) is an implementation of the management model (see Section 4.4). The policy engine provides the core functionality of PBM, policy evaluation. It interacts with SMO through the facet interface. The policy engine uses a command line interface implemented in Windows PowerShell.

The rest of the figure presents how policy automation is implemented (see Section 5). OnSchedule is done through SQL Agent Job services, which provides scheduled execution of jobs. OnChange is implemented through the DDL eventing mechanisms of SQL Server, which provides a mechanism to react to engine events. In order to evaluating policies in response to events, the whole execution path (Policy Engine, facet and SMO) needs to reside in SQLCLR. We chose to keep both code paths (inside or outside of SQLCLR); because of security restrictions on what can run inside SQLCLR. With the code path outside, we support broader policies for OnSchedule mode.

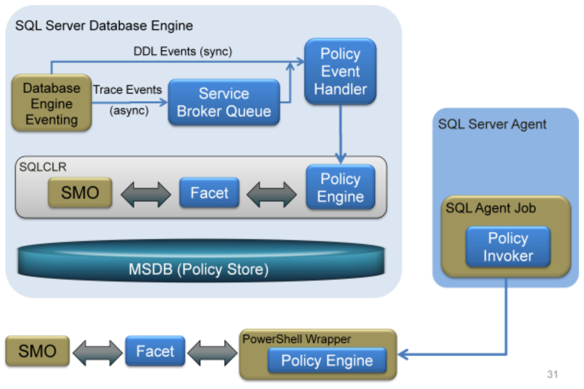

Figure 5: System Architecture

4.4 Management Model Implementation

Each version of Microsoft SQL Server includes a management model called SMO (SQL Server Management Objects) for relational database management entities like Tables, Views, Triggers, etc. In its implementation, it is a CLR-based[3] metadata object model. Management entity types are captured as CLR type definitions, with extra attributes to indicate specific management properties and relationships.

We extended SMO to function as the management model for declarative management. In addition to the core model, its implementation satisfies the basic requirements described in Section 3.1:

- An instance of the model can be populated from a running instance of SQL Server by issuing appropriate queries against the system metadata tables, and assembling objects from the data results. In effect, the model implementation abstracts the implementation details of the metadata catalog tables that maintain the necessary information on the server.
- Allowed changes to a model instance can be propagated back to the server by sending the appropriate CREATE/ALTER/DELETE commands.
- Queries over the model are supported, and they return sets of model entities. For example, the model can be queried for *Databases* that are compressed. These query requests are transparently translated to queries over the server metadata catalogs, and the results from the server lead to the population of model entities.

The detailed knowledge needed to translate between the model representation and the database server is fully encapsulated within the SMO implementation, and the application consuming SMO does not need this knowledge.

SMO is the basis of the existing management tools in SQL Server as well as the basis for the new Declarative Management features. Management facets are also defined on the SMO entity types. The detailed APIs used to expose facets are beyond the scope of this paper and we refer the reader to the product documentation [20]. The relevant implementation detail is that all facets are discoverable through .NET reflection on the implementation, and that all facets have a common interface to retrieve and change properties. This allows the DBA-facing management tools to build generic declarative management capabilities that still expose the full richness of the management model.

4.5 Client-Side Evaluation

The simplest form of declarative management is evaluation of a specific Health Condition against a specific target. One of the basic design decisions was whether to support client-side evaluation or not. Without a client-side evaluation mechanism, all expression logic would need to be evaluated in the database server — i.e. the Boolean expression in the Health Condition would need to be translated (through the management model implementation) into an equivalent SQL expression. This has obvious merits and elegance. However, it has some practical limitations as well:

- It prevents all use cases in "disconnected" environments — for example, while SQL applications are being developed or over exported metadata definitions.
- It disallows the use of management facets whose property computations are complex. Remember that a management facet can abstract a complex mapping between the facet properties and the underlying server metadata. This mapping may involve procedural logic that cannot be translated into corresponding declarative SQL query constructs.
- It does not work for management models of anything other than a database server — for example, to manage the configuration of SQL Server product installation (configuration properties stored in the operation system registry).

