# CONTENTS

# DBMS Performance Troubleshooting in Cloud Computing Using SQL Transaction Clustering

Arunprasad P. Marathe
Huawei Research Canada
Markham, Ontario, Canada
arun.marathe@huawei.com,ap.marathe@gmail.com

## ABSTRACT

Database management systems traditionally provide user-, table-, index-, and schema-level monitoring. For cloud-deployed applications, the research reported herein provides preliminary evidence that transaction cluster-level monitoring simplifies performance troubleshooting—especially for online transaction processing (OLTP) applications. Specifically, problematic rollbacks, performance drifts, system-wide performance problems, and system bottlenecks can be more easily debugged. The DBSCAN algorithm identifies transaction clusters based on transaction features extracted directly from a DBMS server—a job previously done using SQL log-mining. DBSCAN produces more accurate clusters when inter-transaction distances are computed using the angular cosine distance function (*ACD*) rather than the usual Euclidean distance function. Choice of *ACD* also simplifies DBSCAN parameter tuning—a task known to be nontrivial.

## 1 INTRODUCTION

A user books air tickets for himself and family members using an online web portal—front-end to a prototypical online transaction processing (OLTP) application. He makes several flight searches, books tickets, and provides frequent flyer numbers of the passengers. The airline reservation system implements this activity using a transaction. A second user performs different flight searches before booking a ticket for herself, but does not provide a frequent flyer number because it is not handy. The two transactions have both similarities and differences. Because they access the same tables in similar fashions, there may be a reason to believe that their performances are similar—a hypothesis that can be exploited if found true.

A study of the various applications contained in two popular OLTP benchmarking toolkits OLTP-Bench [2] and Sysbench [6] reveals that each application contains transactions that can be neatly divided into a small number of non-overlapping *transaction clusters* (between 1 and 10 for the two toolkits).

Self-similar transactions within a cluster differ in parameter values, statement orders, statement counts, statement types, rows read or updated, and so on. Nevertheless, this research shows that each cluster has a characteristic performance profile—termed its *signature*—at the level of which an OLTP application can be monitored. Sample cluster-levels metrics are average values of: transactions/sec (TPS); number of rows (read, updated, or sent to client); locking time; and so on.

Cluster-level monitoring is much simpler than transaction- or statement-level monitoring, and cluster count is independent of an OLTP application's load. Benefits of clustering multiply when that OLTP application is deployed in cloud where DBA's have to monitor performance of many applications simultaneously [17]. This paper demonstrates how transaction clustering helps a cloud DBA simplify debugging of several performance problems: identification of problematic transaction rollbacks; performance bottlenecks; system-wide performance issues; performance drifts; and so on. The cluster-level performance monitoring is not meant to replace existing tools: table-level and index-level data will continue to provide the necessary drill-downs, but help in determining where to drill-down should be valuable.

The DBSCAN algorithm [3] determines transaction clusters. DBSCAN is usually run with the Euclidean distance function, but this research demonstrates that when used with a normalized distance function called the *angular cosine distance* (*ACD*), DBSCAN finds more accurate clusters, and DBSCAN parameter tuning becomes easier—welcome news for a cloud DBA who cannot hand-tune the parameters for each OLTP application. DBSCAN parameter tuning is a known difficult task [4, 15], and therefore, suitability of *ACD* is a research contribution.

To calculate inter-transaction distances, transaction attributes are extracted into feature vectors. Previous research has relied on SQL log-mining for transaction feature extraction, whereas this research proposes to use simple server-side extensions instead. Regular-expression based SQL log mining is error-prone, and parsing SQL text may require parser duplication. Using server-side extensions, no (re)parsing of a SQL statement is required beyond the one initiated upon a statement's submission. A MySQL implementation demonstrates that server-side feature extraction is feasible. Similar infrastructure already exists in most modern DBMS engines (Oracle, SQL Server, PostgreSQL, and so on), and hence the solution has wider applicability.

## 2 SQL TRANSACTION CLUSTERS

SQL transactions within a cluster are similar (but not identical), and transactions in different clusters are dissimilar. Cluster determination is a three-step process. First, certain distinguishing attributes (called *features*) are extracted from a transaction, and an *n*-element *feature vector (FV)* is formed. Second, the distance between two transactions—defined to be the distance between their feature vectors—is computed using a distance function. Third, a clustering algorithm uses the feature vectors and the distance function to determine clusters.

### 2.1 Feature vector construction

In this research, extracting the following transaction features proved adequate.

(1) Statement type: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `COMMIT`, `ROLLBACK`, `BEGIN`, and so on.
(2) Table name(s)—possibly empty—referenced in the statement in 'schema.table' format.
(3) Counts associated with table names indicating frequency.

For other applications, different or additional features may need to be extracted.

A transaction $X$'s feature vector $FV(X)$ is a concatenation of four sub-vectors $FV_S$, $FV_I$, $FV_U$, and $FV_D$ for the four major SQL statement types SELECT, INSERT, UPDATE, and DELETE, respectively. All of the four sub-vectors are computed similarly, and therefore, only the construction of $FV_S$ is described.

$FV_S$ is of length $n$—the total number of tables in the OLTP system's schema, where each table is schema-qualified, and occupies a specific position in the vector to enable cross-feature vector comparisons. If a table $T$ is referenced by *all* of the SELECT statements in a transaction a total of $k$ times ($k \geq 0$), then the vector element for $T$ inside $FV_S$ has value $k$.

$FV_I$, $FV_U$, and $FV_D$ are computed similarly from all of the INSERT, UPDATE, and DELETE statements in a transaction.[1]

**Listing 1: Transaction $X_1$**

```
SELECT C_ID FROM Customer WHERE C_ID_STR = '50665'
SELECT * FROM Customer WHERE C_ID = 50665
SELECT * FROM Airport, Country WHERE AP_ID = 180 AND
    AP_CO_ID = CO_ID
SELECT * FROM Frequent_Flyer WHERE FF_C_ID = 50665
UPDATE Frequent_Flyer SET FF_IATTR00 = -14751,
    FF_IATTR01 = 8902 WHERE FF_C_ID = 50665 AND
    FF_AL_ID = 1075
UPDATE Customer SET C_IATTR00 = -14751, C_IATTR01 =
    89025 WHERE C_ID = 50665
COMMIT
```

Consider the transaction $X_1$—taken from the SEATS workload of [2]—shown in Listing 1.

