Do you know how to configure your enterprise relational database to reduce incidents?

Ioana Giurgiu, Adela-Diana Almasi and Dorothea Wiesmann
IBM Research - Zürich, Säumerstrasse 4, CH-8803 Rüschlikon, Switzerland

***Abstract*—With the advancement of relational databases, the number of configuration parameters that control memory allocation, concurrency, cost of query plans, I/O optimization, logging, recovery or transaction consistency, increases. Users and even expert database administrators struggle to tune these parameters in order to ensure high availability and performance, and in many cases rely on their experience and some rules of thumb. Research on improving database manageability has shown that this is a critical, but hard problem. In this paper, we propose a highly accurate multivariate statistical model that identifies databases which are bound to raise high volumes of incidents over time. Moreover, we show that by adding detailed configuration parameters to the model, we can better link the problems reported in incident tickets to specific poor database configurations. Finally, we analyze trends of top-ranked parameters and compare their values between problematic and non-problematic databases, in order to suggest better configurations.**

I. Introduction

Consider the following real-life scenario from a large business enterprise that provides financial services. The server administration team is in charge, among others, of maintaining the company's Web site, which the clients use to run their personal financial transactions. Over the past hours, the site has become sluggish, leading to the delay or even the inability to execute client transactions. The administrators collect monitoring data and track the problem down to poor performance of queries issued against one of the back-end databases. Realizing that the database in question needs re-tuning, the team tries to identify the volume of reads and writes to the database and input this information into the database tuning advisor [1], [2], [3]. Not surprisingly, since the database has already been designed based on a previous invocation of the advisor, the new recommendation fails to alleviate the problem.

Such situations appear often appear in practice. Enterprise database management systems (DBMS) provide many configuration parameters, that allow for fine tuning. However, due to lack of in-depth expertise and knowledge of the expected workloads, most teams set these parameters to default values. In the scenario above, the database administrators find dozens of configuration parameters that highly impact the query optimizer's cost model, such as buffer pool sizes, number of concurrent transactions, amount of heap memory for sorting and so on. Even for the best of experts, tuning these parameters is extremely difficult, due to the exploding number of possible combinations. Moreover, what appears to be an application-related performance issue can have system implications.

Tuning database configuration parameters is hard, but critical, since bad settings can be orders of magnitude worse in performance than good ones. Changes to some parameters cause local and incremental effects on resource usage, while others cause changing of query plans or shifting bottlenecks from one resource to another. These effects vary depending on the hardware platforms, workload, and configuration schemas. Groups of parameters can have non-independent effects, e.g., the performance impact of changing one parameter may vary based on different settings of another parameter.

In this paper, we learn from past experiences and propose a new approach to identifying databases that due to their poor configurations are bound to generate high volumes of incidents over time. More specifically, we focus on two major DBMS, namely IBM DB2 [4] and MSSQL [5]. We describe two database family-specific multivariate statistical models that consider both server and database configuration parameters, linked to database incident tickets, in order to classify server - database pairs into problematic and non-problematic. Each model is implemented as a hierarchical set of models, which evolve in complexity by adding more database configuration features as we go along. The contributions of this paper are multi-fold. First, we show that by employing a classic random forest model, we are able to identify problematic server - database pairs with high accuracy. Second, we show that by adding detailed database configuration parameters, the model performance increases significantly. This is due to the fact that we are able to better link the problems reported in incident tickets to the actual root cause, residing in the way databases are configured. Finally, we identify relevant shifts in specific configuration parameters between problematic and non-problematic databases. These findings allow us to suggest desirable ranges of values for such parameters, in order to reduce or even avoid future severe incidents.

The remainder of the paper is organized as follows. In Section II, we describe the data collection process. Section III presents several examples of database configurations to avoid and potential solutions. In Section IV, we provide an overview of our statistical models, while Section V presents preliminary results in terms of model accuracy and feature importance. Finally, we compare and contrast our method with existing approaches in Section VI, and conclude in Section VII.

II. Data collection

We collect database incident tickets, server and database configuration properties from 4 IT environments. The incident tickets span 19 months (January 2013 to July 2014). For both server and database parameters, we use the latest snapshots extracted by the database management service line teams.

Database incident tickets – We gather the incident severity, opening and closing timestamps, responsible support team, as well as detailed description and resolution. Then, we classify tickets into 7 classes, as described in Table I. For each IT environment we use the incidents related to backup, database errors and tablespace issues, as well as disk space

TABLE I: Incident failure classes based on root cause and their distributions across the 4 IT environments.

Class	Description	% in IT1	% in IT2	% in IT3	% in IT4
Backup	Backup issues (e.g., scheduled backup has failed, backup cronjob has errors)	3%	5.2%	2.4%	3.4%
DB error	Various database problems, except tablespace issues (e.g., database is crashing, database performance is low, maximum number of connections exceeded, transactions are hanging)	26.5%	1.7%	4.7%	0.2%
DB tablespace	Tablespace issues (e.g., DMS tablespace too small)	3.7%	2.6%	0.2%	0.2%
Disk space	Incidents generated by the used disk space exceeding threshold (e.g., 90% space used in /opt)	20%	2.8%	11.3%	12.2%
Performance	Any performance degradation incidents on the server	0.6%	¡0.1%	0.9%	0.4%
Other	Incidents not belonging to any other category or with incomplete descriptions and resolutions	2.8%	7.6%	14.3%	6.2%
Diagnostics	Transient events that require diagnostics (e.g., temporary CPU spikes)	43%	80%	66%	77%
% incidents used		54.2%	12.4%	19.7%	16.8%
Incidents used		11807	2611	8233	11322

and performance problems. The dataset contains 33973 tickets, spread across the IT environments as shown in the last row of Table I. Further, we expand the dataset with server properties, by extracting hostnames from incident tickets. Specifically, the server features we consider are the server purpose, age (i.e., calculated from the technology availability date) and architecture, as well as operating system family and version. The total number of servers linked to the tickets is 1052.

