Understanding and Validating Database System Administration

Fábio Oliveira, Kiran Nagaraja, Rekha Bachwani,
Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen
Department of Computer Science
Rutgers University, Piscataway, NJ 08854

Abstract

A large number of enterprises need their commodity database systems to remain available at all times. Although administrator mistakes are a significant source of unavailability and cost in these systems, no study to date has sought to quantify the frequency of mistakes in the field, understand the context in which they occur, or develop system support to deal with them explicitly. In this paper, we first characterize the typical administrator tasks, testing environments, and mistakes using results from an extensive survey we have conducted of 51 experienced administrators. Given the results of this survey, we next propose system support to validate administrator actions before they are made visible to users. Our prototype implementation creates a validation environment that is an extension of a replicated database system, where administrator actions can be validated using real workloads. The prototype implements three forms of validation, including a novel form in which the behavior of a database replica can be validated even without an example of correct behavior for comparison. Our results show that the prototype can detect the major classes of administrator mistakes.

1 Introduction

Most enterprises rely on at least one database management system (DBMS) running on commodity computers to maintain their data. A large fraction of these enterprises, such as Internet services and world-wide corporations, need to keep their databases operational at all times. Unfortunately, doing so has been a difficult task.

A key source of unavailability in these systems is database administrator (DBA) mistakes [10, 15, 20]. Database administration is mistake-prone as it involves many complex tasks, such as storage space management, database structure management, and performance tuning. Even worse, as shall be seen, DBA mistakes are typically not maskable by redundancy (as in an underlying RAID subsystem) or standard fault-tolerance mechanisms (such as a primary-backup scheme). Thus, DBA mistakes are frequently exposed to the surrounding systems, database applications and users, causing unavailability and potentially high revenue losses.

Previous work has categorized DBA mistakes into broad classes and across different DBMSs [10]. However, no previous work has quantified the frequency of the mistakes in the field, characterized the context in which they occur, or determined the relationship between DBA experience and mistakes. Furthermore, no previous work has developed system support to deal with DBA mistakes explicitly.

In this paper, we address these issues in detail. We first characterize (in terms of class and frequency) the typical DBA tasks, testing environments, and mistakes, using results from an extensive survey we have conducted of 51 DBAs with at least 2 years of experience. Our survey responses show that tasks related to recovery, performance tuning, and database restructuring are the most common, accounting for 50% of the tasks performed by DBAs. Regarding the frequency of mistakes, the responses suggest that DBA mistakes are responsible (entirely or in part) for roughly 80% of the database administration problems reported. The most common mistakes are deployment, performance, and structure mistakes, all of which occur once per month on average. These mistakes are caused mainly by the current separation of and differences between testing and online environments.

Given the high frequency of DBA mistakes, we next propose system support to *validate* DBA actions before exposing their effects to the DBMS clients. As we described in [16], the key idea of validation is to check the correctness of human actions in a *validation environment* that is an extension of the online system. In particular, the components under validation, called *masked* components, are subjected to realistic (or even live) workloads. Critically, their state and configurations are not modified

when transitioning from validation to live operation.

In [16], we proposed trace and replica-based validation for Web and application servers. Both techniques rely on samples of correct behavior. Trace-based validation involves periodically collecting traces of live requests and replaying the trace for validation. Replica-based validation involves designating each masked component as a "mirror" of a live component. All requests sent to the live component are then duplicated and also sent to the mirrored, masked component. Results from the masked component are compared against those produced by the live component. Here, we extend our work to deal with DBMSs by modifying a database clustering middleware called Clustered-JDBC (C-JDBC) [7].

Furthermore, we propose a novel form of validation, called *model-based validation*, in which the behavior of a masked component can be validated even when we do not have an example of correct behavior for comparison. In particular, we use model-based validation to verify actions that might change the database structure.

We evaluate our prototype implementation by running a large number of mistake-injection experiments. From these experiments, we find that the prototype is easy to use in practice, and that validation is effective in catching a majority of the mistakes the surveyed DBAs reported. In particular, our validation prototype detected 19 out of 23 injected mistakes, covering all classes of mistakes reported by the surveyed DBAs.

In summary, we make three main contributions:

- We present a wealth of data on the behavior of experienced administrators of real databases. This contribution is important in that actual data on DBA mistakes is not publicly available, due to commercial and privacy considerations.
- We propose model-based validation for the situations when the behavior of the components affected by the DBA actions is supposed to change and there are no instances of correct behavior for comparison.
- We implement a realistic validation environment for dealing with DBA mistakes. We demonstrate the benefits of the prototype through an extensive set of mistake-injection experiments.

The remainder of the paper is organized as follows. The next section describes the related work. Section 3 describes our survey and analyzes the responses we received. Section 4 describes validation and our prototype. Section 5 presents our validation results. In Section 6, we broaden the discussion of the DBA mistakes and the validation approach to a wider range of systems. Finally, Section 7 draws our conclusions.

2 Related Work

Database administration mistakes. Only a few papers have addressed database administrator mistakes in detail. In two early papers [11, 12], Gray estimated the frequency of DBA mistakes based on fault data from deployed Tandem systems. However, whereas today's systems are mostly built from commodity components, the Tandem systems included substantial custom hardware and software for tolerating single faults. This custom infrastructure could actually mask several types of mistakes that today's systems may be vulnerable to.

The work of Gil *et al.* [10] included a categorization of administrator tasks and mistakes into classes, and a comparison of their specific details across different DBMSs. Vieira and Madeira [20] proposed a dependability benchmark for database systems based on the injection of administrator mistakes and observation of their impact. In this paper, we extend these contributions by quantifying the frequency of the administrator tasks and mistakes in the field, characterizing the testing environment administrators use, and identifying the main weaknesses of DBMSs and support tools with respect to database administration. Furthermore, our work develops system support to deal with administrator mistakes, which these previous contributions did not address.

Internet service operation mistakes. A few more papers have addressed operator mistakes in Internet services. The work of Oppenheimer *et al.* [17] considered the universe of failures observed by three commercial services. With respect to operators, they broadly categorized their mistakes, described a few example mistakes, and suggested some avenues for dealing with them. Brown and Patterson [4] proposed "undo" as a way to rollback state changes when recovering from operator mistakes. Brown [3] performed experiments in which he exposed human operators to an implementation of undo for an email service hosted by a single node. In [16], we performed experiments with volunteer operators, describing all of the mistakes we observed in detail, and designing and implementing a prototype validation infrastructure that can detect and hide a majority of the mistakes. In this paper, we extend these previous contributions by considering mistakes in database administration and introducing a new validation technique.

Validation. We originally proposed trace and replica-based validation for Web and application servers in Internet services [16]. Trace-based validation is similar in flavor to fault diagnosis approaches [1, 8] that maintain statistical models of "normal" component behavior and dynamically inspect the service execution for deviations from this behavior. These approaches typically focus on the data flow behavior across the systems com-

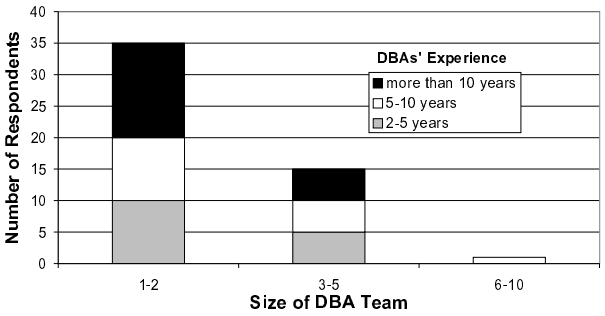

Figure 1: *Distribution of DBAs across team sizes.*

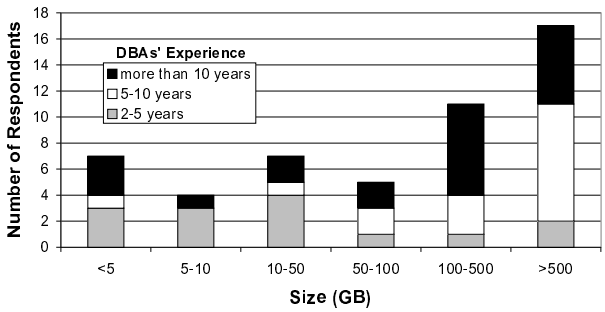

Figure 2: *Distribution of DBAs across database sizes.*

ponents, whereas trace-based validation inspects the actual responses coming from components and can do so at various semantic levels.

