Predicting In-Memory Database Performance for Automating Cluster Management Tasks

Jan Schaffner *, Benjamin Eckart #, Dean Jacobs *, Christian Schwarz *, Hasso Plattner #, Alexander Zeier #

# Hasso Plattner Institute
University of Potsdam
August-Bebel-Str. 88
14482 Potsdam, Germany
firstname.lastname@hpi.uni-potsdam.de

* SAP AG
Dietmar-Hopp-Allee 16
69190 Walldorf, Germany
firstname.lastname@sap.com

Abstract—In Software-as-a-Service, multiple tenants are typically consolidated into the same database instance to reduce costs. For analytics-as-a-service, in-memory column databases are especially suitable because they offer very short response times. This paper studies the automation of operational tasks in multi-tenant in-memory column database clusters. As a prerequisite, we develop a model for predicting whether the assignment of a particular tenant to a server in the cluster will lead to violations of response time goals. This model is then extended to capture drops in capacity incurred by migrating tenants between servers. We present an algorithm for moving tenants around the cluster to ensure that response time goals are met. In so doing, the number of servers in the cluster may be dynamically increased or decreased. The model is also extended to manage multiple copies of a tenant’s data for scalability and availability. We validated the model with an implementation of a multi-tenant clustering framework for SAP’s in-memory column database TREX.

I. INTRODUCTION

Analyses over large data sets that require a high degree of inter-node parallelism (e.g. log file processing at Facebook [1]) have been a major focus of recent research on cloud computing. Cloud computing is, however, equally attractive when many relatively small data sets need to be handled. An important case here is Enterprise Software-as-a-Service (SaaS), where a service provider develops an enterprise application and operates the system that hosts it for many businesses. Enterprise SaaS solutions commonly maintain data in a farm of conventional databases. To reduce total cost of ownership, multiple businesses are consolidated into each database instance, a technique referred to as multi-tenancy [2]. As an example, in October 2007, the SaaS CRM vendor RightNow had 3000 tenants distributed across 200 MySQL database instances with 1 to 100 tenants per instance [3].

In this setting, the database farm is exposed to continuous variations in the request rates of tenants, which are due to diurnal as well as longer-term usage patterns. For example, our experience at SAP with running a hosted business warehouse service [4] is that average request rates per tenant are twice as high in the week before the end of a quarter as in all other weeks. This ties in well with the promise of cloud computing to elastically scale computing resources to match demand over time.

Variations in request rates necessitate performing administrative tasks such as expanding or contracting the cluster and migrating tenants to balance the work across the servers. To keep costs low, a high degree of automation is required to avoid manual intervention by administrators. Automating cluster management requires profound knowledge about the effect that placing a tenant on a given server will have on that server’s ability to handle requests according to a desired service level objective (SLO).

In this paper, we make the following contributions. We develop a model for characterizing the load on an in-memory column database containing multiple tenants with different request rates. Previous cost models for in-memory databases are focused on characterizing the costs for individual queries [5], [6], while our model aims to characterize the database’s ability to handle an on-going stream of queries within pre-defined response time goals. More specifically, our model predicts how much load a server can sustain before query response times in the 99th percentile exceed one second (which is our SLO). We extend this model to capture drops in capacity incurred when tenants are migrated between servers and present an algorithm for moving tenants around the cluster to ensure that the SLO is met. We also extend the model to manage multiple copies of a tenant’s data for scalability and availability. We determine whether a placement of tenants on servers is “fail-safe” in the sense that the SLO is not violated in the event of a single server failure. We compare two replica placement strategies, mirroring and interleaving, in terms of the number of servers they require to mask failures with respect to the SLO.

Our approach to modeling database performance is experimental. Rather than characterizing all constituents of the system and their interactions, we extract the model from observations of a running system. As we will show in this paper, the resulting model is very accurate: our prediction of the response time in the 99th percentile is always within 10% of the actual value. The reason we are able to build
such a robust model using relatively simple tools is that in-memory column databases behave very linearly. We will show that response times can to a large extent be derived from the number of bytes the database scans in a given time interval regardless of the distribution of sizes and requests rates of the tenants. The ability to accurately predict response times makes it possible to run servers at a higher utilization level, thereby decreasing costs.

Our study is based on SAP’s in-memory column database TREX ([7], [8], [9]). To support SaaS, we developed a clustering infrastructure around TREX that replicates data for scalability and availability, supports multi-tenancy, and allows for dynamic cluster sizing. This work is part of a broader research project to adapt TREX to support on-demand data warehousing in a utility computing environment.

Note that there is considerable potential for consolidation through multi-tenancy even for in-memory column databases. As an example, we looked at one cube in the data warehouse of a large SAP customer, a Fortune 500 retail company. This cube contained all the sales records for three years and the fact table had approximately 360 million records. Using TREX’s standard dictionary compression, this cube consumed only slightly more than 2 GB of memory, which is low given modern server hardware. Moreover, SaaS often targets small to mid-sized businesses, which have orders of magnitude less data.