Database configurations – We focus on 2 major relational enterprise databases, IBM DB2 and MSSQL. These, besides Oracle [6], are used by IT environments, due to their multiple benefits: (1) *strategic data management*, (2) *multi-user capabilities*, (3) *better reporting capabilities*, (4) *concurrency control*, (5) *backup and recovery*, and (6) *dedicated support in case of failures*. However, enterprise databases also come with several disadvantages: (1) *complexity*, (2) *size*, (3) *lower performance for specific types of workloads*, (4) *higher maintenance cost*, and (5) *higher impact of failures*, due to resource centralization which leads to increased vulnerability. In this paper, we focus on complexity. Provisioning the functionality expected from a good DBMS makes the software itself very complex. Database designers, developers and administrators must understand this functionality to take full advantage of it, since failure to understand the system can lead to bad design decisions. One of the means of providing good functionality is through configuration parameters, which we refer to next.

1) Instance-level parameters – These parameters specifically refer to database family (e.g., IBM DB2, MSSQL) and version. Across the 4 IT environments, DB2 instances represent 63% (i.e., 651), with the remaining 37% (i.e., 395) being associated to MSSQL. MSSQL instances are distributed between 4 versions: 2000 (7% of all MSSQL instances), 2005 (36%), 2008 (18%), and 2008 R2 (39%). DB2 instances are more diverse in terms of versions. Our dataset is distributed across 9 versions, as follows: 6.1 (0.3%), 7.1 (1%), 7.2 (0.15%), 8.1 (13.5%), 8.2 (3%), 9.1 (8%), 9.5 (21.5%), 9.7 (45%), and 10.1 (5.5%).

2) Database-level parameters – These parameters specifically refer to how an actual database, running within an instance, is configured. Given that each database family has its own configuration parameters, the set of parameters collected from DB2 differs from the one for MSSQL. In Table II, we summarize the configuration parameters collected for both database families. 7 parameters out of the 34 are common to both DB2 and MSSQL. More specifically, these are *db_size*, *appl_heap_size*, *dft_degree*, *num_connections*, *num_partitions*, *log_fil_size* and *lock_timeout*. Additionally, we classify all parameters in 6 categories, based on their usage for database tuning: (1) CO = Configuration, (2) ME = Memory, (3) OP = Optimizer, (4) TC = Transaction Consistency, (5) RR = Rollback and Recovery, and (6) CC = Concurrency. ME parameters are related to how memory is allocated for sort operations, statement and utility heaps, as well as the database total heap size and memory threshold. OP parameters are used in configuring the DBMS to choose a good query plan from a multitude of plans for answering a query. TC parameters are used in enforcing the guarantee that database constraints are not violated, once a transaction commits. RR parameters indicate whether a rollback or recovery operation is necessary and facilitate the process of returning the database to a previous state by keeping history logs of the executed transactions. Both rollbacks and recoveries are crucial to ensure database integrity and consistency. Finally, CC parameters are used to set the concurrency level, namely to ensure that correct results are generated for all parallel operations as soon as possible.

The set containing the above configuration parameters is smaller than the one with instance-level properties only. More specifically, for DB2, the dataset contains 502 instances, running 1567 databases, belonging to environments IT2-4. For MSSQL, the set contains 211 instances, running 408 databases, belonging to IT3-4. The decrease in dataset size is due to two reasons. First, the database management teams responsible with the collection do not have access to environment IT1. Second, IT2 exclusively runs DB2.

III. The case for database manual tuning

A sound statistical method for quantifying the impact of configuration parameters and their interactions on the database performance is to apply a full factorial design. That is to consider every combination of input values of the configuration parameters. However, the major problem of this approach is the large number of configuration parameters that enterprise databases come equipped with. For example, both IBM DB2 and MSSQL each have over 100 configuration parameters, and all of them can have tens, hundreds or even millions of values. In the simplest case where each parameter can only take one of two values, full factorial design assumes at least 2^{100} experiments per each query workload. This is clearly not feasible in practice, and so database administrators rely on their experience and rules of thumb to select appropriate parameters for tuning. Nonetheless, a large effort may be wasted on trying to improve database performance by tuning parameters that have little or no effect on performance. In the remainder of this section, we give a few examples of configurations to avoid and also how the corresponding configuration parameters should be set to reduce incidents and boost database performance.

Use case 1 – lock_timeout. Setting this parameter to -1, which happens more often than not, leads to the application to freeze if lock-wait is encountered. We find several MSSQL

TABLE II: Summary of the configuration parameters for DB2 and MSSQL.