- $FV_S(X_1) = [2, 1, 1, 1]$ because SELECT statements in $X_1$ refer to *Customer* table twice, and the other three tables once each.
- $FV_U(X_1) = [1, 1]$ because UPDATE statements in $X_1$ refer to two tables once each.
- $FV_I(X_1) = FV_D(X_1) = [\ ]$ because there are no UPDATE or DELETE statements in $X_1$.
- $FV(X_1) = [2, 1, 1, 1] + [\ ] + [1, 1] + [\ ] = [2, 1, 1, 1, 1, 1]$

Imagine a second transaction $X_2$ similar $X_1$ that references *Customer* only once. $FV(X_2)$ will be $[1, 1, 1, 1, 1, 1]$.

## 2.2 Angular cosine distance

Angular cosine distance, henceforth *ACD*, measures the distance between two transactions—in particular, between their feature vectors. For two $n$-dimensional vectors $\mathbf{A}$ and $\mathbf{B}$, each with indices $0, 1, \cdots n-1$, and with non-negative values, *ACD* is defined as follows [21].

$$ACD(\mathbf{A}, \mathbf{B}) = \frac{2}{\pi}\left(\cos^{-1}\left(\frac{\sum_{i=0}^{n-1} \mathbf{A}_i\mathbf{B}_i}{\sqrt{\sum_{i=0}^{n-1} \mathbf{A}_i^2}\sqrt{\sum_{i=0}^{n-1} \mathbf{B}_i^2}}\right)\right) \quad (1)$$

*ACD* is a distance measure or *metric*, and furthermore is unitary or normalized: $0.0 \leq ACD(\mathbf{A}, \mathbf{B}) \leq 1.0$. Because *ACD* distances are unitary, a closeness threshold *Eps* (for example, 0.2) can be defined so that two transactions at most *Eps* apart are considered 'close'; otherwise, they are declared 'far'. Normalized distance functions enable easy similarity definition: $similarity = 1.0 - distance$. 'Close' transactions have 0.8 (or 80%) similarity—something even a non-expert can understand.

For $X_1$ and $X_2$ of Section 2.1, $ACD(X_1, X2) = ACD([2, 1, 1, 1, 1.1], [1, 1, 1, 1, 1, 1]) = 0.197$, or they are $1.0 - 0.197 = 0.803$ (80.3%) similar which seems intuitively correct.

[1] INSERT, UPDATE, and DELETE statements can also have embedded SELECT queries, and those are handled similarly to the way $FV_S$ is.

Let $FV(X_3) = [1, 1, 1, 1, 1, 0]$. $X_3$ does not update the *Customer* table, and hence the 0. $X_3$ is somewhat dissimilar to $X_1$ and $X_2$, and indeed, $ACD(X_1, X_3) = 0.295$, or they are only 70.5% similar which again seems intuitive. With the closeness threshold *Eps* set to 0.2, $X_1$ and $X_2$ will be considered close, and may end up in the same cluster, whereas $X_1$ and $X_3$ will not belong to the same cluster.

## 2.3 DBSCAN parameter tuning

An open-source DBSCAN implementation [16]—instrumented to use *ACD*—performs transaction clustering. (By default, it uses the Euclidean distance function.) The author did not consider other clustering algorithms because DBSCAN has been found adequate for transaction clustering previously [11, 23].

DBSCAN's two tunable parameters *Eps* and *minPts* define *density*. A hyper-sphere of radius *Eps* with at least *minPts* points inside is considered *dense*. *ACD* does not eliminate hand-tuning DBSCAN parameters, but provides twofold help.

- *Eps* = 0.2 means that *independent of workload*, two transactions have to be at least 80% similar before they can belong to the same cluster. With such unnormalized distance functions as the Euclidean, *Eps* values are workload dependent.
- For many workloads, *ACD*-based DBSCAN clustering is not very sensitive to the two parameter values, and a good starting point is (*Eps, minPts*) = (0.2, 10). (Section 5.2.)

# 3 MySQL EXTENSIONS FOR TRANSACTION FEATURE EXTRACTION

Feature extraction uses the data from three in-memory tables shown in Fig. 1 that contain transaction-level, statement-level, and table-level information. The three tables will henceforth be referred to by the acronyms *e_t_h*, *e_s_h*, and *e_s_t*.

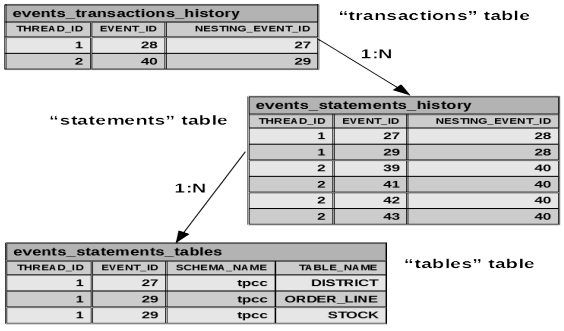


**Figure 1: Relationships between the three tables.**

Each completed transaction appears as a row in *e_t_h*, and the statements within are captured in *e_s_h* (1:N relationship). The newly added *e_s_t* table contains zero or more rows for each statement present in *e_s_h*—one for each distinct table reference in that statement (another 1:N relationship). Such statements as COMMIT and ROLLBACK do not refer to any tables, and therefore, have no presence in *e_s_t*. The columns in Fig. 1 only capture inter-table relationships. The other columns added to the three tables—not explicitly shown in Fig. 1—capture transaction-, statement-, and table-level statistics.

Using a SELECT query involving multi-way joins among *e_t_h*, *e_s_h*, and *e_s_t* tables, such information as transaction text; statement type; statement text; statement run-time; transaction

run-time; table names appearing in statement and their counts; number of rows examined; locking times; and so on is easily extracted.

Tables similar to the ones depicted in Fig. 1 already preexist (or can be easily created) in SQL Server [10], Oracle [12], and PostgreSQL [13], and therefore, server instrumentation of the kind described in this section is possible in those products.