Replica-based validation has been used before to tolerate Byzantine failures and malicious attacks, e.g. [5, 6, 13]. In this context, replicas are a permanent part of the distributed system and validation is constantly performed via voting.

Model-based validation is loosely related to two approaches to software debugging: model checking (e.g., [21]) and assertion checking (e.g., [9, 18]). Of these, it is closest to the PSpec system [18] for assertion checking. However, because PSpec was concerned with performance problems as opposed to detecting human mistakes, the authors did not consider structural issues such as component connectivity and database schemas. Besides its focus on human mistakes, model-based validation differs from other assertion-checking efforts (e.g., [9]) in that our assertions are external to the component being validated. Model-based validation differs from model checking in that it validates components dynamically based on their behavior, rather than statically based on their source codes.

In this paper, we extend our work on trace and replica-based validation to database servers, which pose a number of new challenges (Section 6). For example, our previous validation prototype did not have to manage hard state during or after validation. In validating database systems, we need to consider this management and its associated performance and request buffering implications. Furthermore, we propose model-based validation to validate actions without examples of correct behavior.

Other approaches to dealing with mistakes. Validation is orthogonal to Undo [4] in that it hides human actions until they have been validated in a realistic validation environment. A more closely related technique is "offline testing" [2]. Validation takes offline testing a step further by operating on components in an environment that is an extension of the live system. This allows validation to catch a larger number of mistakes as discussed in [16].

3 Understanding DB Administration

We have conducted an online survey to unveil the most common tasks performed by DBAs, the problems and mistakes that occur during administration, and aspects of the environment in which DBAs carry out their duties. We have posted the survey (available at http://vivo.cs.rutgers.edu/dba_survey.html) to the USENIX SAGE mailing list and the most visible database-related Usenet newsgroups, namely *comp.databases.** and *comp.data.administration*. We next present an analysis of the 51 responses we received.

3.1 Main Characteristics of the Sample

The DBAs who replied to our survey represent a wide spectrum of organizations, DBMSs, experience levels, DBA team sizes, and database sizes. In particular, judging by the DBAs' email addresses, they all work for different organizations. The DBAs use a variety of DBMSs, including Microsoft SQL Server, Oracle, Informix, DB2, Sybase, MySQL, PostgreSQL, Ingres, IMS, and Progress. The most common DBMSs in our sample are MS SQL Server (31%), Oracle (22%), and MySQL (13%).

The DBAs are highly experienced: 15 of them had between 2 and 5 years of experience, 16 had between 5 and 10 years of experience, and 20 had more than 10 years of experience. Figures 1 and 2 show the sizes of the administration teams to which the DBAs belong, and the sizes of the databases they manage, respectively. The figures show the breakdown of DBAs per experience level.

From Figure 1, it is clear that most of the DBAs work alone or in small groups, regardless of experience level. This result suggests that the size of DBA teams is determined by factors other than DBA experience. Nevertheless, a substantial number of DBAs do work in larger teams, increasing the chance of conflicting actions by different team members. Finally, Figure 2 shows that DBAs of all experience levels manage small and large databases. However, one trend is clear: the least experienced DBAs (2–5 years of experience) tend to manage

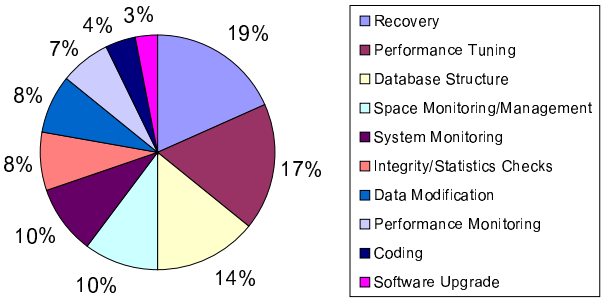

Figure 3: *Task categories. The legend lists the categories in decreasing order of frequency.*

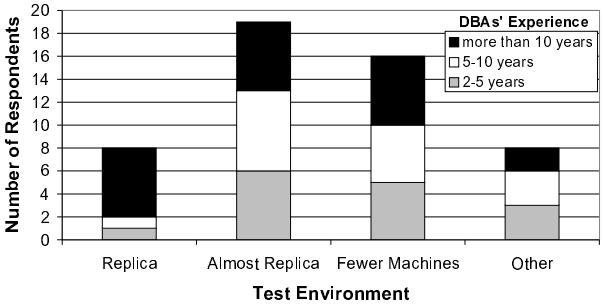

Figure 4: *Test environment configurations.*

smaller databases; they represent 12% of the DBAs responsible for databases that are larger than 50 GB, but represent 55% of those DBAs in charge of databases that are smaller than 50 GB.

The experience level of the DBAs who participated in our survey and the diversity in organizations, DBMSs, and team and database sizes suggest that the data we collected is representative of common DBA practices.

3.2 Common DBA Tasks

We asked the DBAs to describe the three most common tasks they perform. Figure 3 shows the categories of tasks they reported, as well as the breakdown of how frequently each category was mentioned out of a total of 126 answers (several DBAs listed fewer than three tasks).

The *Recovery* category corresponds to those tasks that prepare or test the DBMS with respect to recovery operations, such as making backups, testing backups, and performing recovery drills. *Performance Tuning* tasks involve performance optimizations, such as creating or modifying indexes to speed up certain queries, and optimizing the queries themselves. The *Database Structure* tasks involve changing the database schema by adding or removing table columns, or adding or removing entire tables, for example. The *Space Monitoring/Management*, *System Monitoring*, *Performance Monitoring*, and *Integrity/Statistics Checks* tasks all involve monitoring procedures, such as identifying the applications or queries that are performing poorly. The *Data Modification* category represents those tasks that import, export, or modify actual data in the database. Finally, the *Coding* task involves writing code to support applications, whereas the *Software Upgrade* tasks involve the upgrade of the operating system, the DBMS, or the supporting tools.

The three most frequently mentioned categories were Recovery, Performance Tuning, and Database Structure, respectively amounting to 19%, 17%, and 14% of all tasks described by the DBAs. If we collapse the Space Monitoring/Management, System Monitoring, and Performance Monitoring categories, we can see that 27% of the reported tasks have to do with checking the system behavior. From a different perspective, 24% of all tasks are related to performance, in which case the DBAs are engaged in either identifying the reasons for poor performance (Performance Monitoring) or looking for opportunities to further improve the overall DBMS performance (Performance Tuning). If we consider that checking and updating database statistics are carried out to allow for more accurate optimizations, the fraction of performance-oriented tasks is actually 32%.

Twenty one DBAs (41%) reported that they use third-party support tools for these different tasks. Several of these DBAs actually use multiple types of tools. In more detail, 14 DBAs use tools for Performance Monitoring tasks, 12 DBAs use tools for Recovery tasks, 12 DBAs use tools for Database Structure tasks, 7 DBAs use tools for Performance Tuning tasks, and 7 DBAs use tools for granting/revoking access privileges; the latter tools may be needed in Database Structure, Coding, and Software Upgrade tasks.

3.3 Testing Context

Another important set of questions in our survey concerned the context in which the above tasks are performed. More specifically, we sought to understand the testing methodology that DBAs rely upon to verify the correctness of their actions. The next paragraphs describe our findings with respect to the environment and approach used for testing.