Parameter	Class	Description	Family
db_size	–	Database size in GB	MSSQL, DB2
$page_size$	CO	Size of any page in the database	MSSQL
$buff_page$	CO	Size of the database buffer pool	DB2
$appl_heap$	ME	Number of private memory pages available to be used	MSSQL, DB2
$appl_memory$	ME	Controls the maximum amount of memory allocated by the database to service application requests	MSSQL
$sort_heap$	ME	Maximum number of private memory pages to be used for private sorts, or the maximum number of shared memory pages to be used for shared sorts	DB2
$stmt_heap$	ME	Size of the statement heap used during compilation of an SQL statement.	DB2
$util_heap$	ME	Maximum amount of memory that can be used simultaneously by the BACKUP, RESTORE and LOAD utilities	DB2
db_heap	ME	Size of database heap (contains control block information for tables, indexes, table spaces and buffer pools)	DB2
$dbmem_thresh$	ME	Maximum percentage of committed, but currently unused, database shared memory that the database manager will allow before starting to release committed pages of memory back to the operating system	DB2
dft_degree	OP	Specifies the default value for the CURRENT DEGREE special register and the DEGREE bind option	MSSQL, DB2
max_appls	CC	Specifies the maximum number of concurrent applications that can be connected to a database	MSSQL
avg_appls	CC	Determines how much buffer pool will be available at run-time for the access plan chosen	DB2
$log_primary$	RR	The fixed amount of storage allocated to the recovery log files	DB2
log_second	RR	Number of secondary log files that are created and used for recovery log files (only as needed)	DB2
$num_ioservers$	CO	Number of I/O servers used to perform prefetch I/O and asynchronous I/O by utilities, such as backup and restore	DB2
$num_dbbackups$	CO	Specifies the number of database backups to retain for a database	DB2
$num_connections$	CC	Number of external connections supported by the database	MSSQL, DB2
$num_partitions$	CC	Number of partitions in which the database is distributed	MSSQL, DB2
log_fil_size	RR	Size of each primary and secondary log file	MSSQL, DB2
log_buf_size	RR	Amount of the database heap to use as a buffer for log records before writing these records to disk	DB2
$lock_timeout$	TC	Number of seconds to wait to obtain a lock	MSSQL, DB2
min_commit	TC	Minimum number of group commits to be performed prior to writing to disk	DB2
$page_corruption$	RR	Indicates how many database pages are corrupted	MSSQL
$dft_queryopt$	OP	Used to direct the optimizer to use different degrees of optimization when compiling SQL queries	DB2
seq_detect	OP	Flags whether the database manager is allowed to detect sequential page reading during I/O activity	DB2
$dlchktime$	CO	Time interval for checking whether any deadlocks exist on the database	DB2
$backup_compression$	OP	Indicates whether backups are to be compressed	MSSQL
$db_consistent$	RR	Indicates whether the database is in a consistent state	MSSQL
$db_autoshrink$	OP	Indicates whether the database can reclaim any space it takes	MSSQL
$auto_restart$	RR	Indicates whether the auto restart option is enabled	DB2
$backup_pending$	RR	Indicates whether a backup operation is pending	DB2
$restore_pending$	RR	Indicates whether a restore operation is pending	DB2
$rollfwd_pending$	RR	Indicates whether or not a roll-forward recovery is required	DB2

and DB2 databases in our dataset for which lock_timeout is set to -1. This value is recommended only for development environments, to allow the DBA to identify and resolve lock-wait situations. Depending on the number of concurrent users (i.e., num_connections), it should be set to at least 30 seconds to avoid rollbacks and timeouts due to peak workloads. However, values too high can also lead to stalling transactions.

Use case 2 – num_ioservers. I/O servers are used to perform prefetching and setting this parameter too low or too high will hurt database performance. The recommended value spans from the number of physical disks that the database resides on + 1 to at most 4 to 6 times the number of CPU cores on the server. We find multiple mis-configurations num_ioservers in our dataset, with its value always exceeding 6 * number of CPUs. For instance, for some databases, num_ioservers is set to 255 when the CPUs on the server vary from 7 to 32 cores.

Use case 3 – dft_queryopt. This parameter specifies the level of optimization when compiling SQL queries. Depending on the type of workload, one should choose lower values for everyday queries (i.e., recommended values are 3 or below, with 0 or 1 being suitable for simple SELECT queries) and higher for multi-dimensional, complex queries (i.e., recommended values are between 5 and 7, with 7 being usually used for queries running longer than 30 seconds). We find a few databases ($< 3\%$) that have dft_queryopt set to 9, a value to be avoided under most circumstances in enterprise databases.

Use case 4 – sort_heap. Sort overflows occur when the amount of memory needed for a sort exceeds sort_heap. Of course, in case data statistics are out of date, the DBMS can request too small a sort heap, which also results into overflows. A good minimum size for the sort heap is 128 pages for databases with simple, short queries and between 4096 and 8192 pages in case of complex queries. We find 12% DB2 databases with sort_heap set to less than 128 pages.

Use case 5 – log_buf_size and db_heap. log_buf_size specifies the amount of database heap (db_heap) to use as a buffer for log records prior to writing to disk. Log records get written to disk whenever a transaction commits or the log buffer is full. A larger size for the buffer results in less frequent writes to the disk and more log records being written at one time. As a result, the minimum recommended values are 256 pages for databases running many simple queries and 128 for less frequent queries. When unable to increase the log buffer size, the recommendation is to increase first the database heap. We find several DB2 databases with log_buf_size less than 128 pages, although db_heap is at least 10x higher.

IV. MULTIVARIATE STATISTICAL MODELS FOR DATABASES

In our previous work [7], we have proposed a predictive model for the automatic classification of servers into *problematic* and *non-problematic*. In this paper, we borrow these concepts to build a statistical model for identifying problematic pairs of servers and databases. First, we describe the problem setting and then provide a summary of the multivariate models built with random forest and the performance measures used.

Let S be the set of servers and corresponding databases along with their configuration information and linked database incident tickets. We define a *problematic server - database* pair as exceeding predefined thresholds of incident ticket

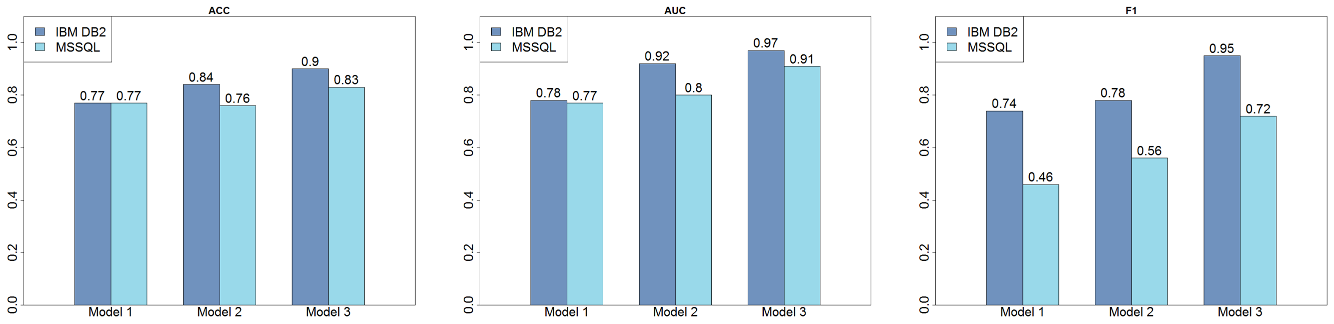

Fig. 1: ACC, AUC and F-score across the DB2 and MSSQL models.

volumes by ticket type and severity. High severity incidents indicate severe database problems that require immediate administration, and thus have a higher overall impact. The goal of our multivariate statistical model is to automatically infer such problematic server - database pairs from server and database configuration parameters. To achieve this, we formally represent each such pairs by vectors of server and database features $\mathtt{sd} \in \mathtt{S}$ and input it in the predictive model $\mathtt{M}$. Once trained on the available set of pairs, $\mathtt{M}$ associates a probability for each $\mathtt{sd}$, where $\mathtt{M(sd)} \in [0,1]$. All problematic server - database pairs will have an associated $\mathtt{M(sd)} > 0.5$.