For these reasons, we decided to support client-side evaluation of policies, optimizing the behavior when possible to leverage the backend database server.

The basic requirement is to evaluate the property values of the appropriate target facet (SMO serves this purpose), and also to evaluate the Boolean expression specified by the Health Condition. This requires an expression evaluator that can run within the client — while this is not a difficult task; however, one must ensure that the evaluation semantics on the client are identical to that

[3] CLR: The Microsoft Common Language Runtime

on the server. The reason for this semantic equivalence becomes evident next.

A more interesting form is to evaluate a Health Condition that also has an associated policy target binding. To continue with Example 1, consider the policy that requires that *SSN* columns in the *Payroll* application are encrypted. The Health Condition specifies a constraint on a facet of the *Column* entity type, and the target binding query specifies that this is for columns of tables in the *Payroll* database. There is a simple (but inefficient) mode of evaluation which is to instantiate all Columns in the client-side management model, filter out those that aren't in the target binding, and then evaluate the Health Condition against the remainder. Obviously, this violates basic principles of performant database query processing — the fact that this is processing of metadata doesn't make the issue any less important. In fact, typical large applications have several thousands of tables, each with several columns. This simplistic evaluation model is very inefficient[4].

4.6 Optimized Client-Side Evaluation

The first optimization is the concept of Pushing Target Bindings. Put simply, this translates the target binding into a query that the underlying server can evaluate and return only the data corresponding to the policy targets (in this cases, columns of tables in the *Payroll* database). Our implementation always pushes target binding evaluation to the backend database server.

The second optimization is pushing Health Conditions. Recall that the health condition is expressed in the positive, but the system needs to alert DBAs to unhealthy targets. Thus, pushing Health Conditions to the underlying server requires negating the expression. As discussed in the previous section, this is not always possible – it depends on the implementation of the management facet over which the Health Condition is defined.

Although there are performance benefits to be gained in the cases where the condition can be pushed down, we decided not to enable this optimization in the released product. There are two reasons for this decision.

The DBA does not (and should not need to) know the difference between a facet that has a simple mapping and a facet with a complex mapping. Applying our principle of simplicity, we believe that unexplained variations in performance for a new technology are more unsettling than consistent performance, even if that performance could sometimes be improved.

When previewing the capability with early adopters, a number of DBAs also indicated that they wanted to see the results of successful health condition evaluations as well the failures (in other words, concerns over correctness of policy specification are more important to them than concerns over performance during policy evaluation). In order to validate that they did specify the target binding correctly, they want to see the results of Health Condition evaluation against *all* members of the target set. In some cases, they justified this for reasons of auditing — they wanted to ensure that the other targets were indeed in conformance and print out a policy evaluation report as proof of conformance. Clearly, the performance optimization of Pushing Health Conditions violates this requirement.

This is a classic example of a case where an obvious choice from another domain does not necessarily apply to the management domain. We started working with Declarative Management building on a deep background in query processing. We were certain that performance optimizations like this one would be essential, yet we have learnt that simplicity, stability, and verifiability are more important in practice.

4.7 Expressiveness of Health Conditions

In our initial implementation, we have limited the expressivity of conditions to Boolean expressions over properties. The properties are of certain known data types (integers, strings, Booleans) and the standard relational and arithmetic operators are supported over these types to enable basic expressions. Expressions can be combined via AND, OR, NOT operators.
There are three obvious ways in which this expressive power is not sufficient to fully capture the kinds of intent DBAs need to express:

- Expressions over relationships are not expressible. For example, the Tables in a model are related to the Views that use them. A condition about Tables might be based on the Views that use them (for example, Tables that are used by any View owned by user X). In general, existential and universal quantification over relationships needs to be supported.
- Aggregations over relationships are not expressible. For example, Tables that are used by at least 3 Views.
- Expressions that need to utilize some detailed information that is not abstracted by the management model but that is actually available directly in the underlying system (for example, a registry key setting, a data value in a table, etc). With a large complex system like SQL Server 2008, there are many such special-cases, and it would make no sense to capture them all in a management model.

