AutoDBaaS: Autonomous Database as a Service for managing backing services*

†

Mayank Tiwary
University of British Columbia
Vancouver, Canada
mayank09@cs.ubc.ca

Pritish Mishra
University of Toronto
Toronto, Canada
pritish@cs.toronto.edu

Shashank Mohan Jain
SAP Labs Bangalore
Bangalore, India
shashank.jain01@sap.com

Kshira Sagar Sahoo
Department of CSE, SRM University AP Amravati
Mangalagiri, India
kshirasagar12@gmail.com

ABSTRACT

This work introduces and aim to overcome the potential challenges while deploying automated tuning of relational database as a service for a Platform as a Service (PaaS) provider. Some of the major challenges identified in this work include (i) automated detection of performance throttling (figure out when the performance of the system is affected due to incorrect configurations of knobs) of a database and identify potential points where a database requires a tuning, (ii) scalability and accuracy of tuning service and (iii) applying the recommendations obtained from tuning services wherein applying an obtained recommendation might require a database restart.

In this work, we present a generic tuning service architecture for PaaS providers. To deal with the above challenges, we introduce performance throttling engine which is responsible to detect potential points when a relational database actually needs a knob tuning, which helps in increasing the scalability and accuracy of the tuner deployments (responsible for tuning production landscapes). This work also proposes approaches that facilitate efficiently applying the recommendations without causing much disruption in Quality of Service (QoS) of the underlying database system. Lastly, the results are obtained by evaluation of the proposed methods and modules on multiple cloud native provisioners against various set of metrics.

1 INTRODUCTION

The PaaS customers do not have access to tune the configuration knobs of database/backing services as the service configurations are often abstracted. Tuning of the offered data services often requires DBAs to pitch in, observe/monitor and then, tune the service-instances. This often adds more complexity, as PaaS providers needs to have a DBA for each customer group, where each service offered has tens to hundreds of knobs to be tuned. In literature, there exists a set of various auto-tuners [1], [2] and [3] that aim to automate the tasks of a DBA. These tools are not holistic in nature and are limited to specific classes of parameters.

This work introduces challenges and requirements for introducing generic tuner as a service (AutoDBaaS), which can tune the configuration knobs of the relational data services as per requirement and thus, in-turn reduce the performance dependency on a DBA. In this work we evaluate already existing tuners and try to see how they can be used to tune live production systems. In literature there exists multiple style of tuners, broadly classified as search based [16] and learning based. This work specifically considers learning based tuners (as they can easily tune multiple types of databases and more suitable for PaaS service providers): bayesian optimization (BO) style tuners (like Ottertune [4]) and reinforcement learning (RL) tuners [18] and [17]. We discuss the pros and cons of both RL and BO style tuners for tuning live production systems in coming sections. Potentially, the challenges that drive the design and deployment of a tuning service as per PaaS architecture have been identified as follows:

- Scalability of Tuners
- High Quality Samples
- Metrics for Tuner Evaluation

Scalability of Tuners. As per the architecture of a BO style, it uses previously observed workloads to train a Gaussian Process (GP) regression (or a surrogate model), which recommends a new set of configs. The workload in Ottertune (any large scale machine learning tuner which tries to leverage previous experiences) is a collection of different knob values, obtained with respect to observed database metrics. The workload should contain enough data, where sufficient metric variations are observed across different variations in values of knobs. Or in another sense, a BO stlye tuner like Ottertune needs high volume of high quality samples. With the high volume samples, the Ottertune's workload size increases and causes a GPR training to take a time of around 100 to 120 seconds. Then if the underlying services, asks for recommendation with a high frequency of 5 mins (a typical monitoring time for a transactional data), one Ottertune deployment can be bound to a maximum of 3 to 4 service instances. This can also be inferred as a cost for a BO style tuners - 'recommendation-cost' to service-provider. In this aspect the RL style tuners do pretty well, as they do not need high volume of high quality samples. However, the pros and cons of BO style tuners and RL style tuners are discussed in comming sections.

High Quality Samples. Both RL and BO style tuners need to capture delta metrics (after execution of a workload) from the underlying database to be tuned. The quality of the captured metrics (or the quality of the samples) depends solely on the workload executed on the database. For example - when a client

---

executes TPCC queries to a database, continuously for 10 minutes with 3000 requests per second, will generate a high quality sample. However, in production systems, the throughput of the executing workload is often low (or for most of the time, the production database does not needs a tuning), which cannot produce enough variations in the delta metrics (or often a low quality sample). In production systems, the spikes in throughput graph is seen at specific time-intervals only. In many cases, it is observed that even if there is high throughput, only a certain set of metrics show good variations and rest do not [4]. The quality of samples highly impacts the performance of both RL and BO style tuners. And, in production systems capturing high quality samples is very difficult (or atleast there exists no such way to do so in literature). In a nutshell, the main problem is that a production/live database tuner faces corruption of learning model when it trains over such samples (or collected metrics) which sometimes does not requires any tuning (and this is very much seen on production workloads). However, an offline tuner (which is expected to tune a staging, development, testing landscape database) does not face this issue. This is because in these database all the queries are batched to form a workload and the workload gets executed for a time frame which generates a high quality sample.

Metrics for Tuner Evaluation. The tuner service the knobs, metrics and provides recommendations for performance improvements. As an end user, the expectations will be to get recommendations when the workload pattern changes in real time and the recommendation should actually improve performance. Currently there are ways in literature which can suggest changes in workload patterns [8], [19]. This works use templates (from queries) and cluster them. However, still there is no such information that with change in workload does the database actually needs a tuning. Secondly, just an increase in throughput cannot be a qualifier to suggest performance improvements. The reason for this is that in production systems, the workload pattern always changes i.e. say the throughput was measured when a query set was executed on a production database. Now after applying the recommendations, the next query set will get changed (or the workload gets changed) and the new throughput cannot be compared with the previous one. Hence, measuring performance based on recommendations on production system is also challenging.

A unifying theme to the above challenges is to identify the actual performance throttling on a database system. With respect to changing workload pattern of users SQL workload, performance throttling detection will help in predicting the incorrect knobs.

The paper makes the following novel contributions:

- **Identifying Performance throttling**: The performance throttling is responsible for identifying the database insufficiency to process SQL queries due to incorrectness of configured knobs. It classifies the knobs into different classes and then, for each class of knobs, it predicts throttling. This module increases the scalability of the tuner deployment by reducing the number of recommendation requests (when compared with the periodic nature of making tuning requests). The underlying services request for recommendation only when a performance throttling is detected. Thus, this module acts as a DBA and identifies when a database requires an actual tuning in real-time. This module identifies performance throttles from relational databases only.

- **Applying Recommendations in an effective way**: The tuning agent running on the database VM/Container as an plugin process, applies the recommendation on service-instance by re-loading the configs. The same plugin is also responsible for tuning of knobs that require a restart of the database.

- **Evaluation**: In production systems as the throughput varies, we need to identify new performance metrics to compare the effectiveness of obtained recommendations. To achieve this, we introduce the number of throttles triggered by the performance throttling module, as a metric.