This paper is organized as follows: The next section introduces the TREX clustering infrastructure, called Rock, that is the basis for our experiments. Section III develops our basic model for predicting response times in the 99th percentile. Section IV extends this model to include tenant migration and Section V presents our migration algorithm. Section VI extends the model to manage multiple copies of a tenant’s data and discusses failure handling. Section VII presents related work. Section VIII concludes the paper.

II. THE ROCK CLUSTERING INFRASTRUCTURE

This section describes the Rock clustering infrastructure, which was used in the experiments in this paper. It also describes the benchmark that was used for the experiments as well as all other relevant parts of our experimental setup.

A. The Rock Clustering Framework

The Rock clustering framework runs in front of a collection of TREX servers and provides multi-tenancy, replication of tenant data, and fault tolerance. Figure 1 illustrates the Rock architecture. Read requests are submitted to the cluster by an analytics application. Write requests are submitted by batch importers, which periodically pull incremental updates of the data from transactional source systems. The Rock framework consists of three types of processes: the cluster leader, routers, and instance managers. Each instance manager is paired one-to-one with a TREX server to which it forwards requests.

The cluster leader exists only once in the landscape and assigns tenant data to instance managers. Each copy of a tenant’s data is assigned to one instance manager and each instance manager is responsible for the data of multiple tenants. The cluster leader maintains the placement information in a cluster map, which it propagates to the routers and instance managers so all components share a consistent view of the landscape.

The routers accept requests from outside the cluster and forward them to the appropriate instance managers. If a tenant has multiple replicas, the router balances the load across them. The load on a server is taken to be the number of queries being processed at a given point in time (across all tenants on the server).

We assume there is a single batch importer per tenant and that writes are sequentially numbered1. They can therefore be sequentially applied at every server and there are never inconsistencies due to failures, lost messages or updates arriving multiple times. A router may forward a write request to any one of the instance managers for a tenant, which then propagates the write to the other instance managers for that tenant. Our target application requires consistent reads across multiple queries when drilling down into a data set, and all requests within a drill-down are issued for the same version number. Rock supports this by using multi-version concurrency control (MVCC) based on snapshot isolation [11], which TREX implements natively. Note that the use of the replication functionality in Rock is not mandatory: in fact, it is only used for the experiments in Section VI of this paper.

When a write occurs, TREX adds the data to a per-column write buffer called a delta and logs it to disk. Queries are processed against both deltas and the actual columns. When a delta gets too large, it is merged into the columns: the columns are re-built to include the new data, the new columns are written to disk, and the deltas and log are cleared [12]. Recovery after a failure entails rebuilding the deltas from the log. Merges do not occur during our experiments and are not studied in this paper. Note that merging consumes a significant amount of resources, looks like a load burst, and is thus another reason to use replication.

1The batch importers need to maintain a consistent numbering scheme in the face of failure and recovery, which can be accomplished using algorithms such as Paxos [10].
According to [13], multi tenancy can be realized in the database by adopting a shared-machine, shared-process, or shared-table approach. The shared-table approach, where each table has a tenant_id column, can be made efficient if accesses are index-based. However analytic queries on column databases generally entail table scans, and scan times are proportional to the number of rows in the table. Rock therefore uses the shared-process approach and gives each tenant their own private tables.

B. Experimental Setup

The experiments in this paper are based on a modified version of the Star Schema Benchmark (SSB) [14], which is an adaptation of TPC-H [15].

To produce data for our experiments, we used the SSB data generator. The fact tables vary in size from 600,000 to 6,000,000 rows across tenants and the dimension tables increase linearly with the size of the fact tables. As a point of comparison, the large SAP customer described in Section I has a cube with 360 million rows. Using TREX’s standard dictionary compression, the fully-compressed data sets in our experiments consume between 25 and 204 MB of memory.

While TPC-H has 22 independent data warehousing queries, SSB has four query flights with three to four queries each. A query flight models a drill-down, i.e. all queries compute the same aggregate measure but use different filter criteria on the dimensions. This structure models the exploratory interactions of users with business intelligence applications. We modified SSB so all queries within a flight are performed against the same consistent TREX snapshot.

In our benchmark, each tenant has multiple concurrent users that submit requests to the system. Each user cycles through the query flights, stepping through the queries in each flight. After receiving a response, a user waits for a fixed think time before submitting the next query. To prevent caravanning, each user is offset in the cycle by a random amount. For the experiments presented in Section VI, a random think time drawn from a negative exponential distribution with a mean of 5 seconds was used.

The number of users per tenant is taken to be a relative size factor for that tenant times a system-wide user scale factor. Our experiments vary this scale factor to set the overall rate of requests to the system. Following [16], which studies web applications, we set the user think time to five seconds. This is perhaps too short for more complex applications, but the system behaves linearly in this respect: doubling the think time would double the maximum number of simultaneous users.