In a subsequent version, we intend to extend the expression language to support relationships with quantification and aggregation. However, the third limitation was crucial to address in the initial version, and we addressed it by enabling an Execute ("command") function that takes an arbitrary SQL command to be run on the underlying server and returns a value. In fact, by providing this capability, it even allows quantification and aggregation over relationships to also be expressed via a SQL query expression. The problem with this approach of course is that the declarative management system cannot reason about this logic. In the long-term, declarative management requires that the system understand and reason about management intent. Further, the reliance on a back-end server to execute such expressions undermines some of the rationale for implementing client-side evaluation capabilities (for example, the "offline" use cases). This is purely a short-term "escape hatch" built as a pragmatic measure associated with product delivery timeframes and usage.

4.8 Imposing a Policy

An earlier section described the complexities of imposing a health condition on a management target. In our implementation, we only attempt to impose conditions where the conjunction subexpressions that fail are of the form *[Property = value]* and the *Property* is settable in the management model.

Imposing a policy involves the following steps:

[4] It should be observed though that this is actually how "external" management systems build similar functionality. Since they know nothing about the underlying system being managed, and do not leverage the fact that it is a rich query processor, they pull all the metadata out of the system instead of pushing the policy logic into the system.

- Identifying the targets defined by the target binding.
- Imposing the health condition on each of them.

We do not attempt to maintain atomicity across the entire operation — doing so could involve a long-running transaction that has negative effects on the system. Instead, we complete the evaluation phase for all the targets, identify the subset of targets that fail the evaluation and need to be changed, and then attempt to change each in an independent atomic manner.

5. POLICY AUTOMATION

Mature database systems provide management automation capabilities – in SQL Server 2008, the automation service is called SQL Agent [20]. SQL Agent allows procedural tasks (scripts or programs) specified by the DBA to be run when particular events occur. The events could be driven by a recurring schedule or could be ad-hoc driven by changes in the system being managed. The task definitions as well as task execution histories are stored in a management database (called *msdb*). The automation service is reliable and scalable. There are built-in mechanisms to send messages to operators. SQL Server DBAs are very familiar with SQL Agent and have used versions of this capability for more than a decade. Policy automation is built on top of this automation infrastructure.

The same policy evaluation engine used for ad-hoc evaluation is at the core of automated evaluation. It is activated via specific change events, evaluates relevant policies that apply to the management target that has changed, and acts upon any violations as governed by the automation modes described below:

- On-Schedule: Activated by SQL Agent scheduled events, and logs any violations.
- On-Change – Log: Activated by asynchronous metadata change events, and logs any violations.
- OnChange – Prevent: Activated by a synchronous metadata changes, and on policy failure rolls back the metadata change itself.

Each of these modes raises design and implementation issues described next. The other possible automation modes described in Section 3.4 were prioritized lower based on customer feedback and scheduled for the next version of the product.

The automated policies themselves are persisted in the same automation database (msdb) as the rest of SQL Agent. Depending on the automation mode, other artifacts like triggers or SQL Agent tasks are created to implement the intent of the policies.

5.1 On-Schedule Evaluation

Scheduled policy evaluation is conceptually identical to client-side evaluation, except that it is executed as a SQL Agent task. We added a new task type to evaluate policies. We also had to extend the SQL Agent persistence service to persist policies[5].

The implementation of the new task type for policy evaluation requires a significant fixed cost associated with launching a new task – this is because of the need to launch a new process and load a number of managed libraries. There is no fundamental conceptual problem in lowering this cost, but it was a practical constraint in the initial version because of various technologies involved that could not be changed. Faced with this initial launch cost, there would clearly be a performance problem if there were a large number of scheduled policies. We addressed this problem by observing that in a practical system, there are relatively few unique schedules, and many policies would be scheduled on the same recurrence schedule. Consequently, we aggregate the policies that need to be evaluated at the same time, and amortize the task initialization costs across them.

It also became important to record the history of policy execution, since many customers want to use scheduled policy execution as a conformance audit mechanism.