Multivariate models – Random forests [8] are ensemble learning methods used for classification and regression. They operate by constructing a multitude of decision trees at training time and outputting the class most common among the individual trees, which are fitted using the CART procedure [9]. The benefits of using random forest models are multi-fold. First, the generalization error can be estimated by computing the *out-of-bag* (OOB) error during the fitting phase. Second, the model measures the variable importance through permutation, which allows us to understand how various features (i.e., server and database configuration parameters) impact the condition of a server - database pair of being problematic. Third, they are able to capture nonlinear relations between input and output variables, and are robust against outliers. We refer the reader to [7] for detailed discussions on choosing random forest against linear models. The remainder of this section describes the DB2 and MSSQL models.

Since configuration parameters differ from one database family to another, we cannot build a model that fits all. Instead, we build separate models for DB2 and MSSQL. Both models are built as a 3-layered set of models, that add in complexity by progressively including additional configuration parameters. We describe the features considered for each model next.

Model 1. We start with a general model that considers 6 features: server purpose, age and architecture, OS family and version, as well as database version. Since this information is available for all 4 IT environments, we run the model on the extended dataset, consisting of 1052 servers. It is important to note that the model classifies servers, and not server - database pairs, because the database version is not database-specific, but instance-specific. Also, the database family feature is excluded, because we run separate models for DB2 and MSSQL.

Model 2. Next, we add 3 database-specific properties, namely database size, number of connections and number of partitions. As discussed in Section II, environment IT1 does not provide such information, and therefore we exclude it from this model run. Given that environment IT2 exclusively uses DB2, the DB2 run considers IT2-4 (502 instances, 1567 databases), while the MSSQL run considers only IT3-4 (211 instances, 408 databases). We note that unlike the first model, we now identify problematic *server - database* pairs.

Model 3. We add the detailed configuration parameters in the third model. Apart from the 9 features considered so far, we include 12 more predictors for MSSQL and 24 for DB2 (as described in Table II) and run for both database families.

Performance measures. To evaluate the performance of the considered models, we use classification accuracy, area under the ROC curve (AUC) and F-score. We note that in the case of large class imbalance in the dataset, accuracy is not the best measure. Instead, we report balanced accuracy, accuracy over the problematic class, accuracy over the non-problematic class, AUC and F-score, defined as follows:

$$Acc^{-} = \frac{\text{True Negatives}}{\text{True Negatives} + \text{False Positives}}$$

$$Acc^{+} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

$$\text{Acc} = \frac{Acc^{-} + Acc^{+}}{2}$$

$$F-score = \frac{2 \cdot \frac{\text{True Positives}}{\text{True Positives+False Positives}} \cdot Acc^{+}}{\frac{\text{True Positives}}{\text{True Positives+False Positives}} + Acc^{+}}$$

V. Evaluation and Discussion

In this section we evaluate the model performance and discuss trends of configuration parameters for problematic and non-problematic server - database pairs.

A. Models performance

First, we evaluate the accuracy of the 3-layered DB2 and MSSQL models. Our assumption is that by adding more database configuration parameters, the model will gain more knowledge in how to correctly identify problematic server - database pairs. Fig. 1 shows that ACC, AUC and F-score increase as we add more predictors. For the F-score, model 3 shows a 21% improvement for DB2 and 26% improvement for MSSQL compared to model 1. Similarly, AUC improves by 19% and 14% for DB2 and MSSQL, and ACC increases by 13% and 6%, respectively.

We note that the DB2 model accuracy is higher compared to MSSQL. This is attributed to two reasons. First, the MSSQL model contains only half the number of configuration

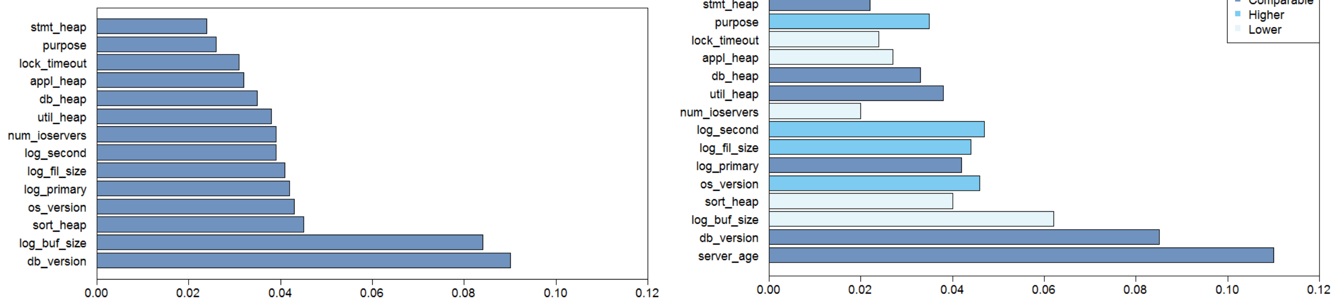

Fig. 2: Predictor importance for DB2 across all (left) and problematic databases (right) in the context of Model 3.