Testing environment. Figure 4 depicts the distribution of test environments, breaking results down with respect to the DBA experience levels. We categorize the environments according to how similar they are to the online database environment: an exact replica (*Replica*), an environment that differs from the online environment only with respect to how powerful the machines are (*Almost Replica*), an environment with fewer machines than the online environment (*Fewer Machines*), or other configu-

Problem category	Average frequency	Number of DBAs per experience level			% of all problems	Caused by DBA mistakes
		2–5 years	5–10 years	> 10 years		
Deployment problems	Once a month	4	3	3	11	most
Performance problems	Once a month	2	2	4	9	most
General structure problems	Once a month	2	4	3	10	most
DBMS problems	Once a month	1	1	2	5	none
Access-privilege problems	Once in 2 months	3	5	1	10	all
Space problems	Once in 2 months	3	3	8	16	some
General maintenance problems	Once in 3 months	6	5	5	18	most
Hardware problems	Once a year	3	6	5	16	none
Data problems	Once a year	–	1	3	5	some

Table 1: *Reported problems, estimated average frequency of occurrence, number of DBAs who mentioned each problem as frequent broken down by DBA experience, percentage of DBAs who mentioned the problem as frequent, and, qualitatively, how often it is caused by DBA mistakes.*

rations (*Other*). Irrespective of the test environment, the test machines are typically loaded with only a fraction of the online database.

We can see that 84% of the DBAs test their actions in environments that are different from the online environment. Further, DBAs of different experience levels use these non-replica testing practices in roughly similar percentages. We conjecture that these practices are a result of the performance and cost implications of using exact replicas for testing. Regardless of the reason, it is possible for actions to appear correct in these environments, but cause problems when migrated to the online environment. Not to mention the fact that the migration itself is mistake-prone, since it is performed by the DBA manually or via deployment scripts.

Another interesting observation is that 8 DBAs use other approaches to testing. Three of these DBAs replicate the online databases on the online machines themselves, and use the replicated databases as test instances. This approach is problematic when the DBA actions involve components shared by online and testing environments, e.g. the operating system or the I/O devices.

Two of the most experienced DBAs also mentioned a well-structured testing environment comprising three sets of machines: (1) "development machines" used for application developers to ascertain that a particular application interacts with the necessary databases as expected; (2) "integration machines", which host all applications and are subjected to more aggressive tests, including the use of load generators; (3) "quality-assurance machines" used for ensuring that the system conforms with established standards before it can be deemed deployable.

Testing approach. Another important issue is how DBAs test modifications to existing databases and newly designed databases before deployment to the production system. Most DBAs (61%) report that they perform testing manually or via their own scripts. Two other DBAs report that, depending upon the nature of the actions to be performed, they do not carry out any offline testing whatsoever. Finally, 2 DBAs reported testing by means of documented operability standards that specify a set of requirements to be satisfied before deployment to production. In such cases, to determine if the requirements are obeyed, performance tests, design reviews, and security analyses are carried out.

3.4 Problems in DB Administration

We asked the DBAs to describe the three most frequent problems that arise during database administration. Based on the descriptions they provided, we derived the categories listed in Table 1. The table shows how frequently (on average) these DBAs estimate each problem category to occur, how many DBAs with different experience levels alluded to the category, the percentage of DBAs who mentioned the category, and qualitatively how often a problem in the category is caused by a DBA mistake. In the following paragraphs, we describe the problems, their causes, and how they affect the system.

Deployment problems. This category of problems occurs when changes to the online system cause the database to misbehave, even though the changes may have been tested in an offline testing environment. These problems occur in DBA tasks that involve migrating changes from a testing environment to the online environment (i.e., Performance Tuning, Database Structure, Data Modification, Coding, and Software Upgrade tasks), and are typically due to DBA mistakes committed during this migration process. Specifically, the DBAs reported a number of causes for these problems: (1) bugs in the DBA's deployment scripts, which are aggravated by the DBMSs' typically poor support for debugging and logging changes; (2) DBAs forget to change the structure of the online database before deploying a new or recently

modified application; (3) DBAs accidentally propagate the changes made to the database in the testing environment to the online system; (4) DBAs make inappropriate changes directly to the online database; (5) DBAs forget to reapply indexes in the production database; and (6) applications compiled against the wrong database schema are deployed online.

Note that causes (1)–(5) are DBA mistakes. Some of them affect the interaction between the database and the applications. If the deployed database and the applications are not consistent, non-existent structures might be accessed, thus generating fatal SQL errors.

As shown in Table 1, deployment problems occur frequently (once a month on average), according to the 10 DBAs who mentioned this category. Out of all DBAs, 40% of them mentioned the lack of integrated versioning control for the database and its applications as the main weakness of current DBMSs and third-party tools with respect to deployment. Another 27%, 13%, and 13% mentioned the complexity of the DBMSs and tools, poor support for comparisons between test and online environments, and the burden posed by interdependencies between database objects, respectively.

Performance problems. Typically, when the DBMS delivers poor performance to applications or users, the culprit is the DBA, the application developer, or both. Two DBA mistakes compromising the DBMS performance were mentioned: (1) erroneous performance tuning, in the face of the plethora of configuration parameters offered by DBMSs; and (2) inappropriate database design, including database object structures and indexing scheme. These mistakes can occur in DBA tasks that involve performance tuning, removing/adding database objects, or changing the software (i.e., Performance Tuning, Database Structure, Coding, and Software Upgrade tasks). On the application developer side, the DBAs complained about poorly designed queries that take long to complete and consume a lot of resources.

As shown in Table 1, performance problems happen frequently (once a month on average), according to 8 DBAs. In fact, 4 of these DBAs have more than 10 years of experience and consistently commented on DBA-induced poor performance in particular.

General structure problems. Pertaining to this category are incorrect database design and unsuitable changes to the database, both produced by the DBA during Database Structure tasks and leading to malformed database objects, and ill-conceived code on the application developer's part. The DBAs mentioned two particular instances of incorrect database design in our survey: duplicated identity columns and columns too small to hold a particular type of data. Four DBAs also mentioned that mistakes in database design and poor application code are responsible for deadlocks they have observed. In fact, these 4 DBAs observe deadlocks very frequently, once in two weeks on average.

As Table 1 shows, general structure problems happen frequently (once a month on average), according to the 9 DBAs who mentioned them.

DBMS problems. Four DBAs were victims of bugs in DBMSs. Three of them said that the bugs had only minor impacts on the database operation, but the other said that a DBMS bug was the cause of an outage that lasted half a day. DBMS bugs were not mentioned by many DBAs, but the 4 DBAs who did mention them claim that these bugs occur once a month on average.

Access-privilege problems. Another category of problems affects the privileges to access the database objects. These problems can occur in DBA tasks that involve removing/adding database objects or changing the software (i.e., Database Structure, Coding, and Software Upgrade tasks). According to the DBAs, these problems are caused by two types of mistakes: (1) DBAs do not grant sufficient rights to users or applications, resulting in their inability to access the whole (or parts of the) database; and (2) DBAs grant excessive privileges to some users or applications. Obviously, the latter situation causes a serious security vulnerability.

According to the 9 DBAs who mentioned this category, access-privilege problems also occur frequently (once in two months on average). Interestingly, 1 DBA who mentioned this category uses a third-party tool specifically for granting/revoking access privileges to database objects.

Space problems. This category consists of disk space exhaustion and tablespace (i.e., the space reserved for a set of tables and indexes) problems. These problems are most serious when the DBA fails to monitor and manage the space appropriately (i.e., Recovery and Space Monitoring/Management tasks). Disk space exhaustion is caused chiefly by growing transaction logs, alert logs, and the like. Some DBAs mentioned that the unpredictability of the application users' behavior makes it difficult to foresee a disk space shortage.