2 SYSTEM DESIGN

This work presents a generic architecture, which can be easily integrated with any available cloud platform provisioners. As shown in Figure 1, the overall deployment of database tuning service, in an abstract form, is divided into two parts: (i) tuner instances - responsible for executing the ML pipeline to generate new config recommendations and (ii) config director instances - responsible for managing all available customer service-instances. The tuner instances can be spawned via either containers or VMs. There can be more than one tuner instances (also depends upon tuner scalability), where each tuner stores a workload W in a database, where a workload is combination of knob config parameters and metrics observed against those parameters (also called as training samples). Technically as described in [4], a tuner workload W is a set S of N matrices $S : \{X_0, X_1, X_2, ..., X_{N-1}\}$, where $X_{m,i,j}$ is the value of a metric m observed when executing a user SQL workload on database having configuration, j and the workload identifier, i. The tuner service uses the workload W for initial training of both BO and RL style tuners. These workloads are stored in database which is present on a different instance. This database acts as a common central data repository for all tuner instances. Tuning agent runs on the same database (and communicates to DB using Domain Sockets) which is responsible for identifying new workloads and uploads new workloads data periodically to the central data repository. The tuning services running on different IaaS'es, fetch the new workloads from the central data repository. This helps all tuning services to get the new unknown workloads, which might have been observed on a different IaaS, and create a better ML model.

The metric readings and recommendation request calls (we call it something like a tuning request) are event-based and triggered from the performance Throttling Detection Engine (TDE). The TDE gets periodically executed on the database master VM (like a plugin) and triggers recommendation requests to the config director. The TDE runs periodically on the master VM of the underlying database service and is responsible for figuring out performance throttling due to incorrect knob values with respect to current executing user workload. The config director receives the metric data (or queries in case of a RL based tuner) from service instances and triggers recommendation requests to tuner instances. The config director performs load balancing of recommendation request tasks across multiple tuner instances. The Service Orchestrator agent running on database services, is responsible for performing all life-cycle operations of service instances and maintains credentials. When the config director receives a new recommendation for a database service instance from a tuner, the config director passes the new configs synchronously to Data Federation agent (DFA) and Service-Orchestrator, while simultaneously storing it into the config data repository. The DFA
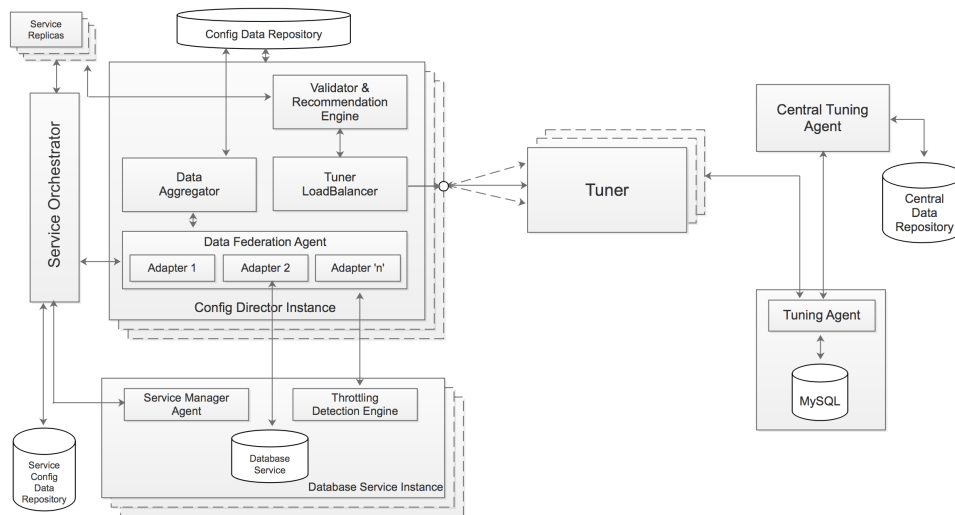
Figure 1: AutoDBaaS architecture.

fetches the credentials from Service Orchestrator layer and hits the APIs of TDE to apply configs to all nodes of the database service (such as all VMs/Containers of the service-instance). The DFA has multiple adapter implementations to get connected to various kinds of database services. As per the architecture, the tuner deployment is capable to tune multiple databases instances (one to many).

2.1 Tuner Instances

The tuner instances as shown in Fig. 1, can be any type BO or RL style tuners. Or can even be a hybrid combination. RL vs BO style tuners is offcourse a debatable topic as both have their pros and cons. BO style tuners when fully tuned with high volume of high quality samples can tune an underlying database in just two to three recommendations, where as RL style tuners need to experience large number of recommendations over the database (as per try-and-error strategy) to learn good configurations for the new workload pattern. At the same time, RL style tuners are highly sclable as they can quickly generate new configurations when properly trained (RL style tuners do not need high volume of high quality samples). Both BO and RL style tuners learning model relay on captured metrics when database actually needed tuning. And metrics/samples captured from database when database did not needed any throughput tuning (this is the often production system case), this corrupts the learning models of tuners. With such wrongly captured metrics BO style tuners face a cascading style corruption and RL style tuners face corrupting the current learning model. This problem of both RL and BO style tuner does not enable them to tune live production workloads, which are often characterized by low throughput or when the database does not needs any tuning. Well this is the major motivation for driving this work.

3 IDENTIFYING PERFORMANCE THROTTLES IN DATABASE

One of the crucial initial steps that the DBA performs before tuning is, monitoring the database to identify whether the database actually needs tuning or not. This module executes periodically as a part of TDE, gathers statistics based on the metrics/features collected using a rule-based approach and identifies potential points when a database needs a recommendation for tuning its configs. As per the proposed performance throttling approach, the config knobs of a relational database can be categorised, based on their properties, into three classes:

- Memory knobs
- Background writer knobs
- Async/Planner estimate knobs

The working of each sub-module is different and is explained as follows:

3.1 Memory Knobs

Memory knobs are the set of knobs which are dependent upon the resource (VM or container) hardware limits. The major portion of memory used by the database is utilised to keep the data in buffer. One of the approaches for identifying throttles could be to find out the actual working database size. To identify this, we use the algorithms proposed by authors in [5] where the authors use gauging techniques to identify the actual working page set. However, the major challenge with this knob is encountered while attempting to update this knob, since it requires a restart of the database. The TDE, collects this information and keeps on sending it to config director instances, where config-director collects the number of throttles and checks the size of the working page set and adjusts this knob value only during the scheduled maintenance downtime.

The other knobs belonging to this segment are related to working area of the database. The knobs related to the working area of memory depends upon the total number of active connections and if it is found to be in-sufficient, then database uses disk or system swap space to perform work operations like sorting, or maintenance operations like index-creation, storing temporary tables, table alter, etc. To get the memory usage details probes needs to be created in the codebase, which is arduous and dependent on freedom given from vendors. Alternatively, figuring out the disk usage while query execution, the query plans can be used as a potential source of information. We use query templating as described in [6], to reduce the total queries (to be examined in production systems), where the queries are converted to a template having a template-id. The queries collected from streaming logs are pre-processed and then converted

to generic templates (having no actual parameters/arguments). The final template selection takes place from the pool of queries by reservoir sampling (for capturing samples from streaming logs) [7]. The selected query templates undergoes execution plan evaluation by substituting the actual (most frequent) parameters to the template. From the plans/streaming logs, it can be easily inferred how much memory/disk the query is going to take. If any of the selected templates (from reservoir sampling) uses disk while execution, signifies that the memory is in-sufficient for execution of queries and now the TDE triggers a memory based throttle signal and asks for knob recommendation from a tuning service (raises a tuning request to the tuner).