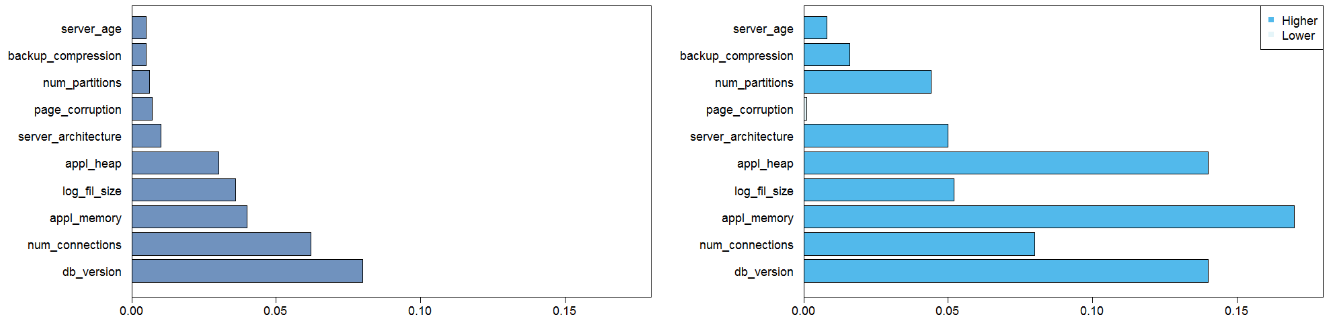

Fig. 3: Predictor importance for MSSQL across all (left) and problematic databases (right) in the context of Model 3.

parameters (i.e., 12 vs. 24 for DB2). Second, the MSSQL databases in our dataset appear to be much more uniform in their configurations, indicating that the parameters driving the occurrences of problems might not be contained in the included set. In comparison, the DB2 dataset shows a greater variation across database settings.

B. Predictors importance

One of the benefits of using random forest is that after the classification step, one can check the predictor importance across all features used. Next, we present the top predictors across DB2 and MSSQL for all 3 models.

Model 1 – On the one hand, over all servers running DB2, the top predictors are server age, followed by database version, OS version and server architecture. However, when looking only at the servers identified as problematic, the highest predictor is the database version, closely followed by the OS version and server age. On the other hand, the order changes across all MSSQL servers. The purpose seems to be most important (i.e., problematic servers are exclusively database nodes, not application, backup or infrastructure), followed by the database version and the server architecture. Considering only the problematic servers, the architecture is by far the highest predictor (2.5x larger than the second ranked feature), then purpose, database version and server age.

Server age for DB2 servers ranges from 1 to 16 years, while over 90% of the MSSQL servers are either 3- or 4-years old, with very few either being less than 2-years old or 5- to 6-years old. This difference between the DB2 and MSSQL servers explains why server age plays a primary role for DB2. Looking at the server purpose, all problematic MSSQL servers are DB infrastructure, while application ones are non-problematic. Similarly for DB2, application and DB servers dominate the problematic set. Next, we focus more closely on the database and OS versions. We expect that older database versions are run mostly on older OS versions, and that they correlate more with incident tickets. On the one hand, this is the case for DB2, as versions below 10 run on Unix and AIX instances that have been released more than 5 years ago. Over 70% such database - OS combinations are classified as problematic. On the other hand, more recent MSSQL versions (i.e., especially service packs of standard versions) seem to be slightly more problematic, although our dataset does not contain any databases running the last 2 releases. Also, all MSSQL databases run on the same OS family and version. This limitation is attributed to the lower variation in our MSSQL dataset, which prohibits us to study the effect of running other MSSQL versions on a mix of OS versions. Finally, we look at server architecture. On the one hand, all DB2 databases run on various IBM architectures. Results show no significant differences between various architectures relative to problematic and non-problematic databases, which correlates well with the predictor importance ranking. On the other hand, MSSQL databases mostly run on IBM System X or HP Proliant architectures, with the latter being somewhat more correlated with incident tickets than the former.

Model 2 – For DB2 databases, the top predictors are server age, OS version, number of connections and server architecture. Unlike for model 1, the database version becomes less important compared to the number of connections, but its importance is still higher than the actual database size and its number of partitions. When looking only at the problematic databases, the number of connections is the top predictor. We

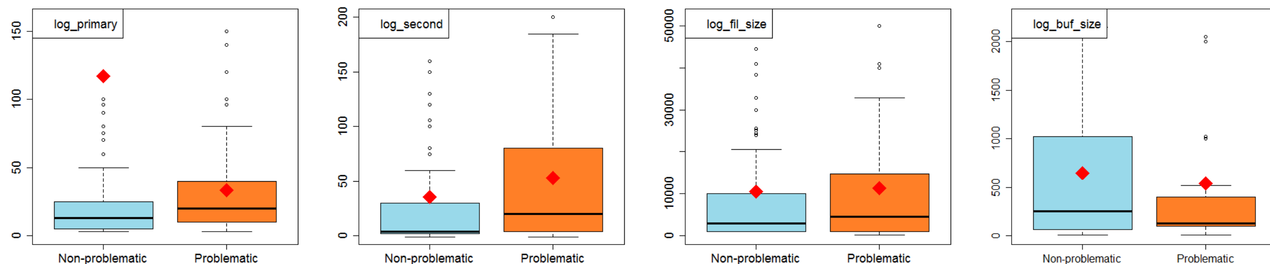

Fig. 4: Comparison between problematic and non-problematic DB2 databases across log_primary, log_second, log_file_size and log_buf_size.