The batch importer for each tenant performs one write every five minutes. Each write makes a tenant’s fact table grow by 0.05% of its size at the beginning of the run. Writes for different tenants occur at different times, so the overall amount of data in the system gradually increases. The execution times of writes was not measured because they are submitted by a batch process that is not visible to users.

The first ten minutes of each benchmark run were cut off to ensure that the system is warmed up. The next ten minutes after the warm-up are called the benchmark period. All queries submitted after the benchmark period were cut off as well. All experiments were run on large memory instances on Amazon EC2, which have 2 virtual compute units (i.e. CPU cores) with 7.5 GB RAM each. For disk storage, we used Amazon EBS volumes, which offer highly-available persistent storage for EC2 instances.

III. The Basic Model: Bytes Scanned per Second

Throughout this paper, we use the term workload to refer to the actual amount of work a server receives\(^2\). The capacity of a server is reached when the workload becomes so high that response time goals are violated. We require that 99% of all queries have sub-second response times, which is analogous to the performance guarantee of Amazon Dynamo [17]. The query with the highest response time among 99% of all queries with the lowest response time is called the 99th percentile value.

We experimentally developed a model to determine how many tenants with different sizes and different request rates can be consolidated onto one server without violating the response time goal. Processing aggregation queries of larger tenants takes longer than processing aggregation queries of smaller tenants because more data needs to be scanned. We describe workload as a function of request rate and size for each tenant and show that it is related to the number of bytes that are scanned in a given time interval.

To examine the maximum possible workload depending on request rate and tenant size, we conducted several tests using the Star Schema Benchmark as described in Section II-A.

A. Relation of Request Rate and Tenant Size

In this experiment, a single server was packed with a group of equally-sized tenants and the total number of users was distributed equally among all tenants. There were no writes. The benchmark was run multiple times using different tenant sizes (denoted as \( t_S \)) and different request rates (\( t_R \)). The same fixed amount of memory was used in all cases so the configuration contained more or less tenants depending on the chosen size. Figure 2 shows the maximum request rate per tenant that could be achieved without violating the response time goal.

The maximum request rate decreases slightly exponentially in relation to the size of the individual tenants. Figure 2 shows the same dataset both on normal and logarithmic scales (note x2 and y2 axes). For the logarithmic plot, the shape of the curve is linear. The relation can thus be described as:

\[
\log(f(t_S)) = m \cdot \log(t_S) + n \tag{1}
\]

where the y-intercept \( n = 3.617 \) and the gradient \( m = -0.945 \) can be estimated using the Least-Squares Method. This equation can be rewritten as:

\[
t_R = \frac{1}{t_S^m} 10^n \tag{2}
\]

\(^2\)In particular, the term workload in this paper does not refer to the queries in our benchmark.
Based on this equation, Equation (3) defines the workload incurred by a tenant as a function of its request rate and size.

\[ w(t_R, t_S) := \frac{t_R t_S^{-m}}{10^n} \]  

Our goal is to calculate the workload on a server given arbitrary values for size and request rate. Therefore, the function is normalized such that \( w(t_R, t_S) = 1 \) denotes the point where the 99th percentile value exceeds the response time goal of 1000 ms. Decreasing \( t_R \) or \( t_S \) would result in a function value smaller than 1 and, consequently, the server would not violate the response time goal.

This equation shows that, to a large extent, the workload corresponds to the number of bytes that are scanned in a given time interval regardless of the distribution of sizes and request rates of the tenants. The relation is not exactly linear (\( m \) is not exactly -1): there is a slight advantage to processing a larger number of smaller requests. Our response time goal is violated when 1% of all queries have a response time of more than 1 second. The likelihood of a query being slower than 1 second increases as tenants become larger given the distributions of response times we observed during our experiments. The processing overhead for individual queries is hidden behind this phenomenon.

**B. Relation of Workload and 99th Percentile Value**

Typically, the tenants on a server will have different sizes and request rates. Equation 4 defines the total workload on a server as the sum of the workloads over the set of all tenants \( T = \{ (t_{S1}, t_{R1}), \ldots, (t_{Sn}, t_{Rn}) \} \) on the server.

\[ w(T) := \sum_{t \in T} \frac{t_R t_S^{-m}}{10^n} \]  

To analyze the relation between workload and 99th percentile value, we tested four server configurations with a different total data set size ranging from 1.5 to 3.2 GB. In each configuration, the amount of data and the request rate for each tenant varied. Again, the experiments were conducted on a single server without writes. The results are shown in Figure 3.