However, there can be potential cases when the memory allocated for the buffer is maximum (which means the memory left for other processes becomes less), it is observed that TDE un-necessarily triggers throttle signals. This is a case, where the underlying instance configuration limit is in-sufficient (or the usage has reached the caps limit). When the size of the database is sufficiently higher than the actual memory allocated to database process, it is observed that the TDE frequently triggers throttle signals and is unable to understand that the throttles are being caused because of limited hardware resources. To deal with this cases, we need filtration approach, which identifies such situation and stops the un-necessary throttles (one potential case is the underlying VM hardware resource is in-sufficient and customer needs to upgrade to another plan or ask for more resources for the VM). We face the following challenges when designing such filters:

- There are a specific set of queries which trigger consecutive throttles from one memory knob (like use of aggregate queries triggers a throttle from working memory in PostgreSQL). Situations like this can cause increasing working memory continuously with each recommendation obtained and hence decreasing other knobs (to make room for increase of working memory). However, even after increasing the knob values to the maximum, throttles can get triggered. This situation can easily be captured by rule-based engine and throttles can be filtered.

- For a certain query, consecutive throttles are observed intermittently against different knobs. For example the first two throttles came from working memory and next two throttles came from *maintenance_work_mem*. This becomes very difficult to manage and identify with a rule based approach especially when number of knobs are high.

- There are a specific set of queries which triggers consecutive throttles from more than one set of knobs at a time (like use of aggregate queries, index creation queries, temp table creation queries, etc causes trigger of throttle from multiple knobs). Situations like this are difficult to be captured by rule based engine (becomes more complex when knob numbers increases or is already high) and needs a different approach.

We observed and collected such queries and table shown in Fig. 2 shows the same. In PostgreSQL, working memory is used by the execution engine to perform internal-sorting, joins, hashtables, etc. We evaluated amount of working memory used by TPCC and CH-Bench, YCSB and Wikipedia bench in absence of indexes. We observed that Wikipedia and YCSB queries do not use working memory (due to absence of complex queries like aggregate, joins, and order-by). The table illustrates the actual

working memory allocated and the amount of disk and memory used by queries.

Benchmark – Query Type	Work Mem (in MB)	Disk space utilized (in MB)	Memory utilized (in MB)	Execution Latency (in ms)
tpcc – Scan Query	0.5	3	0.5	8310
tpcc – Scan Query	1	0	0.64	3953
ch-bench – Scan Query	125	260	125	15413
ch-bench – Scan Query	150	0	137	6801
tpcc – Complex Aggregation Query	350	475	350	53013
tpcc – Complex Aggregation Query	375	0	367	19835

Figure 2: Queries and Memory statistics observed on PostgreSQL running on AWS VM, type-t3.x_large
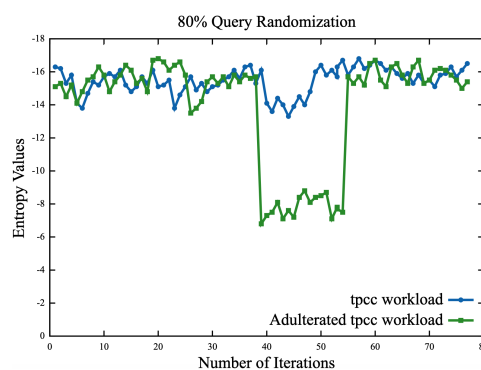

Figure 3: Entropy variation with 80% adulteration probability on Production SQL Workload
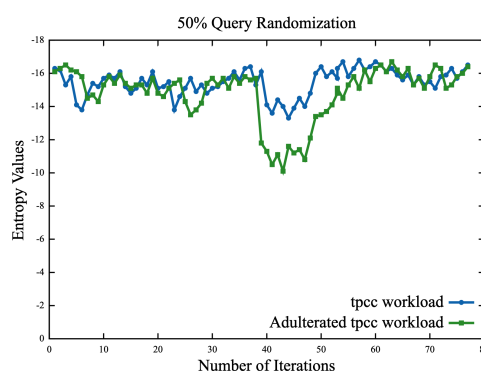

Figure 4: Entropy variation with 50% adulteration probability on Production SQL Workload

In this case, a probabilistic approach is needed to predict the pattern of SQL queries which can cause a potential throttle in performance. The queries which cause more use of working memory are mostly Join, aggregate queries, sorting queries (ORDER BY). On production systems, the frequency of rest queries like index creation or alter table is comparatively lesser. The worst-case scenarios could be all queries are fired with similar proportion. To deal with such cases, or to identify such randomness/query proportion, (to measure the probability distribution) entropy is

used. Entropy of a discreet variable X with possible outcomes $x_1, x_2, x_3, ..., x_n$ can be defined as:

$$H_n(X) = -\sum_{i=1}^{n} p(x_i)log(p(x_i)) \quad (1)$$

where $p(x_i)$ is the probability of the ith outcome of X. With a more abstract approach, a generalized entropy can be defined as:

$$\eta(X) = -\sum_{i=1}^{n} \frac{p(x_i)log(p(x_i))}{log(n)} \quad (2)$$

Value of $\eta(X)$ can range from 0 to 1 i.e. $\eta(X) \in [0, 1]$. This helps in determining the threshold value of entropy, as for any number of classes the normalized entropy ranges between 0 and 1.

The queries are grouped into specific categories (grouping of obtained query templates), such as Join queries, Select queries, Alter-table queries, Update queries, etc and a hash table is built for each category. The classification of queries is done based on the trigger of throttle from knobs, for example - complex aggregation queries are grouped to one class which triggers throttles to working memory knob. Similarly, we create individual class for each given knob. From the generated logs, we create a hash table containing the class of queries and its frequency. Once the entropy value is evaluated, it can be inferred from multiple observations, that the entropy value is less when high randomness is present or all queries are fired with similar proportion (the query frequency from classes are evenly distributed). This, thus, indicates the SQL queries will, probably, again trigger a throttle (when underlying instance configuration is in-sufficient). However, if the entropy value is high, the degree of randomness is quite less or probability is quite evenly distributed. Thus, provided, if the query class, which is constrained by throttles, has less frequency, it can be concluded that in future, the throttles will not be triggered (as here the limits have not reached the caps and the underlying database depends on the tuner recommendation for knob optimization).

As part of the proposed flow, if more than 8 throttles are triggered consecutively, the entropy value is evaluated, and if the entropy value is higher along-with the memory-knobs reaching maximum cap value, the TDE triggers a plan update (increasing the hardware limits of instance) request to customer and recommendation requests are not sent to config director. Else, it is estimated that the throttles will soon reduce and the same job waits for next 8 throttles before calculating the next entropy value. The graphs shown in Fig. 3 and 4 shows the calculated entropy values while executing TPCC and an adulterated TPCC workload. The TPCC workload was adulterated with index creation, index drop, complex-joins, temp-table creation, order-by and aggregate queries.