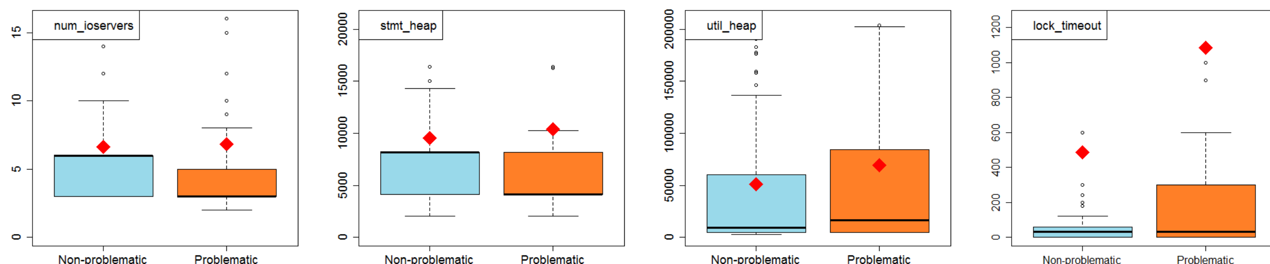

Fig. 5: Comparison between problematic and non-problematic DB2 databases across num_ioservers, stmt_heap, util_heap and lock_timeout.

expect that problematic databases should have a higher number of connections. Indeed, on average, problematic databases have 51 active connections per database, while non-problematic ones only 33. Although this parameter is usually set based on need, higher values correlated with more problems can be an early indication of resource contention (e.g., concurrent write and read operations on the same table) between competing connections. The size and number of partitions are lower ranked because their average values are comparable between problematic and non-problematic databases (e.g., most databases are not partitioned, thus the number of partitions is 1).

For MSSQL databases, the top predictors are size, number of connections, server age, database version and OS version. Looking only at the problematic databases, the server architecture is the top predictor, closely followed by the number of partitions, connections and database size. We compare these 3 features across problematic and non-problematic databases and note the following: (1) problematic databases have on average 51 connections, compared to 36 for the non-problematic ones; (2) the average number of partitions is higher for problematic databases, specifically 1.9 vs. 1.1; (3) problematic databases are larger in size with an average factor of 6-7x.

Model 3 – Fig. 2 shows the importance of the top 15 predictors across all DB2 databases, as well as relative only to the problematic ones. As seen, out of all the server properties, the age, OS version and purpose are ranked within the highest predictors. 5 ME (i.e., sort_heap, util_heap, db_heap, appl_heap and stmt_heap) and 4 RR (i.e., log_file_size and log_buf_size, log_primary and log_second) configuration parameters make a difference. Looking at the predictor importance values for problematic databases, trends vary from those across all databases. The parameters ranked higher for problematic databases are OS version, log_file_size, log_second and server purpose. Those ranked lower are log_buf_size, sort_heap, num_ioservers, appl_heap and lock_timeout. We note that the opposite effect in ranking is seen across the non-problematic databases (i.e., log_buf_size, num_ioservers, appl_heap and lock_timeout have higher importance).

In Fig. 3 we show the ranking of the top 10 parameters across all and problematic only MSSQL databases. Server age has a low importance, although across DB2 it is the leading predictor. The database version is the most important feature, followed by 2 CC, 2 ME and 2 RR configuration parameters. Predictor importance for problematic databases shows a more dramatic shift compared to DB2. Most of the parameters are significantly higher ranked than relative to all MSSQL databases. For instance, the correlations of appl_heap, appl_memory, backup_compression and server architecture are larger by factors of 5x, 4x, 3x, and 5x respectively. The only parameter ranked lower for problematic databases is the page_corruption, specifically by 7x.

The benefits of evaluating predictor importance are twofold. First, we show that a generic model cannot fit all database families. In fact, there are significant differences between which parameters correlate to DB2 problematic databases and which correlate to MSSQL ones. Second, by comparing predictor rankings across all and problematic databases, we identify configurations that are likely to cause incident tickets. Next, we use these observations to study the trends of specific top-ranked configuration parameters for both DB2 and MSSQL.

C. Trends of configuration parameters for DB2

Next, we compare the problematic and non-problematic DB2 databases across the top-ranked configuration parameters (as seen in Fig. 2) and show the results for a subset of them

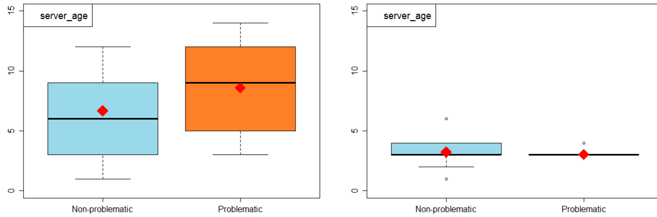

Fig. 6: Comparison between problematic and non-problematic DB2 (left) and MSSQL (right) databases across server age.

in Fig. 6, 4 and 5. One can see that the average age of servers hosting problematic databases is higher, namely 9 years compared to 7 years. This is expected since problematic DB2 databases mostly run on servers at least 8 years old. Regarding log_primary and log_second, we see a mixed picture. First, non-problematic databases have on average 4 times more primary logs. Too few such logs can easily affect the database in performing transactions. Second, the difference across the number of secondary logs is less dramatic, but on average problematic databases have 53 such logs, compared to 36. Our assumption is that the main driver is log_primary, since by the time a database switches to writing to the secondary logs, it would have already reached an erroneous state due to insufficient primary logs. Therefore, it would seem better to set a higher number of allowed primary logs, then of secondary logs. Also related to logging are log_file_size and log_buf_size. Mean and median values across problematic and non-problematic are close enough, with the problematic ones having a higher average for log_file_size (11400 4KB pages vs. 10500 4KB pages) and a lower average for log_buf_size (540 vs. 647). These results are explained as follows. First, log_file_size defines the sizes of primary and secondary logs, where log_primary is the main driver. Second, log_buf_size represents the amount of database heap to use before writing to disk, in the context where both problematic and non-problematic databases have comparable heaps. Fig. 5 shows that on average lock_timeout for problematic databases is 2x larger (1083 vs. 487 seconds). This is a surprising finding, since we expected to see lower lock_timeout values in case of database problems. Looking closer, we note that for 45% of the problematic databases this parameter is set to -1 and for 20% of them the values exceed 600 seconds, while for most non-problematic databases lock_timeout is set between 1 and 600. Therefore, both too small or too large timeouts can lead a database into deadlocks and resource starvation. num_ioservers is comparable across the two sets, although we note a shift towards lower values for problematic databases. Finally, we compare several memory configuration parameters. On average, util_heap is larger for problematic databases by 27% (69328 4KB pages vs. 51192 4KB pages). Although, on a first glance this is surprising, looking at the fact that utility heap is mostly used for recovery and restore operations, the correlation to database incidents becomes clear. stmt_heap is also slightly higher for problematic databases, which means more time is spent in compiling SQL queries and indicates an overall higher usage by external applications. We omit plots for appl_heap, sort_heap and db_heap, because the values across the 2 sets are comparable and the Wilcox tests are not statistically significant ($p > 0.05$). For the other parameters discussed so far, all Wilcox tests are significant ($p < 0.03$).