Some DBAs also reported tablespaces unexpectedly filling up. Further, a few DBAs mentioned the impossibility of extending a completely used tablespace. Another tablespace-related problem occurs in the context of tablespace defragmentation, an operation that DBAs perform to prevent performance degradation on table accesses. A typical procedure for defragmenting a tablespace involves exporting the affected tables, dropping them, and re-importing them. During the defragmentation, 1 DBA was unable to re-import the tables due to a bug in the script that automated the procedure and, as a result, the database had to be completely restored.

According to the 14 DBAs who mentioned space problems, they occur once in two months on average.

General maintenance problems. This is the category of problems that DBAs mentioned most frequently; 16 DBAs mentioned it. These problems occur during common maintenance tasks, such as software or hardware upgrades, configuring system components, and managing backups. Regarding upgrades, some DBAs alluded to failures resulting from incompatibilities that arose after such upgrades. Other DBAs mentioned mistakes such as the DBA incorrectly shutting down the database and the DBA forgetting to restart the DBMS replication capability after a shutdown. In terms of configuration, 3 DBAs described situations in which the DBMS was unable to start after the DBA misconfigured it trying to improve performance. Regarding the management of backups, DBAs listed faulty devices and insufficient space due to poor management.

As Table 1 shows, general maintenance problems happen frequently (once in three months on average), according to the DBAs.

Hardware and data problems. Hardware failure and data loss are the least frequent problems according to the DBAs; they occur only once a year on average. 14 DBAs mentioned hardware failures, whereas only 4 mentioned data loss as a problem.

3.5 Summary and Discussion

We can make several important observations from the data described above:

1. Recovery, performance, and structure tasks are the most common tasks performed by DBAs. Several categories of tasks, including performance tuning, database restructuring, and data modification typically require the DBA to perform and test actions offline and then migrating or deploying changes to the online system.

2. The vast majority of the DBAs test their actions manually (or via their own scripts) in environments that are not exact replicas of the online system. The differences are not only in the numbers and types of machines, but also in the data itself and the test workload.

3. A large number of problems occur in database administration. The most commonly cited were general maintenance, space, and hardware problems. However, deployment, performance, and structure problems were estimated by several DBAs to be the most frequent.

4. Qualitatively, DBA mistakes are the root cause of *all or most* of the deployment, access-privilege, performance, maintenance, and structure problems; these categories represent 58% of the reported problems. They are also responsible for *some* of the space and data problems, which represent 21% of the reported problems. Unfortunately, the DBA mistakes are typically not maskable by traditional high-availability techniques, such as hardware redundancy or primary-backup schemes. In fact, the mistakes affect the database operation in a number of ways that may produce unavailability (or even incorrect behavior), including: (1) data becoming completely or partially inaccessible; (2) security vulnerabilities being introduced; (3) performance being severely degraded; (4) inappropriate changes and/or unsuitable design giving scope for data inconsistencies; and (5) careless monitoring producing latent errors.

5. DBAs of all experience levels make mistakes of all categories, even when they use third-party tools.

6. The differences and separation between offline testing and online environments are two of the main causes of the most frequent mistakes. Differences between the two environments can cause actions to be correct in the testing system but problematic in the online system. Applying changes that have already been tested in a testing environment to the online system is often an involved, mistake-prone process; even if this process is scripted, mistakes in writing or running the scripts can harm the online system.

These observations lead us to conclude that DBA mistakes have to be addressed for consistent performance and availability. We also conclude that DBAs need additional support beyond what is provided by today's offline testing environments. Thus, we next propose validation as part of the needed infrastructure support to help DBAs reduce the impact of mistakes on system performance and availability. As we demonstrate later, validation hides a large fraction of these mistakes from applications and users, giving the DBA the opportunity to fix his/her mistakes before they become noticeable, as well as eliminates deployment mistakes.

4 Validation

In this section, we first briefly review the overall validation approach. This review is couched in the context of a three-tier Internet service, with the third tier being a DBMS, to provide a concrete example application surrounding the DBMS. Then, we discuss issues that are specific to validating DBMSs and describe our prototype implementation. We close the section with a discussion of the generality and limitations of our approach.

4.1 Background

A validation environment should be as closely tied to the online environment as possible to: (1) avoid latent errors that escape detection during validation but become activated in the online system because of differences between the validation and online environments;

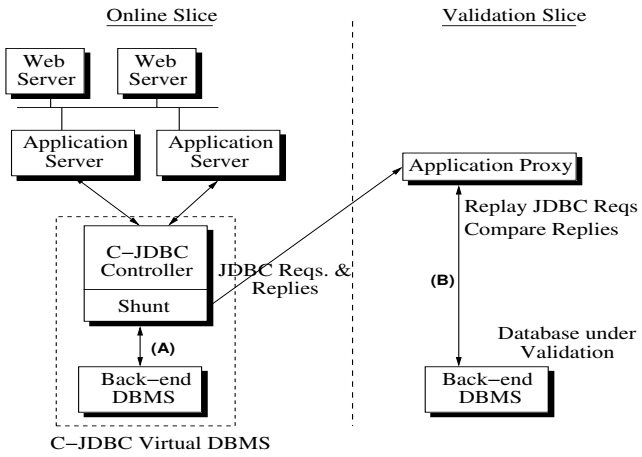

Figure 5: *Validation for a three-tier Internet service. In the figure, the two back-end databases are mirrored replicas. The database node in the validation slice is undergoing validation after the DBA has operated on it.*

(2) load components under validation with as realistic a workload as possible; and (3) enable operators to bring validated components online without having to change any of the components' configurations, thereby minimizing the chance of new operator mistakes. On the other hand, the components under validation, which we shall call *masked* components for simplicity, must be *isolated* from the online system so that incorrect behaviors cannot cause system failures.

To meet the above goals, we actually host the validation environment on the online system itself. In particular, we divide the components into two logical slices: an online slice that hosts the online components and a validation slice where components can be validated before being integrated into the online slice. Figure 5 shows this validation architecture when a component of the DBMS tier is under validation. To protect the integrity of the online service without completely separating the two slices (which would reduce the validation slice to an offline testing system), we erect an isolation barrier between the slices but introduce a set of connecting *shunts*. The shunts duplicate requests and replies (i.e., inputs and outputs) passing through the interfaces of the components in the live service. Shunts either log these requests and replies or forward them to the validation slice.

We then build a validation harness consisting of *proxy components* that can be used to form a virtual service around the masked components; Figure 5 shows an application proxy being used to drive a masked DBMS. Together, the virtual service and the duplication of requests and replies via the shunts allow operators to validate masked components under realistic workloads. In particular, the virtual service either replays previously recorded logs or accepts forwarded duplicates of live requests and responses from the shunts, feeds appropriate requests to the masked components, and verifies that the outputs of the masked components meet certain validation criteria. Proxies can be implemented by modifying open source components or wrapping code around proprietary software with well-defined interfaces.

Finally, the harness uses a set of *comparator functions*, which compute whether some set of observations of the validation service match a set of criteria. For example, in Figure 5, a comparator function might determine if the streams of requests and replies going across the pair of connections labeled *(A)* and *(B)* are similar enough to declare the masked database as working correctly. If any comparison fails, an error is signaled and the validation fails. If after a threshold period of time all comparisons match, the component is considered validated.

Given the above infrastructure, validation becomes conceptually simple. First, a script places the set of components to be worked on in the validation environment, effectively masking them from the live service. The operator then acts on the masked components just as he/she would in the live service. Next, another script instructs the validation harness to surround the masked components with a virtual service, load the components, and check their correctness. If the masked components pass this validation, the script calls a migration function that fully integrates the component into the live service.