In order to showcase the entropy variation, we loaded TPCC with a scale-factor of 18 (which loads around 21GB of data) to Postgresql. However the queries fired mostly hit the working memory and wal-memory knobs ($sort\_buffer\_size$ in MySQL). The amount of working memory used by TPCC as shown in Fig. 2 is around 0.5 MB, which is quite less to generate a throttle from memory based knobs. Hence now we add complex aggregation queries to TPCC (like queries having heavy sorts), which requires nearby 350 MB. Still we are able to trigger throttle for only working memory using TPCC. Now in order to design such a workload which triggers throttles from all defined classes/knobs, we started adding more queries and

procedures, the following queries (analysing from production level performance bottlenecks faced earlier) were added to TPCC bucket:

- complex sorts/aggregation queries - To trigger throttle from Postgresql - $work\_mem$, MySQL - $sort\_buffer\_size$ and $join\_buffer\_size$
- create/delete indexes - To trigger throttle from Postgresql - $maintenance\_work\_mem$, MySQL - key_buffer_size and $sort\_buffer\_size$
- delete queries: To trigger throttle from Postgresql - $maintenance\_work\_mem$
- creating temporary tables and firing complex aggregation queries on it to trigger throttle from Postgresql - $temp\_buffers$ and MySQL - $temp\_table\_size$

Now the new queries are always added to the actual TPCC bucket based on probability as given in Fig. 3 and Fig. 4 (80% and 50%). With the adulterated TPCC workload, we were able to simulate throttles from all set of classes/knobs. The probably distribution with TPCC varies hugely with the probability distributions of adulterated TPCC due to absence of the new queries and results in entropy difference.

3.2 Background writer knobs

The background writer knobs control the writing of dirty pages from buffer, back to the disk. This write process is triggered by background writer processes or periodic checkpointing processes. However, if the checkpointing process is triggered too often and the amount of data written is high, then it leads to higher values in consumption of I/O throughput and disk latency resulting a decrease in throughput of the database. The other processes involved in writing dirty pages back to disk (background writer) helps in mitigating the same problem, with the aim to reduce the amount of data written by a checkpointing process. Usually the background process writes a fixed number of pages back to disk and the left pages are taken care by checkpointing process. In case of write heavy workloads, the background process writes fixed amount of data causing uncertain amount of data written by a checkpointing process. Given a discrete configuration for the set of knobs, for identifying throttles the following set of challenges needs to be overcome:

- To find out optimal value of checkpointing triggered per unit time. This parameter helps in understanding the overall period till which there can be a surge in disk latency, IO, etc.
- To find out optimal value of data written to disk with trigger of a checkpoint. This parameter helps in understanding the max surge the disk IO and latency parameters can go for write operations.
- There are various processes which write back to disk, for example - WAL writer, statistics writer, log writer, archiver, garbage collector, vacuum. This makes it difficult to figure out holistically the exact amount of data written by checkpointing process.

In order to figure out the exact amount of data written by a specific process requires use of user-level statically defined tracing probes (USDT probes). Where, any low-overhead tracing tool like ebpf or dtrace (Linux Foundation - IO Visor project) can use the probes to get information. The other option is to use kernel-probes (uprobes) for tracing, but this is also independent of the database process levels. Hence the safest way to get this data is to move writing of majority of processes to another

disk. In our experimentation, we changed the disk for storing of WAL, statistics, logs, etc. Now only background writer processes or checkpointing processes and vacuum/garbage-collector processes are responsible for writing on the current disk (where the production database files are located). This strategy also guarantees SLA for minimum IOPS for a disk which stores the actual database and at the same time increases cost of extra hardware and operations. Still the checkpointing process can be interpreted with the vacuum/garbage collector processes which is responsible for updating indexes for dead tuples and defragmenting pages on disk. The frequency of this process can easily be controlled and the left slots can be utilised for monitoring of checkpointing processes. During experimentations, we increased the frequency of vacuum/garbage collector to substantially a higher value and neglect the monitoring of checkpointing during the interval when vacuum/garbage collectors are triggered.

To predict/evaluate the values of optimal checkpointing and optimal amount of data written per checkpoint, the proposed approach uses the historic data of the workloads stored in the tuner's database or in short leverages the tuners experiences of tuning write oriented workloads. The workloads which are generated for tuner, are often pre-generated offline or it considers newer workloads as well (workloads from live database systems). The tuner service for recommending new knob values selects a target workload, and then uses the target workloads data to train the GPR. However, in all cases we monitor the disk latency from external monitoring agents such as Dynatrace. The throttling point for these knobs depends upon the disk latency as the performance degrades when the disk latency increases. In order to figure the optimal checkpoint per unit time with amount of data written to disk, the time difference between peaks in disk-latency is observed and averaged out for consecutive peaks. We define checkpointing per unit time based on the same observations. The checkpointing per unit time is calculated only for the highest observed throughput point in mapped workload.

Each database service in order to get the optimal parameters uses the best information seen/tried by tuner in past. Now, when a throttle is triggered, the tuner maps the current workload $'A'$ (workload representing a target underlying database service) to a target workload $'B'$ which had shown similar features in the past, with respect to the current workload. Now, for B, the timestamp value for the most optimal points observed (with respect to maximum throughput) are captured and passed on to the Dynatrace agent and the disk latency readings are collected. The points are the best recommended knob sets obtained using a trained GPR. From this data point, for the entire duration of workload execution on database, the checkpointing per unit time and respective disk latency is observed. Now on live/production systems, the checkpointing per unit time for A is calculated based on the baseline of disk-latency defined (obtained from tuner) earlier. If in $'A'$ the ratio of checkpointing per unit time and disk latency is more than the ratio of checkpointing per unit time and disk latency for B, then the throttle-detection scripts trigger a throttle signal. However, there could still be scenarios, when the workload A has very less data points (config values vs metrics) and for such a scenario, the mappings are initially incorrect for target workloads. Then for that scenario, the number of throttles could be either more or less, however, with each throttle signal that is triggered, the workload size increases and probability of getting mapped to an optimal workload increases. Thus, the proposed approach eventually improves in efficiency with passing time.
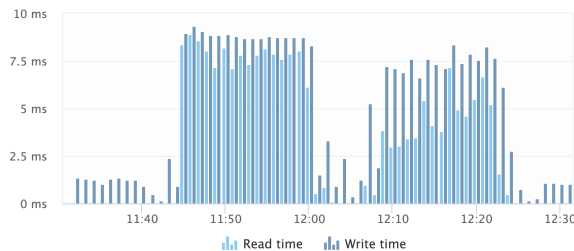

Figure 5: Disk Latency graph for TPCC execution.

The graph, as shown in Figure 5, represents the disk latency incurred when TPCC is executed on PostgreSQL with default knob config values and compared with when it is executed with optimal knob config values. The readings observed from 11:45 to 12:05, show the disk latency values for TPCC execution with default knob values and readings observed from 12:10 to 12:25, show the disk latency values for TPCC execution with optimal config values on PostgreSQL. Here, the TPCC execution on tuned PostgreSQL gives an average disk-write latency of around 6.5 ms and based on this the checkpointing per unit time is obtained. So, this becomes the base line for any workload on live systems which is mapped to the TPCC workload. Here, the major constraint is also that the underlying hardware (storage type as SSD or HDD) should be the same for all systems (databases used for training tuner and live systems).