The predicted workload in Figure 3 was calculated using Equation (4). Interestingly, the shape of the graph is the same even when varying the tenant mix on a server. Up to a workload of 0.9, the graph is increasing linearly. Afterwards, it is increasing exponentially. We assume that the 99th percentile starts to increase exponentially at the point where request rates are so high that queries start to queue up in the database. The graph can be described as a function shown in Equation (5) with the parameters \( a = 333.982, b = 34.914, c = 2.537, d = 7, \) and \( e = 80.334 \), which have been estimated using regression.

\[ f(w) = aw + be^{cw^d} + e \]  

As can be seen in Figure 3, the predicted 99th percentile value is always close to the measured 99th percentile value. This shows that the definition of workload as shown in Equation (4) is also valid for 99th percentile values other than the maximum allowed value of 1000 ms and for server configurations containing tenants with different sizes.

**C. Accuracy of the Model**

To determine the accuracy of the model, we split the measured data into a training set and a test set. The training data was used to generate the model while the test data was used only to validate the model. Figure 4 shows the estimated 99th percentile value using the training data in relation to the measured 99th percentile value using the test data in terms of a Q-Q Plot.

Especially for low 99th percentile values, there is hardly any difference between the estimated and the predicted values. In the non-linear range of the function which starts at a 99th percentile value of approximately 400 ms (see Figure 3) the estimation is less precise. Predicting a value in the exponential
range of the function is more difficult, because even small variations can have a strong impact. However, in the relevant range up to a 99th percentile value of 1000 ms, the variance is always less than 10%. In comparison to more sophisticated machine learning algorithms, regression is a rather simple technique but can be calculated quite fast and proved to be capable of reliably predicting the 99th percentile value in our case.

D. Relation of Workload and 99th Perc. Value with Writes

As described in Section II-A, we configured the benchmark to perform one write for each tenant every five minutes. Each write made a tenant’s fact table grow by 0.05% of its size at the beginning of the run. The results across all four tenant mixes that were used in Figure 3 is shown in Figure 5.

The shape of the graph is similar to the one without writes but has different parameters \( a = 527.712, b = 22174.7, c = 0.337, d = 16, \) and \( e = -22067.7 \). The maximum possible read workload on a server with concurrent writes is reduced to \( w \approx 0.84 \). The reason for this behavior is that data is collecting in the delta, as described in Section II-A, and querying against the delta and the columns is more expensive than querying just against the columns. Recall that during our experiments, merging of the deltas into the columns is not performed.

IV. Migrations

The migration of tenants between servers consumes resources that would otherwise be available for query processing, which is especially crucial in overload situations. In this section, we study the degree to which migration impacts the maximum possible workload of a server and for how long a migration affects the performance of ongoing queries. Our results show that the size of the tenant being migrated does not affect how much capacity is lost during migration. Tenant size only affects the duration of a migration.

During a migration, different operations have to be performed on the source and destination servers. We therefore separately analyze the capacity drop on the source and destination servers.

A. Impact of Migration on Maximum Workload

In our experiments, a single tenant was repeatedly migrated from a source server to a destination server. To perform a migration, the tenant data was packed on the source server, transferred to the destination server, unpacked on the destination server, and loaded into memory on the destination server. In a production system, a complete migration would also entail deleting the tenant on the source server. In our experiments, the tenant was not deleted because it would otherwise not have been possible to repeat the migration process. Our measurements showed that deleting the tenant did not have a measurable impact on ongoing queries.

Figure 6(a) shows the 99th percentile value for different workloads with several migration sizes on the source server in comparison to the 99th percentile value without migrations. Figure 6(b) shows the same measurements for the destination server.

With ongoing migrations, the capacity drops to 0.85 on the source server and 0.82 on the destination server. When migrating differently-sized tenants, the shape of the graph is the same, which suggests that the actual size of the migrated tenant does not impact the 99th percentile value. Migrating a smaller tenant only leads to more migrations that can be performed in the same time period. A detailed discussion of the duration of tenant migrations is presented in the next subsection.

As the graphs in Figure 6 show, the 99th percentile value with migrations still increases linearly in the beginning and then increases exponentially, so Equation 5 again applies. In this case, the parameters for the equation are \( a = 408.25, b = 221.193, c = 3.991, d = 7, e = -59.435 \) on the source server and \( a = 389.158, b = 1184.98, c = 1.430, d = 7, e = -992.869 \) on the destination server.

The quality of the prediction can also be illustrated using a Q-Q plot. Figures 7(a) and 7(b) show the predicted and measured 99th percentile values for the source server and
destination server. As is the case without migrations, the quality of the prediction is very precise up to 600 ms for test and training data. Afterwards the prediction is less accurate, however it is never more than 15% off from the measured values in the relevant range up to 1000 ms.

B. Duration of Migration Impact

This section discusses the length of time during which a migration impacts the response times of read queries on the source and the destination server. Figure 8 shows the duration of impact of a migration on the destination server and the source server for a fixed workload of ongoing queries. During all tests, the query workload was approximately 0.6 (0.605 ± 0.015).