D. Trends of configuration parameters for MSSQL

Next, we study top-ranked MSSQL configuration parameters. All servers with MSSQL installed run Windows 2008 and 9% of them host at least one problematic database. As discussed before, more recent database versions and specific server architectures seem to be positively correlated with incident tickets. Unlike for DB2, servers hosting MSSQL databases are younger, with ages between 1 and 5 and there is no significant shift between the 2 sets, as shown in Fig. 6.

Finally, we compare the top-ranked configuration parameters across problematic and non-problematic databases (Fig. 7). Problematic databases have on average less overall application memory by a factor of 6, but larger application heaps by 35%. These findings suggest that poor database configurations allocate too much of the application memory for the heap. This is most likely a result of several factors that can eventually lead to low database performance: (1) lack of clustered indexes on the database tables (i.e., indexes impose a logical order of the data, whereas using the heap means the database will store data in any memory location that is available); (2) frequent update and delete operations performed on the heap data; (3) memory fragmentation, which leads to unreclaimed space that often cannot store incoming data. As discussed in Section V-B, non-problematic databases are less distributed, since on average they have 1.1 partitions compared to 1.9, and have less connections (36 vs. 51). Finally, looking at log_fil_size, we note that the log files for problematic databases are on average 51.4KB pages, which is at least twice the size of those for non-problematic databases (e.g., 23 KB pages). Our assumption is that log_fil_size is correlated with other configuration parameters not included in our dataset, which leaves us unable to clearly explain the surprising results. Wilcox tests are statistically significant for all configuration parameters, with the exception of page_corruption, for which we find no significant shifts between problematic and non-problematic databases.

E. Discussion

To sum up, some of our findings for DB2 show that: (1) problematic databases are mostly correlated with older versions of DB2 and the OS; (2) no specific server architecture is a strong indicator of incidents; (3) problematic databases have on average 4x less primary logs, but 50% more secondary logs; (4) lock timeout is set on average 2x higher for problematic databases, which is given by 20% of them having the parameter set to values greater than 600 seconds; (5) the utility and statement heap sizes are larger for problematic databases, while the application, sort and entire database heaps are comparable; (6) problematic databases have more concurrent connections, but a similar number of partitions as non-problematic ones. With MSSQL we find that: (1) more recent database versions in combinations with specific server architectures are associated with higher volumes of incidents; (2) problematic databases have on average less application memory by a factor of 6, but higher application heap size; (3) problematic databases have more concurrent applications connected and in general have more than 1 partition. We conclude that in general problematic

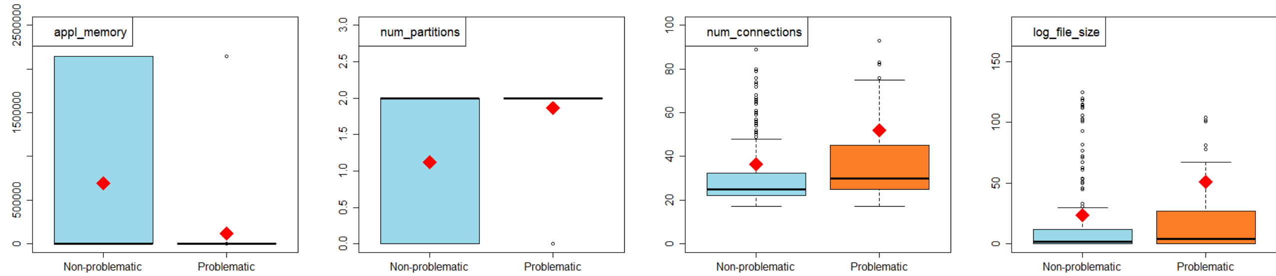

Fig. 7: Comparison between problematic and non-problematic MSSQL databases across appl_memory, num_partitions, num_connections and log_file_size.

databases: (1) are either given too few resources (i.e., less application memory with more concurrent applications), (2) run on older or specific instance, server architecture and OS versions, (3) the actual resource allocation does not take into account external factors (i.e., low number of primary logs associated with more intensive workloads), or (4) data is not organized in a logical order (i.e., using heap memory, instead of clustered indexes). As a future step, we envision adding workload knowledge into the model, and such indicate poor database configurations that are workload-specific.

VI. Related work

Database configuration tuning is one of the major challenges in the field of database systems, since its difficulty comes from the exponential number of possible combinations of parameter values. In [10], the authors discuss the effect of various parameters (e.g., buffer size, cache, RAID levels, indexes, vertical and horizontal partitioning) on database performance, as well design and tuning principles that DBAs should take into consideration. As tuning activities take at least a quarter of a DBA's time [11], solutions range from helping the DBA make a more informed decision to full-fledged automation. One similar approach to ours [12] is to generate a ranking of the parameters based on their impact on the overall database performance. Results show that the ranking is dependent on specific query workloads, which agrees with our view that using rules of thumb for setting generic parameter values does not yield optimal configurations. Another proposed solution is to view the tuning problem as a decision problem and use probabilistic graphical models [13]. Although promising in the experimental phase, in order to obtain near-optimal values for only 4 parameters for Berkeley DB, the method requires 22 days for 60% of the training data. With DBMS like IBM DB2 or MSSQL, which have far more than a few parameters, the approach explodes in runtime.