An interesting security issue exists: under what security account should scheduled policies execute? This is important because the visibility of metadata is controlled by security permissions associated with the user account. When executing a target binding query, if some management entities are not visible to the user account executing the policy, they will not be returned, not be evaluated for conformance, and could therefore be presumed to be healthy. This is also a problem if policies are being used for compliance and auditing. Once again, in keeping with the principle of simplicity and transparency, we ensure that all automated policy evaluation happens in the context of a privileged user account that has visibility to all metadata in the system. Conversely, this could result in security vulnerability by exposing too much information. We mitigate this concern by controlling who can *create* automation policies in the system. By limiting this to users in a specific controlled administrative role, we balance the need for simplicity with the need for security.

5.2 On-Change Evaluation

There are two kinds of on-change evaluation modes — one is asynchronous and one is synchronous. The implementation of the two modes is related, and so they are described together.

Obviously, not all policies can support an on-change evaluation mode. Associated with every facet is a definition of an event that should be raised to signal a change in properties of that facet. With our example of requiring encryption for certain columns, any schema definition change on columns should raise an appropriate event. *If* the underlying database system supports the necessary event, then a policy on a facet can utilize on-change evaluation modes. In the case of SQL Server 2008, most schema and configuration changes on the database engine can raise events if the appropriate event subscriptions have been defined. Some of these are synchronous events (a.k.a. "DDL triggers") and some of these are asynchronous events. The events carry with them some dynamic event data that includes the identity of the object actually changing.

When an event occurs corresponding to a change, the event is routed through multiplexing logic (a complex join) that attempts to match it up against automation policies that might need to react to the event. The logic has the following constraints:

- Only policies on the particular facet(s) corresponding to this event are relevant.
- Only policies that are active (i.e. not disabled) are relevant
- Only policies whose target set includes the object that was changed are relevant.

The last constraint has potentially drastic performance impacts. Remember that the target set is a query definition, so effectively, when an event occurs, we need to determine membership of the

[5] We actually represent and persist Health Conditions as first-class entities separate from the Policies that use them — this enables reuse of Health Conditions, and important consideration that we discuss later

affected object in one of many queries. This is the famous "inverse-query" problem that has been well-studied in publish-subscribe systems [12]. It is very inefficient to execute each query to see if the particular object is a part of the query result. For a system with any significant volume of such events, and for any significant number of policies with target set queries, this has the capacity to overwhelm the resources of the system. This can be especially difficult to justify since none of the changed objects might actually be members of any of the target sets.

For these pragmatic reasons, our product implementation restricted the expressive power of queries that could be specified in the target sets of policies that used on-change evaluation modes. By requiring simple queries that could be easily indexed, we were able to trade off expressive power for acceptable performance.

5.3 Policy History and System Health

The one-million-dollar question for the DBA is: what is the system health with regard to the set of policies? In the management model SMO (see Section 4.4) the basic relationship between entities is containment. For example, an instance contains databases, a database contains tables, views and stored procedures, and a table contains columns, keys and constraints. Such containment is visualized in the management tool, SQL Server Management Studio (SSMS), as shown in Figure 6. Naturally, the DBA would like to know the aggregated health state, in particular, errors, for each node in the hierarchy. Correspondingly, we define the aggregated health state for any sub-tree as follows:

1) An internal node is in violation iff any of its descendents is in violation.
2) A leaf node is in violation iff it violates any relevant policy.

The next step is to define "relevant" policies. DBAs govern their system using a set of automated policies. We say a policy is relevant if and only if it is automated.

For any given policy, it may be evaluated more than once. All policy evaluation history is recorded. Given any point of time, we have enough information to calculate the policy health state up to that point. DBA may need that information for analysis or reporting purposes, however, as part of their job responsibility, it is crucial for them to know the latest health state of the system, so as to take corresponding actions to correct the system. Figure 6 shows the health state of a server instance. As pointed by the arrow, in front of any object icon there is a scroll with a red cross indicates that object is critical. This provides visual cue for DBA to diagnose and correct the problem. Given the importance and frequency of such query, we optimize our data structure and algorithm to answer it efficiently.