For a fixed query workload, the duration of impact of a migration increases linearly with the size of the tenant. This result shows that there is very little fixed overhead in performing a migration. Thus the costs of migrating two smaller tenants is the same as the costs of migrating a larger tenant of the same total size.

V. WORKLOAD BASED TENANT PLACEMENT

In the previous sections, we developed a workload model and estimated the capacity of a single server with and without migrations. As a next step, in this section we present a migration algorithm that uses this model to automatically react to server overload and underload conditions. Finally, we present an experimental evaluation of this algorithm.

A. Migration Algorithm

In contrast to data placement algorithms that reorganize the entire cluster all at once, our algorithm makes incremental changes to the layout to deal with individual server overload and underload conditions. Our approach is more appropriate in settings where request rates can change rapidly, since reorganizing the entire cluster can be a time-consuming operation. Moreover, like most highly-scalable cloud databases, our approach “keeps it simple” by avoiding multi-step administrative processes that must be made resilient to failures.

In Rock, the cluster leader orchestrates the migration process. It continuously monitors the workload on all servers and identifies when overload or underload conditions have occurred. When they occur, it runs our placement algorithm to determine where tenants should be migrated. If necessary, it starts up new servers. The cluster leader then issues commands to the individual servers to perform the migrations. Finally, it shuts down any unused servers.

In the read-only case, a server is judged to be in an overload condition when its workload reaches 0.85; beyond this point, it would be unable to sustain being the source server in a migration. If writes are present, this limit must
be correspondingly lowered. We chose to set the limit for underloads at a workload of 0.3: higher values would result in too many migrations.

In the read-only case, a server may be chosen as the destination for the migration of a tenant only if the workload of that server plus the workload of the tenant is less than 0.82. Again, this limit must be adjusted to accommodate writes. Ongoing migrations to a server temporarily reduce its capacity during the duration of the migrations, which must be taken into account in scheduling multiple simultaneous migrations.

The first step in handling an overloaded server is to choose the set of tenants to move away. Our algorithm selects the smallest tenants first, since that provides the maximum flexibility for placement and distribution. In addition, it allows us to migrate the minimum amount of data to correct the overload condition. As discussed in Section IV, the cost of migrating one larger tenant are the same as for migrating two smaller tenants with the same total size. Next, our algorithm tries to find a server with sufficient capacity for each of the selected tenants; this step is shown in CheckForMigr. We approximate a solution to this bin-packing problem using a variant of Johnson’s First Fit Decreasing [18] algorithm. Tenants are placed in order from largest to smallest on servers in order from least loaded to most loaded.

If there is insufficient capacity in the cluster as a whole to handle the selected tenants, as indicated by the failure of CheckForMigr, a new server is started and CheckForMigr is run again. At this point, the cluster leader initiates the migration of the tenants to their chosen new locations. In addition, if a new server was started, some large tenants are pro-actively migrated to the new server from the most heavily-loaded servers. This step helps to more rapidly balance the load across the cluster.

The algorithm for handling an underloaded server also uses CheckForMigr. In this case, the set of all tenants on the server is passed in as the first argument. If CheckForMigr successfully finds a location for all of the tenants, the migration is initiated and the original server is shut down. If CheckForMigr fails, the algorithm treats the server like a new server as described in the previous paragraph.

Algorithm 1 Check and Plan Migrations

```
1: function CHECKFORMIGR(tenants, currentSrv, otherSrvs)
2:     ORDERBYDESCWORKLOAD(tenants)
3:     for all tenant in tenants do
4:         ORDERBYASCWORKLOAD(otherSrvs)
5:         tenantsDontFit = true
6:         while tenantsDontFit and
7:             server ← next from otherSrvs do
8:             if DOES TENANT FIT(server, tenant) then
9:                 plannedMigr.append(server, tenant)
10:                server.workload ←
11:                    server.workload + tenant.workload
12:                tenantsDontFit ← false
13:            end if
14:        end while
15:        if tenantsDontFit then return
16:    end if
17: end for
18: return plannedMigr
19: end function
```

B. Evaluation

To validate our migration algorithm, we conducted experiments that demonstrate how the system responds to an overloaded and an underloaded server.

1) Overload Situation: In this experiment, we created a cluster containing one overloaded server and one newly-started server. We created the newly-started server in advance so its start up time would not impact the results about migration. Nevertheless, as is the case whenever a new server is started,
the system migrated more than the amount necessary to alleviate the overload condition in order to balance the load across the cluster. Figures 9(a) and 9(b) show the observed workload, the measured 99th percentile value, and the predicted 99th percentile value for these two servers.