4.2 Validation Strategies

In [16], we proposed two validation approaches: *trace-based* and *replica-based* validation. In trace-based validation, for each masked component to be validated, requests and replies passing through the shunts of an equivalent live component are logged and later replayed. During the replay, the logged replies can be compared to the replies produced by the masked component. In replica-based validation, the current offered load on the live service is used, where requests passing through the shunts of an equivalent live component are duplicated and forwarded in real-time to the validation harness to drive the masked component. The shunts also capture the replies generated by the live component and forward them to the harness, which compares them against the replies coming from the masked component.

Unfortunately, trace-based and replica-based validation are only applicable when the output of a masked component can be compared against that of a known correct instance. Many operator actions can correctly lead to a masked component behaving differently than all current/known instances, posing a bootstrapping problem. An example in the context of databases is a change to the database schema (a task that is cited as one of the most common DBA tasks in our survey). After the DBA

changes the schema (e.g., by deleting a column) in the validation environment, the masked database no longer mirrors the online database and so may correctly produce different answers to the same query. The same applies to a previously collected trace.

We propose **model-based validation** to deal with this bootstrapping problem. The key idea behind model-based validation is that a service, particularly one with components that have just been acted on by an operator, should conform to an explicit model. Thus, in model-based validation, we require an explicit representation of the intended consequences of a set of operator actions in a model of the system, and then validate that the dynamic behavior of the masked component matches that predicted by the model.

For example, a simple model for a service composed of a front-end load balancer and a set of back-end servers could specify an assertion to the effect that the resource utilization at the different back-end nodes should always be within a small percentage of each other. This simple model would allow us to validate changes to the front-end device even in the context of heterogeneous servers.

If we can conveniently express these models and check them during validation, we can validate several classes of operator actions that cannot be tackled by trace or replica-based validation. We envision a simple language that can express models for multiple components, including load balancers, firewalls, and servers.

4.3 Implementing Validation for DBMSs

We now describe our prototype validation environment for a service with replicated databases. A replicated database framework allows DBAs to operate on the masked DBMS and validate it while the online DBMS is still servicing live requests, an important property for systems that must provide 24x7 availability.

Our implementation leverages the C-JDBC database clustering middleware [7]. Briefly, C-JDBC clusters a collection of possibly heterogeneous DBMSs into a single virtual DBMS that exposes a single database view with improved scalability and dependability. C-JDBC implements a software controller between a JDBC application and the back-end DBMSs. The controller comprises a request scheduler, a load balancer, and a recovery log. C-JDBC supports a few data distribution schemes. Our prototype uses only one: full data replication across the DBMSs comprising a virtual DBMS. Under full replication, each read request is sent to one replica while writes are broadcast to all replicas.

As shall be seen, critical to our implementation is C-JDBC's capability to disable and disconnect a back-end DBMS, ensuring that its content is a consistent checkpoint with respect to the recovery log, and later reintegrate this DBMS by replaying the log to update its content to the current content of the virtual database.

As shown in Figure 5, our prototype relies on the C-JDBC controller only in the online slice; the application proxy contacts the database under validation directly. We implement the isolation barrier between the online and validation slices at the granularity of an entire node by running nodes over a virtual network created using Mendosus [14], a fault-injection and network-emulation tool for clustered systems. The general idea is to use Mendosus to partition the virtual network into two parts so that online nodes can see each other but not those in the validation slice and vice-versa. Critically, however, Mendosus can migrate a node between the two parts of the virtual network without requiring any change to the node's networking parameters.

The shunting of requests and responses takes place inside the C-JDBC controller, as can be seen in Figure 5. A critical aspect to be observed when implementing a database shunt is that of ordering. Suppose that the controller has three requests to dispatch within a single transaction: two read requests, *R1* and *R2*, and one write request, *W*. If the order in which the online database executes the requests is *R1*, *W*, and *R2*, whereas the database under validation executes *R1*, *R2*, and then *W*, the database replicas will report different responses for *R2* if the write request modifies the data read by *R2*. If a following request depends on *R2*, the situation becomes even worse. This undesirable ordering mismatch can not only trigger false positives during validation, but also, and more seriously, corrupt the database state.

A logical conclusion from this scenario is the paramount need to enforce a partial order of request execution that both the online database replicas and the database under validation must abide by. A number of consecutive read requests can be executed in any order, but blocks of reads separated by writes (or commits) must be executed in the same order with respect to the write (or commit) requests.

Although the C-JDBC controller guarantees that the mirrored databases connected to it are kept consistent with respect to each transaction independently, it does not guarantee ordering across transactions. In other words, in the face of concurrent transactions, the controller cannot determine whether the back-end databases will execute a read request from one transaction before or after a write request from another transaction. The identification numbers that the controller assigns to requests have nothing to do with actual order of execution. For this reason, we had to modify the controller code by introducing a multiple-reader-single-writer lock used only while requests are being shunted. This lock is used to ensure that each write is executed by itself on the back-end DBMSs, providing a complete ordering between all pairs

of read-write and write-write operations. By using this extra lock and capturing requests right before the controller issues them to the back-ends, our shunting code forwards/collects requests in an order that the application proxy can rely on. During validation, requests can be replayed in this order irrespective of the transactions to which they belong.

We have implemented all three validation strategies and two comparator functions, a reply exact-match and a latency-match comparator. We describe the important details of these implementations next.

Trace-based validation. Trace-based validation requires collecting traces from the online system. Given the above infrastructure, collecting a trace is simple: we ask the C-JDBC controller to create a copy of the database on a "trace disk" and to start logging requests and replies to it serialized to after the copy.

At a later point, suppose that the DBA wants to operate on one of two back-end DBMSs within a C-JDBC virtual DBMS, say to create an index to improve performance, and then use trace-based validation to validate his/her actions. He/she would run two scripts, one before operating on each DBMS and the other after doing so:

Script 1 – take the DBMS to be worked on offline. (1) Instruct the C-JDBC controller to checkpoint and disable the appropriate DBMS node. This means that the disabled database is consistent with a particular point in the controller's replay log. (2) Instruct Mendosus to move the masked DBMS into the validation slice. (3) Save the masked DBMS's current state to a persistent backup.

Script 2 – validate the modified DBMS and move it to the online slice. (1) Initialize the DBMS with the database on the trace disk. (2) Start the application proxy. (3) Once validation completes successfully, reinitialize the DBMS with the backup saved in script 1. (4) Instruct Mendosus to move the masked DBMS back to the online slice. (5) Attach the newly modified and validated DBMS to the C-JDBC controller and instruct the controller to integrate it. If validation fails in step (3), the DBA needs to fix any mistakes, and re-start the script.

Replica-based validation. Replica-based validation is quite similar to trace-based validation and can also be run using two scripts. Script 1 is the same as that used for trace-based validation. Script 2 performs the actions enumerated next. (1) Instruct Mendosus to move the masked DBMS back into the online slice, and instruct the controller to bring the masked DBMS up-to-date using its replay log, while keeping the masked DBMS in a *disabled state*. (We actually had to modify the controller to implement this functionality, since the controller would automatically enable a DBMS node after replaying the log.) (2) Now that the masked DBMS is again an exact replica of the online DBMS, start buffering writes (and commits) to the online virtual DBMS, migrate the masked DBMS back to the validation slice, and start the application proxy. (3) Enable the shunting of requests and replies to the application proxy, and restart write (and commit) processing on the C-JDBC controller. (4) Once validation completes successfully, instruct the controller to halt and buffer all incoming requests; migrate the masked DBMS to the online slice and let it connect to the C-JDBC controller. Note that the two DBMSs have exactly the same content at this point, which makes this reintegration quite fast. (5) Finally, instruct the C-JDBC controller to start processing requests again. If validation fails in step 4, the masked DBMS will remain in the validation slice and will be initialized with the state saved in script 1, so that it can be validated again after the DBA fixes any mistakes.