## 4 SYSTEM ARCHITECTURE

A prototype SQL transaction clustering system has been implemented as depicted in Fig. 2. The Linux KVM virtual machine represents a hosted environment running a customer application (OLTP-Bench and Sysbench workloads during experimentation). The top MySQL instance within the Linux KVM has been instrumented as described in Section 3 to enable transaction-level data collection and feature extraction. The Windows 10 machine contains data processing components, including those performing transaction clustering and classification. (A second hosted application would require its own data processing node.)

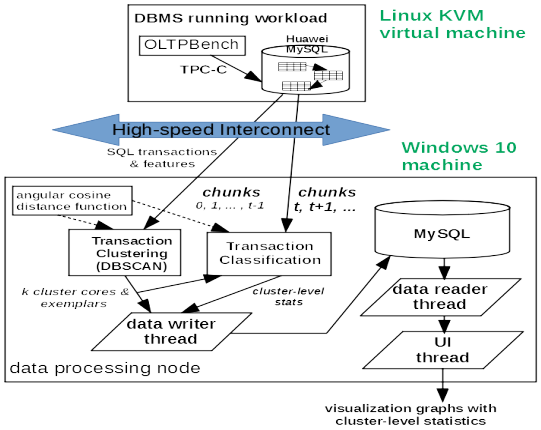


**Figure 2: Architecture of a transaction clustering system.**

A SQL query runs every 5 seconds on the Linux machine, and performs data collection. (The interval ensures minimal data collection overhead, but is a system parameter.) Each 5-second chunk includes feature and non-feature data: SQL statements; statement types; table names; table counts; statement durations; transaction duration; lock times; and rows examined by statements. An epoch-style Linux timestamp is associated with each chunk, and the resulting time-series is shipped to the data processing node where chunk-based iterators process it.

The first $t$ chunks are used to perform transaction clustering. During experimentation, the first 30 seconds worth of transactions ($t = 6$) were found adequate to determine transaction clusters, but $t$ is a system parameter. OLTP workloads do not have ad-hoc queries, and so clusters, once formed, should not change. If that assumption is violated, DBSeer's online implementation of DBSCAN can be used [5], but we leave that for future work.

Once clusters form, the rest of the streaming transactions are simply classified. The average distance (computed using *ACD*) from a transaction $X$ to a cluster's exemplars is calculated, and $X$ is assigned to the cluster for which that average distance is minimum, as long as that minimum value is no more than a threshold (say 0.2). If the threshold is exceeded, $X$ is declared an outlier.

Simple roll-up operations compute cluster-level statistics from transaction statistics as long as transactions are not outliers. If necessary, data about outliers can be captured and processed similarly.

## 5 CLUSTERING EXPERIMENTS

These experiments demonstrate that *ACD*-based DBSCAN is effective at finding transaction clusters, and is not very sensitive to *Eps* and *minPts* parameter values. Workload consists of OLTP-Bench [2] and Sysbench [6]. Out of OLTP-Bench's 15 workloads, the author was able to run 11.[2] Clustering effectiveness requires expected cluster counts which OLTP-Bench already provides, and were manually determined for Sysbench. In Tables 1 and 2, expected cluster counts are indicated in brackets after the workload names. Actual cluster counts are not always integral because each value is an average of 5 runs, and DBSCAN sometimes produces slightly different cluster counts for different samples.

### 5.1 *ACD*-based DBSCAN

*ACD*-based DBSCAN with (*Eps*, *minPts*) = (0.2, 10)—henceforth, $ACD(0.2, 10)$—is an excellent starting point for clustering database transactions. Table 1 captures $ACD(0.2, 10)$'s performance against a baseline provided by the well-known Euclidean distance function. In the Euclidean baseline, *Eps* and *minPts* values are set using a heuristic provided by the DBSCAN authors in a subsequent paper [14]. We will term the resulting baseline *SEKX* after the author names. The heuristic itself works as follows. Let the *dimensionality* of a workload—defined to be maximum feature vector length—be *DIM*. Then:

- Heuristic value of $Eps = (2 * DIM) - 1$
- Heuristic value of $minPts = (2 * DIM)$

$ACD(0.2, 10)$ handily outperforms *SEKX* in 8 out of 13 workloads, and equally important, is never worse than *SEKX* in the remaining 5 workloads. For 9 workloads, *SEKX* puts all of the transactions into single clusters, and hence is ineffective.

The following observations—cross-referenced in the 'Comments' column of Table 1—provide reasons why $ACD(0.2, 10)$'s cluster counts are slightly off in a few cases.

(1) Epinions cluster count is off by 1 because a transaction type selects from *Review* and *Trust* tables separately, and another type selects from their joined version. Both end up in the same cluster because the feature vector construction in Section 2.1 currently does not distinguish them.
(2) YCSB cluster count is off by 1 because two of the transaction types have point and range selects, but are otherwise identical, and that difference is currently not captured as a feature.
(3) AuctionMark and SEATS produced the correct cluster counts with $ACD(0.15, 10)$ and $ACD(0.15, 15)$, respectively.

Observations 1 and 2 suggest possible features that can be extracted, and added to the feature vector.

### 5.2 Sensitivity analysis of *ACD*

When used along with *ACD*, DBSCAN is not very sensitive to various *Eps* and *minPts* values. Table 2 captures *ACD* results for 6 sets of parameters. Approximately, cluster membership criterion becomes more stringent as one reads across a row, and hence cluster counts can be expected to diminish from left to right. As can be seen, *ACD* is not very sensitive to parameter values for most of

[2]Wikipedia turns out to be hard to cluster because two of the transaction types have frequencies of only 0.07% each, and are rarely present in samples. One can ignore 'Wikipedia' results, but they are included for completeness.