The workload on the first server increases up to a value of 0.85 and then remains unchanged. When the value gets high enough, the algorithm starts to migrate tenants to the new server. The first vertical line at a benchmark runtime of 71 seconds indicates the start of the migration process while the second vertical line at 164 seconds indicates the end of the migration process. After migration starts, the observed workload on the overloaded server decreases in four steps, one for each tenant. At the same time, the observed workload on the newly-created server increases in four steps. In the end, the workload on the overloaded server is reduced to 0.49 and a load of 0.36 is moved to the newly-started server. The algorithm cannot achieve an equal load on both servers (which would amount to 0.425) because data of single tenants is never split. The predicted workload is very close to the measured workload on both servers.

2) Underload Situation: In this experiment, we created a cluster containing one server with a load of 0.22 and a second server with a load of 0.25. Figures 10(a) and 10(b) show the observed workload, the measured 99th percentile value, and the predicted 99th percentile value for these two servers.

The algorithm recognizes that all tenants from the first server could be handled by the second server and migrates them across. Altogether, nine tenants are migrated, as shown in the graphs. The predicted workload is very close to the measured workload on both servers except on the source server when all load is removed, which is an artifact of the model.

VI. WORKLOAD IN THE PRESENCE OF Failures

In the experiments presented so far, there was only a single copy of each tenant’s data in the cluster. However replicated data provides several advantages in our context: it provides more opportunities for distributing work across the cluster; it can mask server outages, both planned (upgrades) and unplanned (failures); and it can mask resource-intensive administrative operations such as migrations and the merge of deltas into columns.

In this section, we study strategies for maintaining multiple copies of the data. We motivate a replication technique called interleaving using an experiment which studies throughput in the face of failures. We also extend the workload model which has been developed in Section III to capture workload spikes due to failures.

A. Motivating Interleaved Tenant Placement

As stated in Section II-A, Rock uses an active/active load balancing scheme in the presence of multiple replicas. If a server goes down, the workload which was handled by this server is re-distributed to the servers holding the other copies of the tenants’ data. The re-distribution of workload in the event of a server failure differs depending on how the tenant replicas are assigned to the servers in the cluster. Using the off-the-shelf replication capabilities offered by most modern databases would result in replicating the data on the granularity of a whole server. In doing so, all tenants appearing together on one server will also co-appear on a second server in the cluster. This technique is often referred to as mirroring and is shown in Figure 11. The downside of mirroring is that in case of a failure all excess workload is re-directed to the other mirror. Therefore all servers in a mirrored setup must be 100% overprovisioned to handle failures. A technique for avoiding such hotspots is to use interleaving, also shown in Figure 11. Interleaving entails performing replication on the granularity of the individual tenants rather than all tenants inside a database process. This allows for spreading out the excess workload in case of a server failure across multiple machines in the cluster.

The following experiment demonstrates the impact of the chosen replica placement strategy on a cluster’s ability to serve queries without violating the SLO both during normal operations and failures. We set up a cluster with 100 tenants, where we put 10 tenants on each server. All tenants had exactly the same size (6 million rows in the fact table) and there were two copies per tenant, hence 20 servers in total. We assigned the tenant replicas to the server both using the mirrored strategy, where groups of 10 tenants where mirrored on one pair of servers each, and the interleaved strategy, where we manually laid out the tenants such that no two tenant replicas appear together on more than one server. We then ran both placement configurations under normal conditions and under failures. In the failure case, 1 out of the 20 TREX instances in the cluster was killed every 60 seconds with a mean time to recover of 30 seconds on average.

When an instance manager or a TREX instance in the cluster fail, Rock performs the following actions. First, the cluster leader detects the failure and detaches the EBS volume on which the TREX server stores its tables and log files. The EBS volume that persistently stores the tenant data assigned to the server is then detached from the failed machine and attached to a new server which is drawn from a small pool.
An instance manager and a TREX server are then started on the new server, the log on the EBS volume is processed and the server begins to accept requests. While the new server preloads the data into memory, all queries for the affected tenants are still directed to their replicas on the active servers. In case an instance manager was in the process of forwarding an update to the failed server, this update is now delivered to the new server.

Given the average recovery time in our experiment, 1 out of 20 servers was thus unavailable for approximately 50% of the benchmark period in the failure case. This is a very high failure rate which is unlikely to occur in practice. However, we are interested in modeling the worst case so as to achieve maximum robustness of our workload model (cf. Section IV, where the migration process was repeated throughout the whole duration of the benchmark run).

Table VI-A shows the results of the experiment: Even under normal operating conditions, interleaving achieves 7% higher throughput than mirroring. Our measure of throughput is the total number of concurrently active users per tenant that the cluster can serve without producing response times in the 99th percentile exceeding 1 second. The reason is that statistical variations become problematic when the number of concurrently active users is high. These variations create short-term load spikes, which the interleaved configuration spreads out better in the cluster than mirroring. As expected, the maximum throughput that the mirrored configuration can sustain in the failure case before an SLO violation occurs drops by almost 50% when compared to normal operations. Interleaving, in contrast, completely hides the failure situation from a throughput perspective. Notably, the interleaved configuration can even support 32 more users than the mirrored configuration without failures.