3.3 Async/Planner estimate knobs

The async knobs are based on the ability of the database to parallelise the query execution, whereas the planner estimate knobs helps the query execution planner to estimate the best route. Most of the database recommends to statically set the planner estimate knobs (random page cost, effective cache size. etc) based on the underlying hardware capabilities. Still it is often seen that increasing/decreasing the values of such knobs (from the recommended values) improves the overall query execution. The async knobs are often defined by the number of parallel worker processes supported per relation by the database. During query execution, the parallel workers are taken from a pool of all defined workers. Often, it happens that the requested workers are not available or it could also happen that setting a higher value for these knobs affects the planner estimates. Thus, it always depends upon the nature of query and to what degree it can support parallel executions.

As this categories of knobs directly or indirectly impacts the planner estimates, it is often required to check the planners cost/benefit optimizations. The straight forward way to trigger a throttle would be to manually increase/decrease the knob values and check the overall cost/benefit optimizations. However, to automate this, the TDE needs to carefully take decision on whether to increase or decrease the value and by how much the value should be increased or decreased. Assuming at a given instance of time, for a given production workload, there exists an optimal values of this knobs. And the optimality does not depends on the underlying hardware (as per recommendations), making this a stochastic environment use case. Reinforcement learning is often seen as the best way for analysing the cost/benefit optimizations of query execution planner [8] [9] [10]. Hence, we model this problem as sequential decision problem and address it by using reinforcement learning.

Hence, we use a very basic Markov Decision Process (MDP) as a very basic RL model to solve the above sequential decision problem. In order tp minimise the uncertainty, the MDP starts with random set of actions and with course of time the action probabilities are adjusted, based on the response from the environment. The RL algorithm tries to optimize an agents returns when the episodes are restricted/limited. The RL engine captures all the queries in a time frame (typically a day or two based on the length of the workload). One episode comprises of atleast 350 to 400 steps (set of actions), where the knob values are changed as per policies (policies are random model initialization) and planner cost/benefit estimates are captured for all queries. The cost benefit estimates are then converted to rewards or penalties.

The TDE uses a MDP to trigger a throttle from this category of knobs. A MDP is represented by $\{Q, A, B, N, H\}$. For all given knob in this category, a MDP is given as follows:
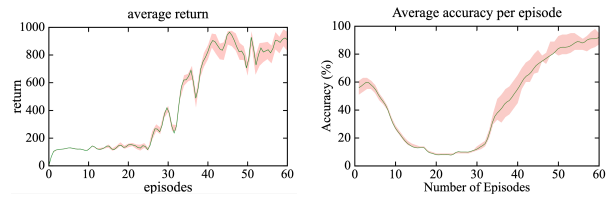
- Q is the finite set of internal states given by $Q = \{q_1, q_2, q_3, ..., q_n\}$, where q_n represents a specific knob value tried before or in current usage.
- $A = \{\alpha_1, \alpha2, \alpha_3, ..., \alpha_n\}$ is the set of actions performed by the automata (increase/decrease the knob value) where each action has its own probability distribution.
- $B = \{\beta_1, \beta_2, \beta_3, ..., \beta_n\}$ is the response from the environment (cost/benefit calculated from query planner)
- N is a mapping function responsible to map current state and input to the next state and
- H is a mapping function responsible to current state and response to figure out the action to be performed.

The TDE triggers the MDP at interval of 2 to 4 minutes, where the MDP performs cost/benefit analysis by fetching all the queries from log, performing reservoir sampling as described above (in throttling detection for memory knobs). For a given knob value (represented by q_n), based on the action probability, the MDP increases/decreases the knob value by unit step (defined statically). Later the TDE calculates the loss/profit in execution time against the sampled queries with respect to the new knob value and old knob value. If there is a loss, which signifies the action is misleading and the MDP penalises the respective action (which adjusts the probability of the given action α_n) and vice-versa. However if a profit is seen with the change of the knob, the TDE triggers a throttle to get a recommendation from the tuner. The graphs in Fig. 6 presents the learning progress for a production workload as shown in Fig. 8. In the initial episodes, it is observed that the learning is less as the agent is suppose to do more and more exploration of knob configs. However, as the iterations continues, we observe more and more learning (as the episodic rewards increases). This draws a balance between exploration and exploitation.

One can argue that if with the course of time the MDP learns about the optimal/sub-optimal values of the knobs, is it really necessary to go and again ask the tuner to get recommendation. Yes, the tuner needs to be asked as the optimality changes with respect to change in workload pattern and secondly the tuners learning models predict best values for the given knobs by utilising the past seen experiences from set of other production systems.

4 APPLYING RECOMMENDATIONS

The potential challenges in this job could be designing the overall orchestration mechanism for applying these configs, considering the prevalent architecture of the database system like multi-node,

(a) Learning progress of proposed policy (b) Average accuracy of learning process

Figure 6: Measuring Reinforcement Learning accuracy on production workload

high-availability constraints, etc. The configs need to be persisted too such that a database reset or re-deployment doesn't overwrite the settings. Additionally, concerns like how to apply the recommendations without causing a downtime of the running database system, must also be addressed.

Generic Approach. An orchestration approach had to be formulated in-order to apply the recommendations, taking into consideration the above-mentioned challenges. As per the architecture explained in Figure 1, the service-orchestrator is responsible for spawning of database system instances for a customer. The config of the spawned database system is generated and applied initially to the database, by the service-orchestrator. If for any reason (like updating the system, applying security patch, etc.), the database system needs a re-deployment, then the service-orchestrator must re-deploy the system with the updated config of the database.

As per the architecture, the Data Federation Agent (DFA) hits API endpoints of TDE to apply the config recommendations. In case of multiple nodes maintaining high availability, the recommendations are first applied to the Slave node(s). If the process crashes in the Slave node, the config recommendations are rejected. Thus, it is ensured that the Master node is up and the process is still able to serve requests. After the config recommendations are applied to the Master node, the recommendations are stored in the persistence storage used by the service-orchestrator. Thus, whenever the service-orchestrator re-deploys the database system in the future, it retrieves the updated config from the persistence storage. An additional concern here could be the failure in one of the intermediate steps. Since, all of the operations are not atomic, but eventually are expected to yield consistent data (i.e., configs must be same for all master/slave nodes and persistence storage used by service-orchestrator), a reconciler process is defined. The reconciler keeps a watch on config of the database system running on the Master node. If the difference in config is observed for a threshold time-period (watcher timeout), the reconciliation occurs and the config stored in the persistence storage is applied to all nodes. Thus, this eventually leads to rejection of the config recommendation due to error in the intermediate process.