Note that script 2 forces the controller to buffer write (and commit) requests to the virtual DBMS for a very short period in replica-based validation; just enough time to migrate the masked DBMS to the validation slice and start the application proxy. Because these operations can be performed in only a few milliseconds, the amount of buffering that takes place is typically very small.

Model-based validation. Our vision is to allow the DBA to specify his/her actions in a simple canonical form and associate a small set of assertions with each action. The system can then validate the actions that are actually performed on the masked DBMS by checking that the assertions hold. The specification of actions in the canonical form should be *much simpler* than the actual execution of these actions (say, as SQL queries and commands). They should also be independent of specific implementations of DBMS, which is important because each implementation uses a different variant of SQL. This simplicity and portability are the main advantages of model-based validation. For example, automatically executing the needed actions from the canonical descriptions would require extensive implementations for all possible DBMSs C-JDBC can use.

We have prototyped a simple model-based validation strategy as a proof-of-concept. Our implementation currently focuses on database structure changes, since these tasks were identified as very frequent by the DBAs we surveyed. It includes four canonical actions: add a table, remove a table, add a column, and remove a column. It also defines a set of assertions that must be true about the database schema after an action has been performed compared to the schema before the action. For example, one of the assertions states that, if the DBA will add a table, the schema after the action should contain all the tables in the previous schema plus the table just created.

While this prototype implementation is quite simple, it is also powerful. The reason is that each canonical

action, such as adding a column, can correspond to a lengthy set of real actions. For example, adding a column to the middle of a table might be quite complicated depending on the DBMS being used [15]. This operation might involve, among many other things, unloading the data from the table and dropping it (which would in turn drop all indexes and views associated with the table); recreating the table with the new column; repopulating the table; recreating all necessary indexes and views; and checking if the application programs work correctly with the modified table. In model-based validation, we are only concerned with the model of the database schema. Thus, the operator might specify his/her action as "I will add a column to Table T between two existing columns, A and B." This allows model-based validation to check that, indeed, in the new schema, table T has one more column that is between A and B.

To address another common set of mistakes described by the DBAs, namely mismatches between applications and the database structure, we combine model-based validation of the DBMSs with trace-based validation of the applications to check that the applications have been updated to correctly deal with the new schema.

In detail, the whole validation process for actions that change the database structure proceeds as follows. First, a script moves the DBMS to the validation slice, asks the DBA to describe his/her intended actions in canonical form, extracts the current schema from the DBMS, and then allows the DBA to act on the masked DBMS. Once the DBA completes the necessary actions, a second script uses model-based validation to check that the corresponding assertions hold. Finally, the DBA brings any application that depends on the database into the validation slice and validates that it works correctly with the new schema using a trace. In the context of the 3-tier Internet service shown in Figure 5, this means that the DBA would move each of the application nodes into the validation slice for validation.

There are three subtle issues that must be addressed when the database schema changes. To make the description of the issues concrete, suppose that we have a service as in Figure 5. First, if a database schema change requires changes to the application servers, then once updated, these servers cannot be returned to the online slice until the new DBMS has been deployed. Second, it may not be possible to properly validate replies from the application servers against replies that were previously logged using a strict comparator, such as exact content matching, when the application servers need to change. Replica-based validation can be used but only after the changes to at least one application server have been validated. Third, during DBMS reintegration into the online system, it may not be possible to replay writes that have been executed on the online system while the masked DBMS was being changed and validated (e.g., writes that depend on the data in a column that has been removed).

To deal with the first issue, the validation of the application servers and DBMSs needs to occur in two phases as follows. During a period of low load, the DBA can move one DBMS into the validation slice, change the schema, use model-based validation to check the correctness of his/her changes, then move 1/2 of the application servers over to the validation slice, update them as necessary, and use trace-based validation to check that they work correctly with the modified DBMS. After validation is completed, he/she can temporarily halt and buffer requests from the first tier, move all validated components back online, and move the remaining unmodified application servers and DBMS into the validation slice. In essence, this is the point where the live service is changed from operating on the old database schema to the new schema.

To deal with the second issue, we observe that it is possible to validate the application servers using traces collected previously in situations where we know that the changes in schema should not cause SQL fatal errors in the application servers. In this case, a previous trace can be used together with a comparator function that disregards the content of the application server replies but checks for fatal SQL exceptions. (In fact, replica-based validation could also be applied using this weaker comparator function.) In case an exception is found, the validation process fails. When schema changes should cause these exceptions in application servers, a synthetic application server trace needs to be generated that should not cause exceptions with the new schema. Again, an exception found during the validation process would mean a potential DBA mistake.

Finally, to deal with the third issue, our implementation denies writes to the online virtual DBMS when the database structure needs to be changed. This behavior is acceptable for the system we study (an online auction service). To avoid denying writes, an alternative would be to optimistically assume that writes that cannot be replayed because of a change in schema will not occur. In this approach, the C-JDBC controller could be modified to flag an error during reintegration, if a write cannot be replayed. If such an error occurred, it would be up to the DBA to determine the proper course of action.

4.4 Summary

In summary, we find that our three validation strategies are complementary. Trace-based validation can be used for checking the correctness of actions for corner cases that do not occur frequently. Replica-based validation can be used to place the most realistic workload possible on a masked component — the current workload of

the live system. Trace-based and replica-based validation allow the checking of performance tuning actions, such as the creation or modification of indexes. Finally, structural changes can be validated using a combination of model-based and trace-based validation.

With respect to the mistakes reported by DBAs, validation in an extension of the online system allows us to eliminate deployment mistakes, whereas trace, replica, and model-based validation deal with performance tuning and structural mistakes. These three categories of mistakes are the most frequent according to our survey.

5 Evaluation

In this section, we first evaluate our validation approach using a set of mistake-injection benchmarks. We then assess the performance impact of our validation infrastructure on a live service using a micro-benchmark.

5.1 Experimental Setup

Our evaluation is performed in the context of an online auction service modeled after EBay. The service is organized into 3 tiers of servers: Web, application, and database tiers. We use one Web server machine running Apache and three application servers running Tomcat. The database tier comprises one machine running the C-JDBC controller and two machines running back-end MySQL servers (that are replicas of each other within a single C-JDBC virtual DBMS). All nodes are equipped with a 1.2 GHz Intel Celeron processor and 512 MB of RAM, running Linux with kernel 2.4.18-14. The nodes are interconnected by a Fast Ethernet switch.

A client emulator is used to exercise the service. The workload consists of a "bidding mix" of requests (94% of the database requests are reads) issued by a number of concurrent clients that repeatedly open sessions with the service. Each client issues a request, receives and parses the reply, "thinks" for a while, and follows a link contained in the reply. A user-defined Markov model determines which link to follow. During our mistake-injection experiments, the overall load imposed on the system is 60 requests/second, which is approximately 70% of the maximum achievable throughput. The code for the service, the workload, and the client emulator are from the DynaServer project [19].

5.2 Mistake-injection Experiments

We injected DBA mistakes into the auction service. Specifically, we have developed a number of scripts, each of which emulates a DBA performing an administration task on the database tier and contains one mistake that may occur during the task. The scripts are motivated by our survey results and span the most commonly reported tasks and mistake types. The detailed actions and mistakes within the tasks were derived from several database administration manuals and books, e.g. [15]. Table 2 lists the mistakes we injected in our experiments categorized by DBA task and problem (see Section 3). Note that we only designed mistakes for problem categories where at least some problems were reported as originating from DBA mistakes (those marked with some, most, or all for "Caused by DBA mistakes" in Table 1).