**Table 1: *ACD* (0.2, 10) versus *SEKX* ($2 * DIM - 1$, $2 * DIM$)**

| Workload & expected cluster count | DIM | Avg. cluster count | | Comments |
|---|---|---|---|---|
| | | *ACD* | *SEKX* | |
| AuctionMark (9) | 9 | 9.4 | 1 | Observation 3 |
| Epinions (9) | 2 | 8 | 1 | Observation 1 |
| SEATS (6) | 8 | 7 | 1 | Observation 3 |
| SIBench (2) | 1 | 2 | 2 | |
| SmallBank (6) | 5 | 6 | 1 | |
| sysbench_ro (10) | 1 | 10 | 10 | |
| sysbench_rw (10) | 5 | 10 | 10 | |
| TATP (7) | 3 | 7 | 1 | |
| TPC-C (5) | 10 | 5 | 1 | |
| Twitter (5) | 2 | 5 | 1 | |
| Voter (1) | 4 | 1 | 1 | |
| Wikipedia (5) | 7 | 2 | 1 | Footnote 2 |
| YCSB (6) | 2 | 5 | 1 | Observation 2 |

the workloads. Even when it is (AuctionMark and sysbench_rw; and to a lesser extent SEATS and TPC-C), graceful degradation is observed. Therefore, it is not crucial for a DBA to get the values of *Eps* and *minPts* spot on, and any reasonable values should perform respectably well. 'Sysbench_rw' is tricky to cluster for its highly symmetrical transactions—all of the transactions are roughly equidistant from all of the other transactions—but two *ACD* configurations perform well.

**Table 2: *ACD* with different *Eps* and *minPts* values.**

| Workload & expected cluster count | Avg. cluster count | | | | | |
|---|---|---|---|---|---|---|
| | *ACD* (.20,10) | *ACD* (.15,10) | *ACD* (.15,15) | *ACD* (.10,10) | *ACD* (.10,15) | *ACD* (.05,20) |
| AuctionMark (9) | 9.4 | 9.4 | 5.4 | 8.2 | 5.4 | 4 |
| Epinions (9) | 8 | 8 | 8 | 8 | 8 | 8 |
| SEATS (6) | 7 | 7.4 | 6 | 8.4 | 7 | 5.2 |
| SIBench (2) | 2 | 2 | 2 | 2 | 2 | 2 |
| SmallBank (6) | 6 | 6 | 6 | 6 | 6 | 6 |
| sysbench_ro (10) | 10 | 10 | 10 | 10 | 10 | 10 |
| sysbench_rw (10) | 10 | 9 | 0.6 | 0 | 0 | 0 |
| TATP (7) | 7 | 7 | 7 | 7 | 7 | 7 |
| TPC-C (5) | 5 | 6 | 5.8 | 6 | 5.8 | 4.8 |
| Twitter (5) | 5 | 5 | 5 | 5 | 5 | 5 |
| Voter (1) | 1 | 1 | 1 | 1 | 1 | 1 |
| Wikipedia (5) | 2 | 2 | 2 | 2 | 2 | 2 |
| YCSB (6) | 5 | 5 | 5 | 5 | 5 | 5 |

# 6 PERFORMANCE TROUBLESHOOTING USING TRANSACTION CLUSTERING

Experiments in this section demonstrate how cluster-level performance monitoring simplify debugging of several real-life problems faced by cloud DBA's.

## 6.1 Cluster signatures

OLTP-Bench's SmallBank workload simulates some operations of a bank, and produces the cluster signatures shown in Fig. 3 with 10-terminal workload, and scale-factor of 1. The three sub-plots capture the average values of the three transaction-cluster-level metrics: lock time (in ms); number of rows examined; and TPS

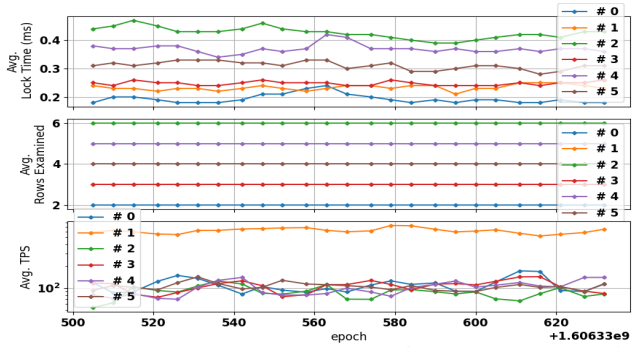


**Figure 3: SmallBank's cluster signatures (10-terminal workload).**

(transactions/sec). Each sub-plot contains six lines—one for each of the transaction clusters identified.[3]

From SmallBank cluster signatures, a cloud DBA can learn several things about the workload. First, six clusters means that there are six types of transactions in SmallBank—as OLTP-Bench confirms. Second, cluster signatures are discernible. Third, small signature variations exist because each data point aggregates 5 seconds worth of transactions (which themselves execute in a multi-tasking environment). Fourth, all of the transactions in a given cluster examine the same number of rows—a SmallBank peculiarity. Fifth, cluster exemplars reveal that the only cluster with read-only transactions is cluster 1—explaining its its highest TPS values. Sixth, when 100 terminals generate workload, the same six clusters form (graphs not included to save space), thereby confirming that clusters are load independent—a practically useful promise of clustering.

## 6.2 Identification of transaction rollbacks

Transaction rollbacks are normal DBMS occurrences, but sometimes their frequencies become problematic. If rollbacks are limited to a transaction type, cluster-level monitoring helps because unexpected additional clusters form.

The TPC-C benchmark [20] has five well-known transaction types. Rollbacks are demonstrated using the *Payment* transaction by modifying its code such that after submitting 2 out of its 7 statements, it rolls back with 20% probability—simulating a problematic high-frequency rollback. To make example even more realistic, a second rollback—simulating a normal and rare DBMS occurrence—happens after the sixth statement with 0.1% probability (overall probability $0.8 \times 0.1 = 0.08\%$). Because the problematic rollbacks are numerous, they should form their own cluster, whereas the normal rollbacks should not.

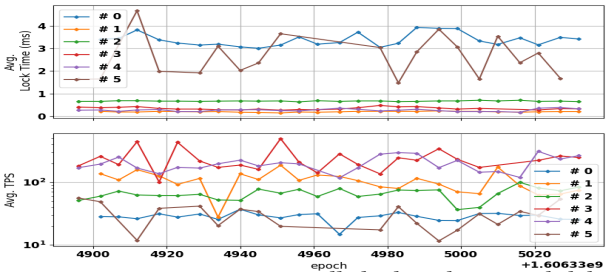


**Figure 4: *Payment* transaction rolls back with 20% probability causing an unexpected sixth cluster (Cluster 1) to appear.**

[3]The second subplot contains only five lines because clusters 1 and 3 examine 3 rows each, and the plotting software cannot distinguish two overlapping lines.