For changing the knobs values, one of the efficient methods is to use Socket Activation (using sockets via *systemd*). This also makes possible to restart the DB since the socket is up and keeps on accepting the incoming requests. However this method only caches the requests but causes a lot of jitter and performance degradation. Another method is to use linux reload signals, upon evaluating this method, we observe very minimal jitter in the performance of the database. A comparative analysis has been

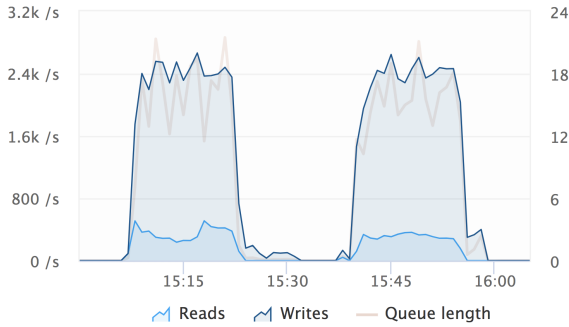presented in graphs in Figure 7 where the performance of the database is observed under identical load conditions.

Figure 7: IOPS graph for TPCC execution.

The graphs shown in Figure 7, the TPCC workload is executed with tuned MySQL. Readings during 15:08 to 15:23, shows the TPCC execution without triggering any config reload signals, where as readings between 15:38 to 15:54, shows the TPCC execution which was accompanied by a config signal reload with a frequency of 20 seconds (even with this high frequency of reloads, the performance is not compromised).

Applying Non-tunable Knobs. 'Non-tunable knobs' has been used as a term to categorise such knobs that cannot be applied/tuned without causing a restart of the database process. Since, the restart of database can only be performed during scheduled downtime window (a pre-announced time-period where a re-deployment of database occurs), an approach had to defined for the tuning of such knobs. The design of the approach can be considered for memory-related knobs (as non-tunable knobs are majorly memory knobs). For a non-tunable memory related knob, like buffer-pool's size, the optimum value of this parameter can be obtained from the working set [5]. Once this optimum value is determined, the database system is initially set up with the same value. However, there could be other memory-related knobs that are dependent on such a non-tunable knob. The value of all such knobs must be within the total memory allocated to the database process. Let us consider the following equation. $A + B + C + D < X$ Here, A could be assumed to be a non-tunable knob like $buffer\_cache$ size. B, C, D could be other tunable memory knobs like $work\_mem$, $maintenance\_work\_mem$ and $temp\_buffers$, where X is the total memory allocated to DB process.

There is always an upper limit on buffer-pool knob out of the total memory pool. This knob is changed only during the scheduled downtimes. During the downtime, if the total working page set size is greater than the maximum limit, then we find out the 99th percentile of this knob obtained during all last recommendations before the last scheduled downtime. If the new averaged value is lesser than the current value accompanied by at-least one entropy hit, then this knob value is reduced. The entropy hit indicates that the other tunable knobs have already raised many throttles and now it is mandatory to create more room for tunable knobs by reducing the buffer knob value.

Now when the memory for buffer value is reduced with respect to the current knob value, it increases more room for other memory related tunable knobs. So if the cost on throughput for tunable knobs is more, tuning services rotates around nearly same values for buffer knob, else in the next iteration it increases

the value of buffer knob (average value of buffer knob obtained till last last scheduled downtime encountered.)

5 PERFORMANCE EVALUATION

The experiments were conducted on AWS instances, where cloud resources is provisioned by cloud-foundry managed by Bosh. The tuner deployment consists of 12 tuner instances with Ottertune and CDBTune (we do not go for QTune due to unavailability of its codebased in opensource) - m4.xlarge with 4vCPU and 16GB memory, 5 config-director instances - m4.xlarge. We connected a total of 80 live-database deployments (spawned through t2.small, t2.medium, m4.large, t2.large and m4.xlarge VM types) to the tuning. For evaluating the experiments, we used PostgreSQL (v9.6) and MySQL (v5.6). All the tuner instances collected data from one common data-repository (m4.xlarge VM plan) which is shared by all tuner instances. The bare-service-replicas were created: one for each plan and were used to test the recommendations obtained. A real-time customer workload (activity for 33 days) is captured for the purpose of some of the below experiments. The SQL workload has 132 tables, 42.13M queries per day (average), 71K Select queries, 41M Insert queries, 34K Update queries and 0.8K Delete queries with a DB size of 59GB. The query arrival rate is shown in Fig. 8
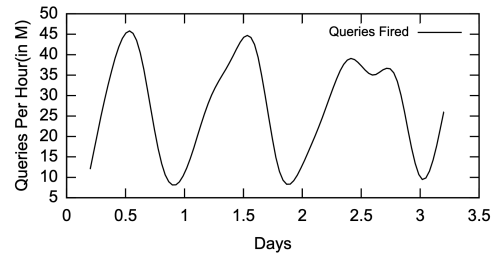

Figure 8: Production workload query arrival rate

Before evaluating the AutoDBaaS on live systems, we perform training of the tuners as per their standard ways [4] [18]. The first experiment we design is to measure the tuning requests per second on production landscape where both ottertune and CDBTune is being used for tuning. Figure 9 showcases one such outcome to illustrate the impact on scalability challenges of a BO style tuner. When comparing scalability of BO vs RL style tuners, a RL style tuner generates new configs very fast (but sometimes takes a long time to come around a good configuration). As per the BO approach, generating a new configuration takes around 200 seconds (which is assumed to be a good configuration). Both the RL or BO style tuners follow periodic approach (with a periodic length of nearly 5 to 10 minutes). In this case we bring in TDE which breaks down the periodic tuning approach. We measure the requests per second for live databases on production landscape where we compare requests per seconds generated when TDE checks in, periodic approach with a period of 5 min and periodic approach with a period of 10 mins. In both the cases it seems like the TDE approach gives a reduction and comes to peak when the workload pattern changes a lot like say morning 8AM to 11AM (when most of the microservice usages surge). The tuning requests per seconds when TDE checks in also directly gets impacted by the efficiency of tuner being used. If the tuner generates good configuration, in the next upcoming iterations, there are pretty less chances of a throttle getting detected.

As the proposed work largely reduces the tuning requests per minute, this evidence seems to directly impact the scalability of the tuning services and specifically the BO style tuners.
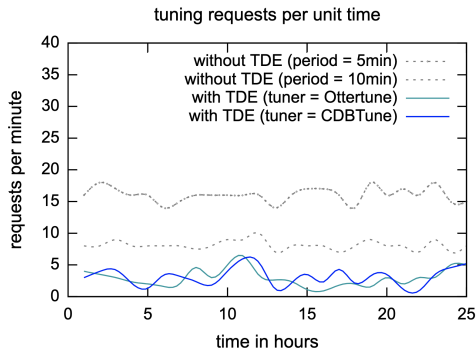

Figure 9: Requests per minute graph for 80 live connected databases

Next we measure the performance throttles due to incorrect knobs for some standard and production workloads. The parameters for the standard workloads was for (1) tpcc, 3300 requests per second with 26 GB of database size, (2) wikipedia, 1000 requests per second with 12 GB of database size (3) twitter, 10000 requests per second with 22 GB of database size and (4) ycsb, 5000 requests per second with a database size of 20 GB. We used oltp-bench to do the benchmarking on postgresql (Fig. 10) and mysql (Fig. 11) on m4-large instances. In order to purely measure the performance throttles, we do not go for a tuning session. These throttles presents averaged score for nearly 20 to 25 iterations. However, for production systems (as described above), we do not run iterations, rather they are actually captured from live systems directly for the workload described above and measured at different timestamps. We observe that for both postgresql and mysql, the write heavy workloads raise more throttles for background writer knobs, read-heavy/mix workloads raise more throttles for memory and async/planner knobs and for production workload it seem like a mix of ratios.