Problem Category	# Mistakes Injected	# Mistakes Caught
Deployment	4	4
Performance	2	1
General structure	6	5
DBMS	0	–
Access-privilege	2	1
Space	1	1
General maintenance	6	5
Hardware	0	–
Data	2	2

Table 3: *Coverage in mistake-injection experiments.*

Table 3 lists the total number of scripts (mistakes) for each problem category reported in our survey and the number of mistakes caught by validation. Overall, validation detected 19 out of 23 injected mistakes.

It is worth mentioning how validation caught the performance mistake "insufficient number of indexes" and the deployment mistake "indexes not reapplied". In both cases, a performance degradation was detected by the performance comparator function. The comparator function uses two configurable thresholds to decide on the result of validation: the maximum acceptable execution time difference for each request (set to 60 seconds in our experiments), and the maximum tolerable execution time difference accumulated during the whole validation (set to 30 seconds times the number of requests used in the validation). In our experiments, which were configured to execute 10,000 requests for validation, the absence of an index caused the execution time difference for 13 requests to be greater than 70 seconds. The average time difference among these 13 requests was 32 minutes, and the total time difference over all 10,000 requests was 7.5 hours. Note that validation could have stopped long before the 10,000 requests were executed. However, to see the complete impact of the mistakes, we turned off a timeout parameter that controls the maximum time that each request is allowed to consume during validation.

Our implementation of model-based validation caught 4 out of 5 mistakes from the General structure category for which model-based validation is applicable. We did not catch the "insufficient column size assumed by

Task Category	Problem Category	Details of Mistakes Injected Using Scripts
DB Structure	Deployment	Mismatch in schema used by application and actual database schema
		Unintended modifications to the database schema
		Indexes not reapplied after modifications to structure
	General structure	Insufficient column size assumed by database schema
		Wrong table dropped
		New column given an incorrect name
		Wrong column removed
		Name of existing table incorrectly changed
		Deadlocks caused by erroneous application programming
	Access-privilege	Access to certain tables not granted
		Excess privileges granted
Space Management	Space	Misconfigured autoextension parameter: maximum data file size too small
	Deployment	Incomplete data reimport during a defragment operation
	General maintenance	Data file parameters incorrectly configured: size and path (2 mistakes)
		Log and data files mistakenly deleted (2 mistakes)
Software Upgrade	General maintenance	Incorrect data reloaded post-upgrade, e.g., due to mistakes in transforming data during migration to a different database (from Oracle to MySQL)
Performance Tuning	Performance	Misconfiguration of buffer pool: size is too small
		Insufficient number of indexes
Data Modification	Data	Unchecked data loss/corruption resulting in loading of incomplete/incorrect data into production system (2 mistakes)
Recovery	General maintenance	Incomplete backup due to erroneous backup scripts or inattentive space management, resulting in incorrect/incomplete data during recovery

Table 2: *Operator mistake fault load used in evaluating validation.*

database schema" mistake because it currently does not include the notion of size. However, this can easily be added to a more complete implementation. The 6th structural mistake, "deadlocks caused by erroneous application programming," could not have been caught by model-based validation because the mistake occurs at the application servers. This mistake was caught by multi-component trace-based validation, however.

We considered the performance mistake "buffer pool size too small" not caught. In our experiment, we changed the MySQL buffer pool size from 256 MB to 40 MB. Regarding its performance impact, the execution time difference for 7 out of 10,000 requests was greater than 1.5 seconds. Had the threshold been set to at most 1.5 seconds, validation would have caught this mistake. This highlights how important it is to specify reasonable thresholds for the performance comparator function.

The second mistake that validation could not catch was "excess privileges granted", a latent mistake that makes the system vulnerable to unauthorized data access. Validation is not able to deal with this kind of situation because the live (or logged) requests used to exercise the masked components cannot be identified as illegitimate once their authentication has succeeded.

The last mistake that validation overlooked was "log files mistakenly deleted". The reason is that the DBMS did not behave abnormally in the face of this mistake.

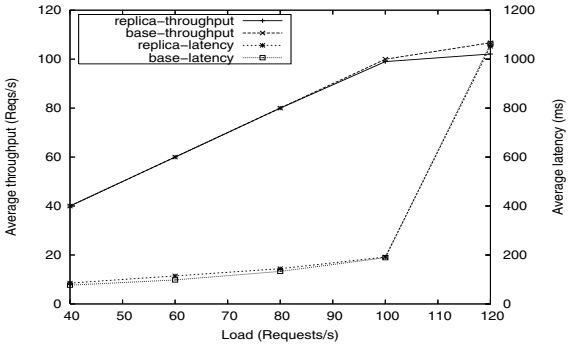

Figure 6: *Performance impact of validation.*

5.3 Performance Overheads

Having shown that validation is effective at masking database administration mistakes, we now consider the performance impact of validation on our auction service.

Shunting. We start by considering the overhead of shunting C-JDBC requests and replies. To expose this overhead, we ran the two back-end database servers on more powerful machines (2.8-GHz Xeon-based machines, each with at least 1 GByte of memory and a 15K-rpm disk) in these experiments.

Figure 6 depicts the average service throughputs (left axis) and average request latencies (right axis) for a system performing replica-based validation and a base sys-

tem that does not shunt any requests or replies, as a function of the offered load. The throughputs and latencies are measured at the client emulator. Note that we do not present results for trace-based validation, since the overhead of logging requests and replies is smaller than that of forwarding them across the isolation barrier.

These results show that the overhead of replica-based validation is negligible in terms of request latency, across the entire range of offered loads. With respect to throughput, the overhead is also negligible until the C-JDBC controller approaches saturation at 120 requests/second; even at that point, the throughput loss is only around 5%. The loss is due to the additional CPU utilization caused by forwarding. In more detail, we find that forwarding imposes an additional 6–10% to the CPU utilization at the controller, across the range of offered loads (logging imposes 3–5% only). In contrast, shunting imposes a load of less than 1 MB/second on network and disk bandwidth, which is negligible for Gigabit networks and storage systems.

Database state handling. We also measured the overhead of preparing a masked DBMS to undergo validation and the overhead of bringing the masked DBMS back online after validation completes successfully. As discussed below, these overheads do not always affect the performance of the online processing of requests.

When using replicas, the time it takes for the operator to act on the masked DBMS affects the overhead of preparing the DBMS for validation, which is essentially the overhead of resynchronizing the online and masked DBMSs. For example, assuming the same infrastructure as the experiments above, 60 requests/second, and 10 minutes to complete the operator's task, we find that preparing a masked DBMS to undergo replica-based validation takes 51 seconds. For a task taking 20 minutes, preparing the masked DBMS takes 98 seconds. These overheads are directly related to the percentage of requests that induce database writes in our workload (6%). During resynchronization, neither the average throughput nor the average latency of online requests is noticeably affected. However, resynchronization does incur an additional 24–31% of average CPU utilization on the controller. After a successful replica-based validation, reintegration of the masked DBMS takes only milliseconds, since the two DBMSs are already synchronized.

When using traces, the overhead of preparing a masked DBMS for validation is not affected by the length of the operator task. Rather, this overhead is dominated by the time it takes to initialize the masked DBMS with the database state stored in the trace. For our 4GB auction database, this process takes 122 seconds. This overhead has no effect on the online requests, since it is only incurred by the masked DBMS. After a successful trace-based validation, the overhead of reintegration is dominated by the resynchronization with the online DBMS. Resynchronization time essentially depends on how long trace replay lasts, leading to similar overheads to preparing a masked DBMS for replica-based validation. For example, if replay lasts 10 minutes, reintegration takes around 51 seconds. The impact of resynchronization on the processing of online requests is also the same as in replica-based validation.

Summary. Overall, these results are quite encouraging since the overheads we observed only impact the C-JDBC controller and only while validation is taking place. Furthermore, services typically run at mid-range resource utilizations (e.g., 50%–60%) to be able to deal with load spikes, meaning that the CPU overhead of validation should not affect throughputs in practice. Operating on the database during periods of low load reduces the potential impact of validation even further.