The modified TPC-C benchmark produces the results shown in Fig. 4. Usually, TPC-C produces five clusters, but six are found. A sample exemplar from Cluster 1 reveals the 'incomplete' *Payment* transaction with a telltale `rollback` issued after two statements.

```
UPDATE WAREHOUSE SET W_YTD = W_YTD + 1704.68994140625
    WHERE W_ID = 2
SELECT W_STREET_1, W_STREET_2, W_CITY, W_STATE, W_ZIP,
    W_NAME FROM WAREHOUSE WHERE W_ID = 2
rollback
```

The normal (rare) DBMS rollbacks (0.08% probability) do not form a cluster because they do not meet DBSCAN's density requirement mentioned in Section 2.3. High-frequency rollbacks are difficult to identify if cluster-level statistics are not kept. Applications often resubmit transactions in case of rollbacks, and users only notice and wonder about degraded performance. An incident report would cause DBA's or programmers to dig through voluminous logs to even begin suspecting a culprit.

## 6.3 Performance drift

Performance drift refers to a situation in which one (or just a few) cluster's performance drifts from its norm. Many situations can cause performance drifts. Here is a typical one: A DBA might forget to reinstate an index that (s)he deliberately dropped during a bulk load operation.

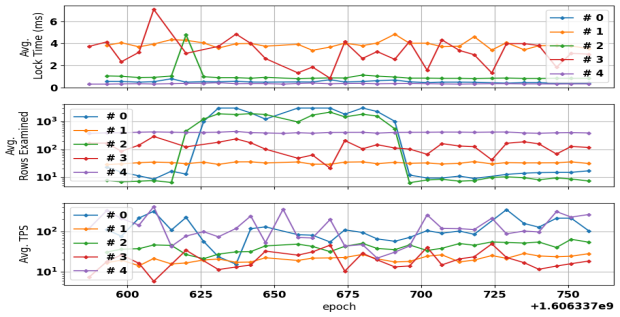


**Figure 5: An index made invisible during [615, 685].**

To simulate a performance drift, a secondary index (used by two out of the five TPC-C transactions) on the *CUSTOMER* table is made invisible[4] to the query optimizer during a portion of the run. When the index is made unavailable, the query optimizer has to use table scans instead of index seeks. As can be seen in Fig. 5, average row counts show dramatic increases (drifts) for clusters 0 and 2 during the interval [615, 685].[5]

When the performance of only one cluster drifts, objects related to only that cluster (e.g., tables, indexes, statistics) are good starting points for debugging. Without cluster-level statistics, such a diagnosis may require considerably more work.

## 6.4 System-wide performance problem

System-wide performance problems are caused by such things as a failed network card, operating system reboot, failed disk, and runaway process hogging CPU's. If all of the clusters experience simultaneous degraded performances, a system-wide issue may be the cause. One such situation is created using a CPU-hogging program that spawns as many processes as the number of CPU cores on the computer (8), and then making them run infinite 'while' loops—thereby creating a CPU bottleneck.

[4]MySQL command used was: `ALTER TABLE CUSTOMER ALTER INDEX IDX_CUSTOMER_NAME INVISIBLE;`

[5]As an aside, if an invisible index makes no difference to any cluster's performance, it might be safe to drop—the reason why that feature was added to MySQL.

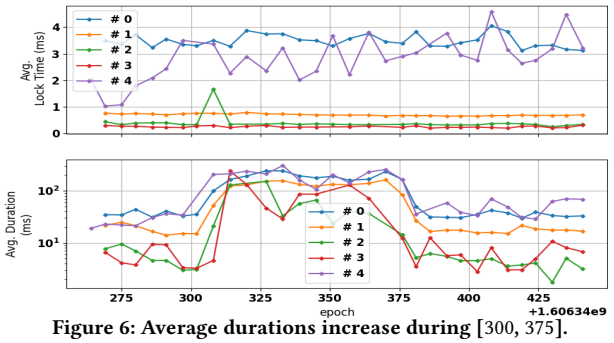


**Figure 6: Average durations increase during [300, 375].**

In the resulting graphs shown in Fig. 6, during the interval [300, 375], average durations of all of the clusters show unmistakable jumps. After about 375, when the offending program is killed, all five average durations return to their baseline values. Interestingly, average lock times are largely unaffected, indicating that for the few transactions that did manage to execute during CPU saturation, lock time did not take a hit. Such observations should provide the DBA a good starting point to formulate a hypothesis before beginning a detailed investigation.

## 6.5 Bottleneck analysis

Cloud applications run on pre-provisioned VM's. Because application behaviour is relatively unknown, a bottleneck may develop—in CPU, memory, disk I/O, network I/O, and so on. Furthermore, bottlenecks may vary by transaction types. Non-cloud DBA's are used to monitoring such operating system-level performance counters as *vmstat*, *iostat*, and *netstat* in Linux for bottleneck identification, but cluster-level statistics offer a complementary method that can provide additional help.

To study whether a VM has sufficient memory, its memory is reduced on the fly from 16 GB to 3 GB while TPC-C workload runs. Such a drastic change in memory allocation is only for demonstration: typical changes should be much smaller.

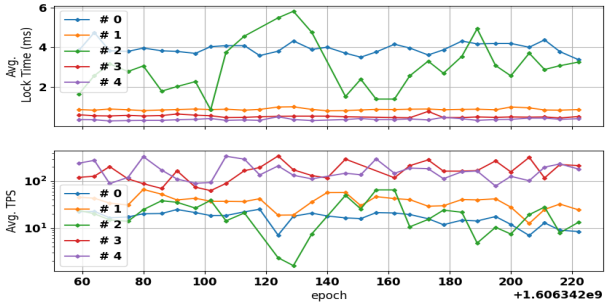


**Figure 7: Memory reduces from 16 GB to 3 GB at timestamp 120.**

In the results captured in Fig. 7, memory reduction happens at timestamp 120 onward. The average TPS values before and after that interval show no discernible changes. There is a noticeable drop at 120 as the operating system seems to adjust to the new memory setting, but soon, normal service resumes. The 'Avg. lock time' metric is also mostly unaffected, and therefore, one can conclude that this VM is well-provisioned for memory.