The next crucial metric for evaluation is Performance. There could be two metrics for measuring the performance of the proposed approach: (i) the throughput of the database system and (ii) the number of throttles encountered by the database system.

In Fig. 12, we measure the average throughput on live database using (1) Ottertune and (2) Ottertune with TDE (i.e. Ottertune only captures high quality samples from TDE). Ottertune uses samples from both production-workloads and offline-workloads (like executing tpcc offline) to train GPR. As per the tuning pipeline of Ottertune, it bootstraps with offline-workloads and starts tuning live systems. It separately captures each experiences from each workload (i.e. eiter live or offline). There is no chances of training model corruption with offline workloads as samples from offline workloads are captured are always of high quality (i.e. there is no such point when a offline workload does not requires a tuning). So when new batches of production system are hooked with the same ottertune instances, ottertune's throughput is roughly the same as compared with Ottertune + TDE as initially Ottertune uses offline samples (i.e. samples taken from offline workloads) to train GPR. However, the samples captured from the first batch of productions systems causes corruption to GPR (with high probability). Hence, when such samples are

Experimental Setup			
Variable used in Fig. 14	workload	Metrics window length	knobs class
#1	YCSB to TPCC	5 min	background writter, async/planner
#2	TPCC to YCSB	5 min	memory, async/planner
#3	YCSB to Wiki	7 min	async/planner
#4	Wiki to YCSB	5 min	NA
#5	TPCC to twitter	6 min	memory, async/planner
#6	Twitter to TPCC	5 min	background writter

Table 1: Experimental parameters and values

utilized to tune other set of production systems, the accuracy of GPR recommendations is extremely low. Hence, in Fig. 12 we hook in the 40th database instance and measure the throughput. As shown in the graphs the proposed approach seems to perform well and the main reason for that is there is no possible learning corruption in learning. For the workload executing in this database, we observed that Ottertune mapped the workload (with high mapping scores) to nearly 14 different workloads (to leverage tuning experiences) where only 4 of them were offline workloads. Similarly, we measure the same set of throughput when CDBTune is used as tuner. Here as CDBTune does not so much utilizes past learning experiences or atleast the way Ottertune does it. CDBTune minimally utilizes offline training but for sure does not uses learning from other production tuning experiences. Therefore in the case of CDBTune, this problem happens directly from the first hooked/subscribed database. The graph shown in Fig. 13 presents the throughput measured on the first database connected to CDBTune.

We also measure the effectiveness of performance throttling with changing of workload pattern by execution of standard workloads. The graph shown in Fig. 14 presents the same. This experiment is designed to measure how throttling detection helps to quickly capture workload change. In this experiment, we loaded 22GB of TPCC data, 24GB of TPCH data, 18.34 GB of YCSB data, 16 GB of twitter data and 20.2GB of wikipedia data on a m4-xlarge instance of postgresql. And we measure the throttles detected upon change of queries (i.e. from one workload to another). We present the details of experiment done here in the below table 1:

In this experiment we also observe and present the class of throttles. The tuner has a direct impact on the total number of throttles. This is because a single throttle triggers a tuning request and tuner recommends back a good configuration. Hence, in case of a very idealistic tuner the underlying database should not trigger more than one throttle as the idealistic tuner is expected to get the best config which would cause no throttles in the next iteration.

Lastly we tried measuring the accuracy of the throttling detection engine for all the classes of knobs what we presented. To evaluate the accuracy of throttles raised, either we could have used the human knowledge, where an administrator would have verified each throttles manually. But, this approach could have been time-consuming and could might result with a biased decision of the administrator. So another way to evaluate the throttles was to use the tuning of an already trained tuner. We trained Ottertune with offline workloads like TPCC, YCSB, Wikipedia and Twitter and then observed the throttles classes and configurations generated by Ottertune. If Ottertune recommends a majority of knob (say out of top 5 ranked knobs) whose class is
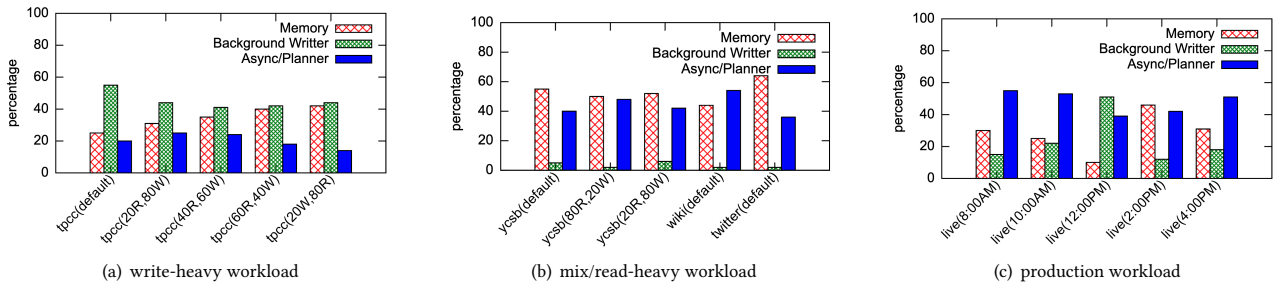
(a) write-heavy workload   (b) mix/read-heavy workload   (c) production workload

Figure 10: Performance Throttles detected on postgresql for varied set of workloads

(a) write-heavy workload   (b) mix/read-heavy workload   (c) production workload

Figure 11: Performance Throttles detected on mysql for varied set of workloads

(a) Average throughput per hour on Post-
gresql

(b) Average throughput per hour on
MySQL

**Figure 12: Throughput graph for live production database
with Ottertune**

(a) Average throughput per hour on Post-
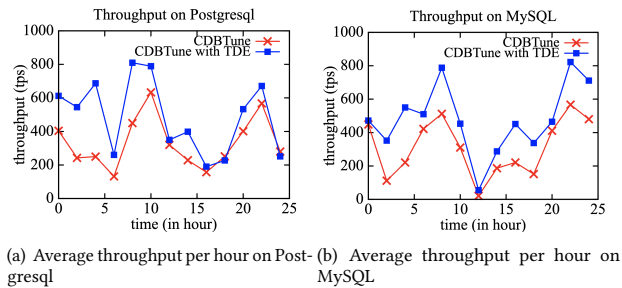gresql

(b) Average throughput per hour on
MySQL

**Figure 13: Throughput graph for live production database
with CDBTune**

same as the class of throttle, then the throttle was accurate else
we consider the throttle to be not accurate. This is specifically
tested on the same workload with which Ottertune was trained
i.e. TPCC, YCSB, Wikipedia and Twitter (as for the same trained

Figure 14: Throttles captured when tuner is Ottertune

data accuracy would be very high). Ottertune recommendations
sometimes perform exploration for the Gaussian Models better
training. We minimize this exploration by setting appropriate
hyper parameters manually. With this settings, Ottertune's rec-
ommendation should least explore and only aim to maximize
the throughput. We loaded similar amount of data on a Post-
gresql m4-xlarge instance as done in the previous experiment.
The graph shown in Fig. 15 presents the same. We observed high
accuracy for memory and background writter knobs and a lower
accuracy for planner/async knobs. However, we are confident
more for planner/async knobs as the throttle points clearly shows
improvement as per cost-benefit analysis of planner estimates.
However, we observed ottertune fails to understand such throt-
tles mainly because of absence of planner estimates in the metric
set that it captures for postgresql.