---

3 Instance pooling is necessary in a production setting because the time for starting up a fresh EC2 instance can take up to six minutes, while the rest of the recovery process takes less than 30 seconds on average.
B. Extending the Workload Model with Failures

In the previous sections we introduced a model to predict whether a server will be able to process requests without violating a pre-defined response time in the 99th percentile. In that case, the focus was on individual database instances. When trying to predict whether the SLO will be met during a failure, one would have to calculate the maximum workload that the individual servers in the cluster receive while a failed server is in the process of recovery. At a large cluster size, however, a single server producing high response times might not affect the response time in the 99th percentile across the whole cluster. The 99th percentile is thus not a good metric to determine the ability of a given placement (of tenants to servers) to handle failures. Yet, predicting the workload of the worst server in the cluster (i.e. the server with the highest workload in case of a failure) is useful: Ensuring that the worst server has a workload of less than \( w = 1 \) w.r.t. our model (see also Section III) is much more rigid than guaranteeing that the response time in the 99th percentile is less than 1 second across the whole cluster.

In the following, we will give a formulation for determining whether a given placement is failsafe in the sense that no server needs to sustain a workload greater than 1 in the event of single server failure. To denote that tenant \( t \) is assigned to server \( i \) we write:

\[
Y_{t,i}^{(k)} = \begin{cases} 
1 & \text{if } t \text{ is assigned to } i, \\
0 & \text{otherwise}
\end{cases}, \quad k \in \{0,1\}
\]

The variable \( k \) is used to number the two copies of a tenant’s data set. The total workload of server \( i \) can be calculated using Equation (6)\(^4\):

\[
w(i) = \sum_{t \in T} \sum_{k=0}^{1} Y_{t,i}^{(k)} \frac{R_t S^{-m}}{10^n}
\]

(6)

The total workload of server \( i \) in the face of a failure, which we denoted as \( \tilde{w}(i) \), can be calculated using Equation (7), where \( j \) denotes the failed server.

\[
\tilde{w}(i) = w(i) + \sum_{t \in T} Y_{t,j}^{(k)} Y_{t,j}^{(1-k)} \frac{R_t S^{-m}}{10^n}
\]

(7)

Figure 12 shows the workload of the worst server in the cluster in terms of our model over the benchmark period for both mirrored and interleaved placement. We show both the actually measured workload on the worst server as well as our prediction for the whole cluster according to Equation (7). The vertical lines denote points in time at which a failure has been inject into the cluster. Since the failures were injected in a round-robin fashion, multiple servers fill the role of the worst server during the course of the benchmark period. As in the previous experiment, all tenants had the same size and request rate. The request rate was also the same for both runs in the chart and was chosen such that the mirrored configuration is on the edge to violating the SLO. When looking at the worst server (i.e. the server with the highest workload) it becomes apparent that interleaving achieves a much better re-distribution of the excess load incurred by failures than mirroring. We also observe that the workload for the mirrored configuration is around \( w = 1 \), which—according to our model—is the maximum workload before the response time goal is violated. We can also see that our predictions for the workload on the worst server in case of a single server failure based on the placement are close to the workload that was actually observed on the servers during the test.

![Figure 12. Workload with Failures for Mirrored and Interleaved (Worst Server)](image)

VII. RELATED WORK

Related work on predicting database performance, such as Krompass et al. [19] and Tozer et al. [20], aims to predict query execution time for single queries under a certain workload in order to determine whether a new query should be admitted or rejected. Ganapathi et al. [21] use statistical machine learning to predict individual query response times for parallel data warehousing databases. Other cost models, especially for in-memory databases, also aim to model execution times for single queries [5], [6]. In contrast to those models, we are interested in whether the available resources are sufficient to achieve a defined SLO and predict the 99th percentile value instead of execution times of single queries.

Related work on automatic migration is mostly focused on moving virtual machines to other servers while guaranteeing service availability [22], [23] or migrating processes [24], which entails procedures for copying memory and processor state to other servers. Those approaches require that all tenants

<table>
<thead>
<tr>
<th></th>
<th>Mirrored</th>
<th>Interleaved</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal operations</td>
<td>4218 users</td>
<td>4506 users</td>
<td>7%</td>
</tr>
<tr>
<td>Periodical failure</td>
<td>2265 users</td>
<td>4250 users</td>
<td>88%</td>
</tr>
</tbody>
</table>

**TABLE I
MA**

The constants \( m \) and \( n \) are specified in Section III-A.

\(^4\)The constants \( m \) and \( n \) are specified in Section III-A.
get private VMs. In Rock, each VM serves multiple tenants and migration takes place on the granularity of a tenant rather than complete processes or virtual machines.