In the next variation, CPU is constrained. Changing CPU count in KVM requires a machine restart, and therefore, an approach similar to the one in Section 6.4 is taken, except that 7 out of the

8 cores are kept busy running infinite 'while' loops. The results appear in Fig. 8.

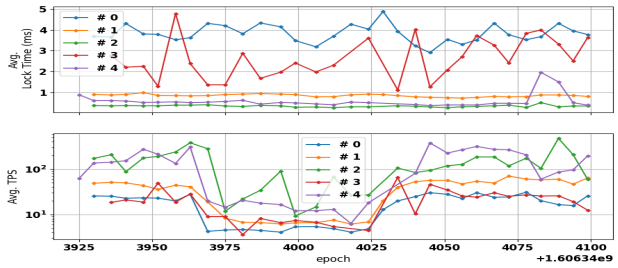


**Figure 8: CPU saturation during the interval [3960, 4025].**

The CPU bottleneck spans the interval [3960, 4025] during which reduced 'Avg. TPS' values are visible. The highest 'Avg. TPS' values are for clusters 2 and 4 (read-only transactions *Stock-Level* and *Order-Status*, respectively). Outside of the CPU bottleneck, those values are somewhat close, but during the bottleneck, *Order-Status* transaction's performance takes a bigger hit (Y-axis is log-scale) suggesting *Order-Status* is much more sensitive to CPU than *Stock-level* is in this environment. If *Order-Status* is deemed important (say because customers check statuses their orders often), it may make sense to over-provision for CPU rather than for memory if a choice is to be made between the two.

## 7 RELATED WORK

Identifying transaction clusters from SQL text arriving at a database server is important for two reasons. First, applications deployed in cloud environments are often web-applications [17] using object-relational mappings to submit SQL queries, and do not use stored procedures. Second, as noted by Stonebraker *et al.* [19], because of SQL's 'one language fits all' approach, transaction code may use a mix of stored procedures, prepared statements, and Java/C++/C# code.

Clustering itself is a broad and well-studied topic [22]. SQL query clustering and classification has been studied under two granularities: query-level and transaction-level. SQL query features previously tried include terms in `SELECT`, `JOIN`, `FROM`, `GROUP BY`, and `ORDER BY` clauses, table names, column names, normalized estimated execution costs [7, 9]; and features have been converted into vectors, graphs, or sets [7, 18]. As a general observation, fewer features suffice in self-similar OLTP workloads; ad-hoc workloads require more features. Such distance functions as cosine, Jaccard, and Hamming have been tried for clustering SQL queries [1, 9, 18], although only the Euclidean has been tried at the transaction level before [5]. Before this research, feature extraction has mined SQL text from DBMS logs [9, 18], or MaxScale proxy server [11]; server-side feature extraction is novel.

## 8 CONCLUSIONS AND FUTURE WORK

This research makes a case that in addition to user, table, index, and schema level monitoring provided, DBMS's should start to provide transaction-cluster-level monitoring. In applications deployed in the cloud, and for OLTP workloads, that additional level simplifies debugging of performance problems: unexpected transaction rollbacks, performance drifts, bottleneck identifications, and so on. Angular cosine distance-based DBSCAN is an improvement over Euclidean-based DBSCAN with *SEKX* heuristic [14] (better clusters and simplified DBSCAN parameter tuning).

Future work may investigate the following features for transaction clustering: column names to possibly identify index issues; predicate types (point queries vs. range queries) and counts; access paths used; isolation level; number of sorts; join tables; and so on. Multiple cluster-level signatures may help because an application may have distinct 'peak' and 'off-peak' behaviours. Whether *ACD* is suitable with such clustering algorithms as BIRCH [24] and $k$-means [8] remains to be seen. Server-side feature extraction can be attempted in other modern database systems using minor extensions to preexisting scaffoldings.

## ACKNOWLEDGMENTS

Matthew Van Dijk implemented the server-side extensions needed for transaction feature extraction. Anonymous reviewers provided valuable feedback.

## REFERENCES

[1] Rakesh Agrawal, Ralf Rantzau, and Evimaria Terzi. 2006. Context-sensitive ranking. In *Proceedings of the SIGMOD 2006 Conference*. 383–394.
[2] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288.
[3] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*. 226–231.
[4] Junhao Gan and Yufei Tao. 2015. DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation. In *Proceedings of the SIGMOD 2015 Conference*. 519–530.
[5] GitHub. 2020. *DBSeer*. Retrieved September 17, 2020 from https://github.com/barzan/dbseer
[6] GitHub. 2020. *sysbench*. Retrieved August 5, 2020 from https://github.com/akopytov/sysbench
[7] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* 12, 12 (2019), 2118–2130.
[8] Stuart P. Lloyd. 1982. Least squares quantization in PCM. *IEEE Trans. Inf. Theory* 28, 2 (1982), 129–136.
[9] Vitor Hirota Makiyama, Jordan Raddick, and Rafael D. C. Santos. 2015. Text Mining Applied to SQL Queries: A Case Study for the SDSS SkyServer. In *SIMBig*.
[10] Microsoft. 2019. *System Dynamic Managemenet Views*. Retrieved August 4, 2020 from https://docs.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/system-dynamic-management-views?view=sql-server-ver15
[11] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. 2013. Performance and resource modeling in highly-concurrent OLTP workloads. In *Proceedings of the SIGMOD 2013 Conference*. 301–312.
[12] Oracle. 2020. *About Dynamic Performance Views*. Retrieved September 22, 2020 from https://docs.oracle.com/cd/B19306_01/server.102/b14237/dynviews_1001.htm#i1398692
[13] PostgreSQL. 2020. *The Statistics Collector*. Retrieved August 4, 2020 from https://www.postgresql.org/docs/9.6/monitoring-stats.html
[14] Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. 1998. Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and Its Applications. *Data Min. Knowl. Discov.* 2, 2 (1998), 169–194.
[15] Erich Schubert, Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Trans. Database Syst.* 42, 3 (2017), 19:1–19:21.
[16] Scikit Learn. 2019. *DBSCAN*. Retrieved August 2, 2020 from https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html
[17] Alexandre Verbitski *et al.* 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the SIGMOD 2017 Conference*. ACM, 1041–1052.
[18] Gökhan Kul *et al.* 2018. Similarity Metrics for SQL Query Clustering. *IEEE Trans. Knowl. Data Eng.* 30, 12 (2018), 2408–2420.
[19] Michael Stonebraker *et al.* 2007. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the VLDB 2007 Conference*. 1150–1160.
[20] Transaction Processing Performance Council 1992. *TPC-C*. Retrieved September 22, 2020 from http://www.tpc.org/tpcc/
[21] Wikipedia. 2019. *Cosine similarity*. Retrieved August 11, 2020 from https://en.wikipedia.org/wiki/Cosine_similarity
[22] Dongkuan Xu and Yingjie Tian. 2015. A Comprehensive Survey of Clustering Algorithms. *Annals of Data Science* 2 (2015), 165–193.
[23] Dong Young Yoon, Ning Niu, and Barzan Mozafari. 2016. DBSherlock: A Performance Diagnostic Tool for Transactional Databases. In *Proceedings of the SIGMOD 2016 Conference*. 1599–1614.
[24] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. 1997. BIRCH: A New Data Clustering Algorithm and Its Applications. *Data Min. Knowl. Discov.* 1, 2 (1997), 141–182.