6   RELATED WORK

Facebook introduced Pressure Stall Information (PSI) [11] for
evaluation and control of computational resources across large
data centers. It is one of the first canonical ways of measuring re-
source pressure increase as it develops based on pressure metrics
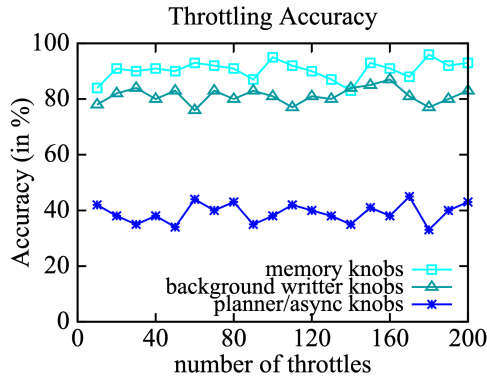such as memory, CPU and I/O using *cgroup*2 and *oomd*. There

Figure 15: Accuracy of performance throttles on Postgresql

are many methods which exists in literature which try to capture similar pressure on computational resources. However in case of databases these approach seems to be in-efficient in figuring out pressure on individual database knobs. This is also because of the reasons that database knobs are mostly indirectly-non-linearly dependent on the computational resources.

Oracle came up with a database internal monitoring mechanisms [14] and [15] to identify the bottlenecks in performance due to misconfigurations in internal components or knobs. Here, authors propose 'database time' of query as a parameter to figure out performance bottlenecks. Later this information is given to DBA's for tuning of knobs. The system uses heuristics from performance measurements for tuning of memory knobs and however does not tunes all set of knobs.

In literature, there are many knob tuning approaches [12] and [13], which are either specific to specific databases or tune only a subset of knobs. Other PaaS providers like AWS RDS service, gives freedom to consumers to tune it based on the workload. The architecture of Ottertune seems to meet the requirements of PaaS tuning offerings to customers, based on its capabilities to tune multiple databases by leveraging the workloads seen by tuner in the past. Oracle came up with autonomous database, a similarly solution came from Microsoft, however the approaches does not tune more than one database (unable to leverage experience gained by another system) and is only coupled to tune one database at a time (and thus increasing cost of tuning).

Also there exists multiple works that have focused on tuning database knobs. However, there exists two main classes: (1) Search based methods like BestConfig [16] and (2) Learning based methods which includes BO style learning or RL style. We do not consider Search based methods for tuning production systems as it expects the user to execute the workload on staging landscapes and tries tuning it. However, it cannot tune live systems as it takes huge amount of time to get to a good configuration. This is because for every tuning from scratch they again start the searching process from scratch. This work mainly focuses to solve specific problems of learning based methods like Ottertune and CDBTune.

7 CONCLUSION

We presented a generic tuning architecture for tuning services to be provisioned by any PaaS model. In this work, we bring in the challenges and drive them to make the AutoDBaaS more robust for production environments. To take up all the challenges,

this work presented (1) methods to monitor database and detect performance throttling, which helps the database to trigger recommendation requests only when potentially required and calculating the monitoring/observation time, (2) methods for applying and validating the obtained recommendations on production systems. Lastly we evaluate the proposed architecture on cloud-foundry managed by Bosh running on AWS. With our approach of detecting performance throttling, we were able to achieve better scalability. On Production systems, due to varying load - throughput, we measure the performance of tuning recommendations in terms of performance throttles hit on production systems. As the existing learning based method needs high quality samples from production system, proposed throttling detection engine enables it to do so. Hence, in this work we also achieve better throughput as throttling detection approach reduces corruption of learning methods.

In the coming future, we would like to explore more on using reinforcement learning methods to capture the performance throttles and making the current TDE free from static rules.

REFERENCES

[1] A. J. Storm, C. Garcia-Arellano, S. Lightstone, Y. Diao, and M. Surendra. Adaptive Self-tuning Memory in DB2. In VLDB, 2006.

[2] D. G. Sullivan, M. I. Seltzer, and A. Pfeffer. Using probabilistic reasoning to automate software tuning. In SIGMETRICS, 2004.

[3] D. N. Tran, P. C. Huynh, Y. C. Tay, and A. K. H. Tung. A new approach to dynamic self-tuning of database buffers. ACM Transactions on Storage, 4(1), 2008.

[4] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In Proc. of the 2017 ACM International Conference on Management of Data. ACM, 1009–1024.

[5] C. Curino, E. P. C. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In SIGMOD Conference, pages 313–324, 2011

[6] L. Ma, D. Van Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon, "Query-based Workload Forecasting for Self-Driving Database Management Systems," in Proceedings of the 2018 International Conference on Management of Data, 2018, pp. 631-645.

[7] J. S. Vitter. "Random sampling with a reservoir". ACM Transactions on Mathematical Software (TOMS), 11(1):37–57, 1985.

[8] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. "Self-driving database management systems". In CIDR, 2017

[9] A Pavlo, Evan P. C. Jones, and S Zdonik. 2011. "On predictive modeling for optimizing transaction execution in parallel OLTP systems". in Proc. VLDB Endow. 5, 2 (October 2011), pp. 85-96.

[10] D Basu, Q Lin, W Chen, H Tam Vo, Z Yuan, P Senellart, and S Bressan. "Cost-Model Oblivious Database Tuning with Reinforcement Learning". In Proceedings, Part I, of the 26th International Conference on Database and Expert Systems Applications - Volume 9261 (DEXA 2015), New York, NY, USA, pp 253-268.

[11] Facebook PSI, https://facebookmicrosites.github.io/psi/docs/overview

[12] B. Debnath, D. Lilja, and M. Mokbel. SARD: A statistical approach for ranking database tuning parameters. In ICDEW, pages 11–18, 2008.

[13] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with iTuned. VLDB, 2:1246–1257, August 2009.

[14] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood. Automatic performance diagnosis and tuning in oracle. In CIdR, 2005.

[15] S. Kumar. Oracle Database 10g: The self-managing database, Nov. 2003. White Paper

[16] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In SoCC. ACM, 338–350.

[17] Guoliang Li, Xuanhe Zhou, Shifu Li, Bo Gao. QTune: A QueryAware Database Tuning System with Deep Reinforcement Learning. PVLDB, 12(12): 2118 - 2130, 2019.

[18] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, and et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In SIGMOD, pages 415–432, 2019.

[19] B. Mozafari and et al. Performance and resource modeling in highly-concurrent oltp workloads. SIGMOD, pages 301–312, 2013.