First, we maintain a "materialized view"6, which we call the health state table of the history table which only contains the latest violations for the relevant policies. We call it the Health State table. Compared to the history table, it is much smaller in size.

Secondly, we utilize the path information of the objects to do a conceptual prefix search to calculate the health state. The Health State table essentially contains objects that violate policies. The question we need to answer is: given a node P, is there any entry in the Health State table that is a descendent of P (inclusive)? Each node in the tree can be uniquely identified by its path. A nice property of the tree structure is the following: assume the path of P is P_{path}, then P_{path} is the prefix for the path of any descendent of P. Conceptually, we store the path for each node in the Health State table and index it. Leveraging the prefix property, now the query becomes: exists Q, that P_{path} is a prefix for Q_{path}, which can be answered efficiently with the index.

[6] Because of the complexity of the maintenance rules, in our implementation we have to maintain the table instead of relying on the materialized view mechanism.

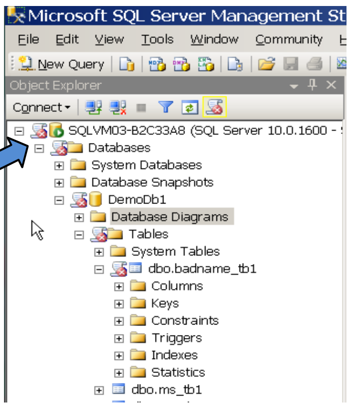

Figure 6: Object Hierarchy and Health State in Mgmt Tool

6. CONCLUSION AND FUTURE WORK

The current management technologies and tools are both DBA-intensive as well as prone to DBA-error. The industry trends — growth of system complexity, consolidation of data centers, increased heterogeneity of environment being managed, sharp increase of relative cost of skilled DBA and the increased relative cost of human-error — cry out for new technology that drastically increases DBA productivity and correctness. We proposed a novel approach, declarative management, to answer this challenge. This paper described the problem space, the basic concepts and an implementation in SQL Server 2008.

This is a first yet foundational step towards reduce the total cost of administration. Future extension includes:

- Developing a "Total Cost of Administration benchmark" and conducting case studies on how SQL Server customers benefit from declarative management.
- Broadening to complete self-managing database systems.
- Application in a cloud-computing infrastructure.
- Applying the declarative management concepts to data-tier applications. Build management model about applications and manage them by policies.
- Developing intelligent facets that abstract the management complexity of underlying system.
- Integrating tuning advisors into the eco system. For complex scenarios, for example, performance problem, invoke advisors for corrective action suggestions.

ACKNOWLEDGMENTS

Thanks to the SQL Server Manageability team and to Ciprian Gerea, Grigory Pogulsky, and Sethu Srinivasan in particular for significant contributions to the SQL Server 2008 implementation of policy-based management.

REFERENCES

[1] S. Agrawal, S. Chaudhuri, and V. R. Narasayya, "Automated Selection of Materialized Views and Indexes in SQL Databases," in *Proceedings of the 26th International Conference on Very Large Data Bases* , Cairo, Egypt, 2000, pp. 496-505.

[2] Amazon Web Services. [Online]. http://aws.amazon.com/

[3] M. Armbrust et al., "Above the Clouds: A Berkeley View of Cloud Computing," University of California at Berkeley, Technical Report UCB/EECS-2009-28, 2009.

[4] J. Arwe and et al. (2007, March) Service Modeling Language. [Online]. http://www.w3.org/Submission/sml

[5] P.A. Bernstein et al., "The Asilomar Report on Database Research," *SIGMOD Record*, vol. 27, no. 4, pp. 74-80, 1998.

[6] K. Brown, M. Carey, and M. Livney, "Goal-Oriented Buffer Management Revisited," in *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'96)*, Montreal, Canada, 1996, pp. 353-364.

[7] A. B. Brown and D. A. Patterson, "To Err is Human," in *Proceedings of the First Workshop on Evaluating and Architecting System dependabilitY (EASY '01)*, Göteborg, Sweden, 2001.