# CONTENTS

# DBMS Performance Troubleshooting in Cloud Computing Using SQL Transaction Clustering

Arunprasad P. Marathe
Huawei Research Canada
Markham, Ontario, Canada
arun.marathe@huawei.com,ap.marathe@gmail.com

## ABSTRACT

Database management systems traditionally provide user-, table-, index-, and schema-level monitoring. For cloud-deployed applications, the research reported herein provides preliminary evidence that transaction cluster-level monitoring simplifies performance troubleshooting—especially for online transaction processing (OLTP) applications. Specifically, problematic rollbacks, performance drifts, system-wide performance problems, and system bottlenecks can be more easily debugged. The DBSCAN algorithm identifies transaction clusters based on transaction features extracted directly from a DBMS server—a job previously done using SQL log-mining. DBSCAN produces more accurate clusters when inter-transaction distances are computed using the angular cosine distance function (*ACD*) rather than the usual Euclidean distance function. Choice of *ACD* also simplifies DBSCAN parameter tuning—a task known to be nontrivial.

## 1 INTRODUCTION

A user books air tickets for himself and family members using an online web portal—front-end to a prototypical online transaction processing (OLTP) application. He makes several flight searches, books tickets, and provides frequent flyer numbers of the passengers. The airline reservation system implements this activity using a transaction. A second user performs different flight searches before booking a ticket for herself, but does not provide a frequent flyer number because it is not handy. The two transactions have both similarities and differences. Because they access the same tables in similar fashions, there may be a reason to believe that their performances are similar—a hypothesis that can be exploited if found true.

A study of the various applications contained in two popular OLTP benchmarking toolkits OLTP-Bench [2] and Sysbench [6] reveals that each application contains transactions that can be neatly divided into a small number of non-overlapping *transaction clusters* (between 1 and 10 for the two toolkits).

Self-similar transactions within a cluster differ in parameter values, statement orders, statement counts, statement types, rows read or updated, and so on. Nevertheless, this research shows that each cluster has a characteristic performance profile—termed its *signature*—at the level of which an OLTP application can be monitored. Sample cluster-levels metrics are average values of: transactions/sec (TPS); number of rows (read, updated, or sent to client); locking time; and so on.

Cluster-level monitoring is much simpler than transaction- or statement-level monitoring, and cluster count is independent of an OLTP application's load. Benefits of clustering multiply when that OLTP application is deployed in cloud where DBA's have to monitor performance of many applications simultaneously [17]. This paper demonstrates how transaction clustering helps a cloud DBA simplify debugging of several performance problems: identification of problematic transaction rollbacks; performance bottlenecks; system-wide performance issues; performance drifts; and so on. The cluster-level performance monitoring is not meant to replace existing tools: table-level and index-level data will continue to provide the necessary drill-downs, but help in determining where to drill-down should be valuable.

The DBSCAN algorithm [3] determines transaction clusters. DBSCAN is usually run with the Euclidean distance function, but this research demonstrates that when used with a normalized distance function called the *angular cosine distance* (*ACD*), DBSCAN finds more accurate clusters, and DBSCAN parameter tuning becomes easier—welcome news for a cloud DBA who cannot hand-tune the parameters for each OLTP application. DBSCAN parameter tuning is a known difficult task [4, 15], and therefore, suitability of *ACD* is a research contribution.

To calculate inter-transaction distances, transaction attributes are extracted into feature vectors. Previous research has relied on SQL log-mining for transaction feature extraction, whereas this research proposes to use simple server-side extensions instead. Regular-expression based SQL log mining is error-prone, and parsing SQL text may require parser duplication. Using server-side extensions, no (re)parsing of a SQL statement is required beyond the one initiated upon a statement's submission. A MySQL implementation demonstrates that server-side feature extraction is feasible. Similar infrastructure already exists in most modern DBMS engines (Oracle, SQL Server, PostgreSQL, and so on), and hence the solution has wider applicability.

## 2 SQL TRANSACTION CLUSTERS

SQL transactions within a cluster are similar (but not identical), and transactions in different clusters are dissimilar. Cluster determination is a three-step process. First, certain distinguishing attributes (called *features*) are extracted from a transaction, and an *n*-element *feature vector (FV)* is formed. Second, the distance between two transactions—defined to be the distance between their feature vectors—is computed using a distance function. Third, a clustering algorithm uses the feature vectors and the distance function to determine clusters.

### 2.1 Feature vector construction

In this research, extracting the following transaction features proved adequate.