The placement of tenants to servers based on resource consumption is studied by Yang et al. [25]. Resources are not predicted using a model but are gathered in an observation period during which new tenants are first assigned to an empty machine to observe CPU cycles, memory and disk I/O. After the observation period is over, the tenant is allocated to a shared machine based on the observed resource consumption. Allocating tenants to machines is a bin-packing problem that is approximated using a simple algorithm called First Fit (see [26], [18]), which places a tenant on the first server where it fits. The automatic migration algorithm proposed in this paper uses a similar but improved algorithm called First Fit Decreasing. Note that only newly-arriving tenants are considered in [25], while our approach uses migration to re-allocate previously assigned tenants.

In the parallel databases Teradata [27], Gamma [28], and Bubba [29], the data placement problem entails distributing a fixed collection of relations across a fixed cluster of servers so as to minimize response times. Large relations are fully partitioned, also called fully declustered, across all servers and thus placement is straightforward. For small relations, the placement problem is NP-hard [30] and various heuristics have been proposed. In Bubba, small relations are placed in decreasing order of their access frequency, or “heat”. At each placement step, the algorithm tries to balance the overall heat at each server. In contrast to these approaches we are focused on using migration for making incremental changes to the cluster layout, since re-organizing the whole cluster during placement (and potentially re-partitioning the tables) is too costly given that request rates in our setting can change at fairly aggressive rates.

Rogers et al. [31] propose a framework for minimizing operational costs given a target query workload for a database running in an Infrastructure-as-a-Service environment. Their solution entails a generic problem constraint solver that is used for resource provisioning and online query routing. The problems are formulated as constraints and as a single optimization goal. Other approaches, like the work of Abrahao et al. [32], aim to perform self-adaptive capacity management for Internet services using a cost model featuring rewards and penalties for achieving a desired system throughput. To achieve defined high-level objectives, Kephart et al. describe the use of utility functions [33]. Their approach tries to distribute servers in a pool for different usages by optimizing this utility function. For instance, Paton et al. show how to use utility functions for capturing response time targets and cost for resources in order to enable adaptive workload execution in the cloud [34].

### VIII. Conclusion and Future Work

The long-term goal of our research is to study algorithms for automatic cluster management for multi-tenant in-memory analytics. In this paper, we made the following contributions towards this goal.

We developed a model for predicting the response time in the 99th percentile for an in-memory column database running a scan-intensive query workload. We showed how to use this model to predict whether a database instance will be able to meet a response time goal of 1 second in the 99th percentile given a particular assignment of tenants to this server. The parameters of the prediction model are the sizes and request rates of the tenants placed on a server, which corresponds to how many bytes an in-memory database instance needs to scan in a given interval. Our results show that the same 99th percentile value can be obtained for a set of tenants containing less data but high request rates and a setup with more data but lower request rates.

We extended the prediction model to capture migration of tenant data between servers in the cluster. Our results show that the size of the tenant being migrated does not affect how much capacity is lost during migration. Tenant size only affects the duration of a migration, in the sense that migration time increases linearly with the size of the tenant being migrated. As a consequence, the costs of migrating two smaller tenants are similar to the costs of migrating a larger tenant of the same total size. Based on the quantification of how expensive migration is for the involved servers, we proposed a simple first-fit online bin-packing strategy for automatically balancing the work across the servers in the cluster using migrations. Such automation is key in Enterprise SaaS environments, where manual administrator intervention accounts for large parts of the costs for running the service.

We further extended the model to capture replication and showed how our model can be used to predict whether a given placement of tenants to servers can sustain the failure of a single server without violating the response time goal. We demonstrated that the chosen strategy for assigning tenants to servers strongly impacts whether throughput can be kept up during a failure without violating the response time goal. In particular, the interleaved strategy allows for 88% more throughput than the mirrored strategy for an exemplary set of tenants that we experimentally studied.

As part of future work, we will study how automatic tenant placement and migration in the cluster can be done using interleaving such that load is balanced both during normal operations and in the case of failures. In practice, however, a perfect interleaving in the sense that no two copies of a tenant’s data set co-appear together on more than two servers is hard to achieve. This is especially true when tenant placement is done on-line and big cluster re-organizations are not permissible. In such a setting, only incremental changes are made to the layout, which can be done consciously during normal operations when the costs of migrations are known (as is the case with our prediction model). Many research projects have studied the problem of balancing the workload across all servers in the cluster as it can also be done using the model presented in this paper. It is, however, not clear how balancing of workload and achieving a good tenant interleaving—which we have shown to be a worthwhile property to optimize for during placement—can be done at the same time. While
moving a particular copy of a tenant between two servers could result in a more balanced workload across the servers, it could worsen the interleaving at the same time. The placement problem is even harder in a cloud computing environment, where the number of servers is flexible and can be controlled by the placement algorithm, since a cost dimension is added to the problem. Finding an automatic placement algorithm that takes all these properties into account is essentially a multi-dimensional optimization problem, which we plan to solve in future work.

REFERENCES