[8] S. Chaudhuri and V. Narasayya, "An Efficient, Cost-driven Index Tuning Wizard for Microsoft SQL Server," in *23rd International Conference on Very Large Data Bases*, Athens, Greece, 1997.

[9] S. Chaudhuri and V. Narasayya, "Self-Tuning Database Systems: A Decade of Progress," in *Proceedings of the 33rd International Conference on Very Large Databases*, Vienna, Austria, 2007.

[10] S. Chaudhuri and G. Weikum, "Rethinking Database System Architecture: Towards a Self-tuning RISC-style Database System," in *Proceedings of the 26th International Conference on Very Large Data Bases*, Cairo, Egypt, 2000, pp. 1-10.

[11] (2007) Dynamic Systems Initiative Overview White Paper. [Online]. http://www.microsoft.com/business/dsi/dsiwp.mspx

[12] P. T. Eugster, P. A. Felber, R. Guerraoui, and A. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 114 - 131, June 2003.

[13] L. Galanis et al., "Oracle Database Replay," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, Vancouver, Canada, 2008, pp. 1159-1170.

[14] P. Gil et al., "Fault Representativeness," IST-2000-25425, 2002.

[15] J. Gray, "A Census of Tandem System Availability Between 1985 and 1990," *IEEE Transactions on Reliability*, vol. 39, pp. 409-418, 1990.

[16] J. Gray, "Why do Computers Stop and What Can Be Done About It?," in *Proceedings of 5th Symposium on Reliability in Distributed Software and Database Systems*, 1986.

[17] J. L. Hellerstein, "Automated Tuning Systems: Beyond Decision Support," in *Int. CMG Conference 1997*, Orlando, Florida, 1997, pp. 263-270.

[18] Hewlett Packard. Hewlett Packard Web site. [Online]. https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-15_4000_100__

[19] IBM Corporation. IBM Corporation Web site. [Online]. http://www.ibm.com/software/tivoli

[20] Microsoft. (2009) Books Online. [Online]. http://msdn.microsoft.com/en-us/library/ms130214.aspx

[21] Microsoft Corporation. Microsoft Corporation Web site. [Online]. http://www.microsoft.com/systemcenter

[22] Microsoft Oslo Modeling Platform. [Online]. http://www.microsoft.com/soa/products/oslo.aspx

[23] Microsoft. (2006, May) SQL Server and Oracle Total Cost of Administration White Paper. [Online]. http://download.microsoft.com/download/a/4/7/a47b7b0e-976d-4f49-b15d-f02ade638ebe/Alinean-TCAStudy.pdf

[24] F. Oliveira et al., "Understanding and Validating Database System Administration," in *USENIX Annual Technical Conference, General Track*, Boston, MA, 2006, pp. 213-228.

[25] Opnet. OPNET Corporation Web site. [Online]. http://www.opnet.com/solutions/application_performance/panorama.html

[26] Oracle. (2007, August) Oracle Database 11g Manageability Overview White Paper. [Online]. http://www.oracle.com/technology/products/manageability/database/pdf/wp07/owp_manageability_11g.pdf

[27] J. Rao, C. Zhang, N. Megiddo, and G. Lohman, "Automating physical database design in a parallel database," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data* , Madison, WI, 2002, pp. 558 - 569.

[28] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis, "COLT: continuous on-line tuning," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, Chicago, IL, 2006, pp. 793 - 795.

[29] M. Seltzer and M. Olson, "Challenges in Embedded Database System Administration," in *Proceedings of the 1999 USENIX Workshop on Embedded Systems*, Cambridge, MA, 1999.

[30] M. Vieria and H. Madeira, "A Dependability Benchmark for OLTP Application Environments," in *In Proceedings of the 29th International Conference on Very Large Data Bases*, Berlin, Germany, 2003, pp. 742-753.

[31] G. Weikum, A. Mönkeberg, C. Hasse, and P. Zabback, "Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering," in *28th International Conference on Very Large Data Bases*, Hong Kong, 2002, pp. 20-31.