6 Discussion

In this section, we draw several interesting observations from our experience with DBA mistakes and database validation, as well as relate our findings to our previous validation work [16] on Web and application servers.

First, our survey clearly shows that most DBA mistakes are due to the separation and differences between online and testing environments. We believe that keeping these environments similar (ideally equal) is more difficult for database systems than for Web and application servers. The reason is that the amount of state that would need to be replicated across the environments can be orders of magnitude larger and more complex in the case of databases. This observation leads us to believe that deployment and performance mistakes will always be more common in database systems; structure mistakes have no clear equivalent in the context of Web and application servers. In contrast, configuration mistakes that are dominant in the latter systems are not so frequent in databases.

Second, it is clear also that DBA support tools can help database administration. However, these tools are very specific to DBMS and to the tasks that they support. We believe that validation (or a validation tool) is more generally applicable and thus potentially more useful. The negative side is that a validation tool on its own would not substantially reduce the amount of work required of the DBA; instead, it would simplify deployment and hide any mistakes that the DBA might make.

Third, we found that implementing validation for database systems is substantially more complex than doing so for Web and application servers. There are 3 reasons for the extra complexity: the amount of state in-

volved, the type of replication across DBMSs, and the consistency requirements of the state. The amount of state has implications on performance and request buffering, since requests need to be blocked during certain state management operations. Further, since database systems deal essentially with hard state, replication and state management have to maintain exact database replicas online. Related to the hard state, the strong consistency requirements of ACID force requests to be replayed in exactly the same order at the replicated databases. Strong consistency imposes extra constraints on how requests can be forwarded to (or replayed in) the validation slice. In contrast, Web and application servers involve relatively small amounts of soft state, do not require exact replication (functionality replication is enough), and only require the replicated ordering of the requests within each user session (rather than full strong consistency).

Fourth, we observe that making database structure changes and performing model-based validation without blocking any online requests is a challenge (one that C-JDBC does not address at all). The problem is that, during validation, the SQL commands that arrive in the online slice might actually explicitly refer to the old structure. When it is time to reintegrate the masked DBMS, any write commands that conflict with the new structure become incorrect. We did not face this problem in our previous work, as operations that would correctly change the behavior of the servers were not considered.

Finally, despite the above complexities, we believe that validation is conceptually simple to apply across different classes of systems. The key requirements are: (1) a component to be validated must have one or more replicas that are at least functionally equivalent to the component; (2) the system must be able to correctly adapt to component additions and removals; and (3) the system must support a means for creating a consistent snapshot of its state. For a system with these requirements, validation involves isolating slices, shunting requests and replies, implementing meaningful comparator functions, and managing state.

7 Conclusions

In this paper, we collected a large amount of data on the behavior of DBAs in the field through a survey of 51 experienced DBAs maintaining real databases. Based on the results of our survey, we proposed that a validation infrastructure that allows DBAs to check the correctness of their actions in an isolated slice of the online system itself would significantly reduce the impact of mistakes on database performance and availability. We designed and implemented a prototype of such a validation infrastructure for replicated databases. One novel aspect of this infrastructure is that it allows components of a replicated database to be acted upon and validated while the database itself remains operational. We also proposed a novel validation strategy called model-based validation for checking the correctness of a component in the absence of any known correct instances whose behaviors can be used as a basis for validation. We showed how even a simple implementation of this strategy can be quite powerful in detecting DBA mistakes. We also showed that validation is quite effective at masking and detecting DBA mistakes; our validation infrastructure was able to mask 19 out of 23 injected mistakes, where the mistakes were designed to represent actual problems reported in our survey.

We now plan to explore model-based validation further, not only in the context of database systems but also for other systems, such as load balancers and firewalls.

Acknowledgments

We would like to thank Yuanyuan Zhou and the anonymous reviewers for comments that helped improve the paper. The work was partially supported by NSF grants #EIA-0103722, #EIA-9986046, #CCR-0100798, and #CSR-0509007.

References

[1] BARHAM, P., ISAACS, R., MORTIER, R., AND NARAYANAN, D. Magpie:Real-Time Modelling and Performance-Aware Systems. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)* (May 2003).

[2] BARRETT, R., MAGLIO, P. P., KANDOGAN, E., AND BAILEY, J. Usable Autonomic Computing Systems: The Administrator's Perspective. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC'04)* (May 2004).

[3] BROWN, A. B. *A Recovery-oriented Approach to Dependable Services: Repairing Past Errors with System-wide Undo*. PhD thesis, Computer Science Division, University of California, Berkeley, 2003.

[4] BROWN, A. B., AND PATTERSON, D. A. Undo for Operators: Building an Undoable E-mail Store. In *Proceedings of the 2003 USENIX Annual Technical Conference* (June 2003).

[5] CASTRO, M., AND LISKOV, B. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI'00)* (Oct. 2000).

[6] CASTRO, M., AND LISKOV, B. BASE: Using Abstraction to Improve Fault Tolerance. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)* (Oct. 2001).

[7] CECCHET, E., MARGUERITE, J., AND ZWAENEPOEL, W. C-JDBC: Flexible Database Clustering Middleware. In *Proceedings of the USENIX Annual Technical Conference, Freenix track* (June 2004).

[8] CHEN, M. Y., ACCARDI, A., KICIMAN, E., LLOYD, J., PATTERSON, D., FOX, A., AND BREWER, E. Path-Based Failure and Evolution Management. In *Proceedings of the International Symposium on Networked Systems Design and Implementation (NSDI'04)* (Mar. 2004).

[9] CHEON, Y., AND LEAVENS, G. T. A Runtime Assertion Checker for the Java Modeling Language (JML). In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02)* (June 2002).

[10] GIL, P., ARLAT, J., MADEIRA, H., CROUZET, Y., JARBOUI, T., KANOUN, K., MARTEAU, T., DURES, J., VIEIRA, M., GIL, D., BARAZA, J.-C., AND GRACIA, J. Fault Representativeness. Technical Report IST-2000-25425, Information Society Technologies, June 2002.

[11] GRAY, J. Why do Computers Stop and What Can Be Done About It? In *Proceedings of 5th Symposium on Reliability in Distributed Software and Database Systems* (Jan. 1986).

[12] GRAY, J. A Census of Tandem System Availability Between 1985 and 1990. *IEEE Transactions on Reliability 39*, 4 (Oct. 1990).

[13] KALBARCZYK, Z. T., IYER, R. K., BAGCHI, S., AND WHISNANT, K. Chameleon: A Software Infrastructure for Adaptive Fault Tolerance. *IEEE Transactions on Parallel and Distributed Systems 10*, 6 (1999).

[14] LI, X., MARTIN, R. P., NAGARAJA, K., NGUYEN, T. D., AND ZHANG, B. Mendosus: A SAN-Based Fault-Injection Test-Bed for the Construction of Highly Available Network Services. In *Proceedings of 1st Workshop on Novel Uses of System Area Networks(SAN-1)* (Jan. 2002).

[15] MULLINS, C. S. *Database Administration: The Complete Guide to Practices and Procedures*. Addison-Wesley, 2002.

[16] NAGARAJA, K., OLIVEIRA, F., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)* (Dec. 2004).

[17] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. Why do Internet Services Fail, and What Can Be Done About It? In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'03)* (Mar. 2003).

[18] PERL, S. E., AND WEIHL, W. E. Performance Assertion Checking. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Dec. 1993).

[19] RICE UNIVERSITY. DynaServer Project. http://www.cs.rice.edu/CS/Systems/DynaServer, 2003.

[20] VIEIRA, M., AND MADEIRA, H. A Dependability Benchmark for OLTP Application Environments. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB 2003)* (Sept. 2003).

[21] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using Model Checking to Find Serious File System Errors. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)* (Dec. 2004).