A more recent solution is iTuned [14]. It provides not only guidelines for setting parameter values, but it is a full-fledged tool that gives the user concrete suggestions for parameter values, together with a description of the impact on the workload performance. In [15], the authors consider the idea of self-tuning databases. Using a feedback control loop and carefully constructed mathematical models, the approach shows promising results when applied in a modular DBMS architecture. However, when considered on current enterprise databases, the method quickly escalates in complexity. Along the same lines, [16], [17] propose the idea of classifying, formalizing, obtaining, storing, maintaining, exchanging and individually adapting DBA expert tuning-knowledge as shared domain of understanding in the autonomic management process. Oh et al. [18] analyze how resource usages respond by changing resource sizes in DBMS and use this knowledge to construct a method that automatically selects those resources that affect the overall performance. The main disadvantage of these approaches is that they aim to self-tune the entire plethora of configuration parameters of a DBMS. In [19], the authors restrict their solution to memory self-tuning only, since it is a relatively self-contained component of any database. They propose a cost-benefit model, that uses control theory, to perform autonomous management across heterogeneous memory consumers in IBM DB2. Similarly, Tran et al. [20] deal with the self-tuning of database buffers. Our approach, while not aimed at self-tuning, focuses on the fast classification of databases, the study of configuration parameter trends and the suggestion of compatible combinations of parameter values, across heterogeneous database families.

VII. Conclusion

We propose a methodology for classifying databases into problematic and non-problematic, as well as to rank the configuration parameters based on their impact on DBMS performance. Moreover, we study trends of specific configuration parameters and compare their values -across problematic and non-problematic databases, in order to suggest better configurations. Such an approach is a great aid for DBAs to channel their tuning efforts. In this paper, we present several findings: (1) by adding detailed configuration parameters, the model accuracy increases significantly and we are able to better link problems reported in incident tickets to poor database configurations; (2) there are poor compatibilities between specific OS and database versions or server architectures, (3) there are significant shifts in value for specific parameters between problematic and non-problematic databases, which help DBAs to design better configurations.

Acknowledgment

The authors would like to express their gratitude to the PASIR team at IBM Research - Zurich, as well as the Database Management teams at IBM Brazil for providing us access to the data collections and constructive discussions that helped to improve the quality of the analysis.

REFERENCES

[1] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala, “Database Tuning Advisor for Microsoft SQL Server 2005,” *Proc. of 24th ACM SIGMOD International Conference on Management of Data (SIGMOD’05)*, 2005.

[2] G. Valentin, M. Zuliani, D. Zilio, G. Lohman, and A. Skelley, “DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes,” *Proc. of the 16th International Conference on Data Engineering (ICDE’00)*, 2000.

[3] S. Kwan, S. Lightstone, B. Schiefer, A. Storm, and L. Wu, “Automatic Configuration of IBM DB2 Universal Database,” *Proc. of IBM Perf. Technical Report*, 2002.

[4] IBM DB2, “http://www-01.ibm.com/software/data/db2/.”

[5] Microsoft SQL Server, “http://www.microsoft.com/sql-server/.”

[6] Oracle, “http://www.oracle.com/us/products/database/overview/.”

[7] J. Bogojeska, D. Lanyi, I. Giurgiu, G. Stark, and D. Wiesmann, “Classifying Server Behavior and Predicting Impact of Modernization Actions,” *Proc. of 9th Int. Conference on Network and Service Management (CNSM’13)*, 2013.

[8] L. Breiman, “Random Forests,” *Machine Learning*, 2001.

[9] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. Springer, 2009.

[10] D. Shasha, P. Bonnet, and N. H. Bercich, “Database Tuning Principles, Experiments, and Troubleshooting Techniques,” *Proc. of ACM SIGMOD Record, 33(2)*, 2004.

[11] A. Rosenberg, “Improving Query Performance in Data Warehouses,” 2005.

[12] B. K. Debnath, D. J. Lilja, and M. F. Mokbel, “SARD: A Statistical Approach for Ranking Database Tuning Parameters,” *Proc. of 3rd International Workshop on Self-Managing Database Systems (SMDB’08)*, 2008.

[13] D. G. Sullivan, M. I. Seltzer, and A. Pfeffer, “Using Probabilistic Reasoning to Automate Software Tuning,” *Proc. of 4th ACM SIGMETRICS/PERFORMANCE International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS’04)*, 2004.

[14] S. Duan, V. Thummala, and S. Babu, “Tuning Database Configuration Parameters with iTuned,” *Proc. of the VLDB Endowment, 2(1), 1246-1257*, 2009.

[15] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback, “Self-tuning Database Technology and Information Services: From Wishful Thinking to Viable Engineering,” *Proc. of 28th International Conference on Very Large Data Bases (VLDB’02)*, 2002.

[16] G. Rabinovitch and D. Wiese, “Non-linear Optimization of Performance Functions for Autonomic Database Performance Tuning,” *Proc. of 3rd International Conference on Autonomic and Autonomous Systems (ICAS’07)*, 2007.

[17] D. Wiese and G. Rabinovitch, “Knowledge Management in Autonomic Database Performance Tuning,” *Proc. of 5th International Conference on Autonomic and Autonomous Systems (ICAS’09)*, 2009.

[18] J. S. Oh and S. H. Lee, “Resource Selection for Autonomic Database Tuning,” *Proc. of 21st International Conference on Data Engineering (ICDE’05)*, 2005.

[19] A. J. Storm, C. Garcia-Arellano, S. Lightstone, Y. Diao, and M. Surendra, “Adaptive Self-tuning Memory in DB2,” *Proc. of 32nd International Conference on Very Large Data Bases (VLDB’06)*, 2006.

[20] D. N. Tran, P. C. Huynh, Y. C. Tay, and A. K. H. Tung, “A New Approach to Dynamic Self-tuning of Database Buffers,” *Proc. of ACM Transactions on Storage, 4(1)*, 2008.