(1) Statement type: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `COMMIT`, `ROLLBACK`, `BEGIN`, and so on.
(2) Table name(s)—possibly empty—referenced in the statement in 'schema.table' format.
(3) Counts associated with table names indicating frequency.

For other applications, different or additional features may need to be extracted.

A transaction $X$'s feature vector $FV(X)$ is a concatenation of four sub-vectors $FV_S$, $FV_I$, $FV_U$, and $FV_D$ for the four major SQL statement types SELECT, INSERT, UPDATE, and DELETE, respectively. All of the four sub-vectors are computed similarly, and therefore, only the construction of $FV_S$ is described.

$FV_S$ is of length $n$—the total number of tables in the OLTP system's schema, where each table is schema-qualified, and occupies a specific position in the vector to enable cross-feature vector comparisons. If a table $T$ is referenced by *all* of the SELECT statements in a transaction a total of $k$ times ($k \geq 0$), then the vector element for $T$ inside $FV_S$ has value $k$.

$FV_I$, $FV_U$, and $FV_D$ are computed similarly from all of the INSERT, UPDATE, and DELETE statements in a transaction.[1]

**Listing 1: Transaction $X_1$**

```
SELECT C_ID FROM Customer WHERE C_ID_STR = '50665'
SELECT * FROM Customer WHERE C_ID = 50665
SELECT * FROM Airport, Country WHERE AP_ID = 180 AND
    AP_CO_ID = CO_ID
SELECT * FROM Frequent_Flyer WHERE FF_C_ID = 50665
UPDATE Frequent_Flyer SET FF_IATTR00 = -14751,
    FF_IATTR01 = 8902 WHERE FF_C_ID = 50665 AND
    FF_AL_ID = 1075
UPDATE Customer SET C_IATTR00 = -14751, C_IATTR01 =
    89025 WHERE C_ID = 50665
COMMIT
```

Consider the transaction $X_1$—taken from the SEATS workload of [2]—shown in Listing 1.

- $FV_S(X_1) = [2, 1, 1, 1]$ because SELECT statements in $X_1$ refer to *Customer* table twice, and the other three tables once each.
- $FV_U(X_1) = [1, 1]$ because UPDATE statements in $X_1$ refer to two tables once each.
- $FV_I(X_1) = FV_D(X_1) = [\ ]$ because there are no UPDATE or DELETE statements in $X_1$.
- $FV(X_1) = [2, 1, 1, 1] + [\ ] + [1, 1] + [\ ] = [2, 1, 1, 1, 1, 1]$

Imagine a second transaction $X_2$ similar $X_1$ that references *Customer* only once. $FV(X_2)$ will be $[1, 1, 1, 1, 1, 1]$.

## 2.2 Angular cosine distance

Angular cosine distance, henceforth *ACD*, measures the distance between two transactions—in particular, between their feature vectors. For two $n$-dimensional vectors $\mathbf{A}$ and $\mathbf{B}$, each with indices $0, 1, \cdots n-1$, and with non-negative values, *ACD* is defined as follows [21].

$$ACD(\mathbf{A}, \mathbf{B}) = \frac{2}{\pi}\left(\cos^{-1}\left(\frac{\sum_{i=0}^{n-1} \mathbf{A}_i\mathbf{B}_i}{\sqrt{\sum_{i=0}^{n-1} \mathbf{A}_i^2}\sqrt{\sum_{i=0}^{n-1} \mathbf{B}_i^2}}\right)\right) \quad (1)$$

*ACD* is a distance measure or *metric*, and furthermore is unitary or normalized: $0.0 \leq ACD(\mathbf{A}, \mathbf{B}) \leq 1.0$. Because *ACD* distances are unitary, a closeness threshold *Eps* (for example, 0.2) can be defined so that two transactions at most *Eps* apart are considered 'close'; otherwise, they are declared 'far'. Normalized distance functions enable easy similarity definition: $similarity = 1.0 - distance$. 'Close' transactions have 0.8 (or 80%) similarity—something even a non-expert can understand.

For $X_1$ and $X_2$ of Section 2.1, $ACD(X_1, X2) = ACD([2, 1, 1, 1, 1.1], [1, 1, 1, 1, 1, 1]) = 0.197$, or they are $1.0 - 0.197 = 0.803$ (80.3%) similar which seems intuitively correct.

[1] INSERT, UPDATE, and DELETE statements can also have embedded SELECT queries, and those are handled similarly to the way $FV_S$ is.

Let $FV(X_3) = [1, 1, 1, 1, 1, 0]$. $X_3$ does not update the *Customer* table, and hence the 0. $X_3$ is somewhat dissimilar to $X_1$ and $X_2$, and indeed, $ACD(X_1, X_3) = 0.295$, or they are only 70.5% similar which again seems intuitive. With the closeness threshold *Eps* set to 0.2, $X_1$ and $X_2$ will be considered close, and may end up in the same cluster, whereas $X_1$ and $X_3$ will not belong to the same cluster.

## 2.3 DBSCAN parameter tuning

An open-source DBSCAN implementation [16]—instrumented to use *ACD*—performs transaction clustering. (By default, it uses the Euclidean distance function.) The author did not consider other clustering algorithms because DBSCAN has been found adequate for transaction clustering previously [11, 23].

DBSCAN's two tunable parameters *Eps* and *minPts* define *density*. A hyper-sphere of radius *Eps* with at least *minPts* points inside is considered *dense*. *ACD* does not eliminate hand-tuning DBSCAN parameters, but provides twofold help.

- *Eps* = 0.2 means that *independent of workload*, two transactions have to be at least 80% similar before they can belong to the same cluster. With such unnormalized distance functions as the Euclidean, *Eps* values are workload dependent.
- For many workloads, *ACD*-based DBSCAN clustering is not very sensitive to the two parameter values, and a good starting point is (*Eps, minPts*) = (0.2, 10). (Section 5.2.)

# 3 MySQL EXTENSIONS FOR TRANSACTION FEATURE EXTRACTION

Feature extraction uses the data from three in-memory tables shown in Fig. 1 that contain transaction-level, statement-level, and table-level information. The three tables will henceforth be referred to by the acronyms *e_t_h*, *e_s_h*, and *e_s_t*.

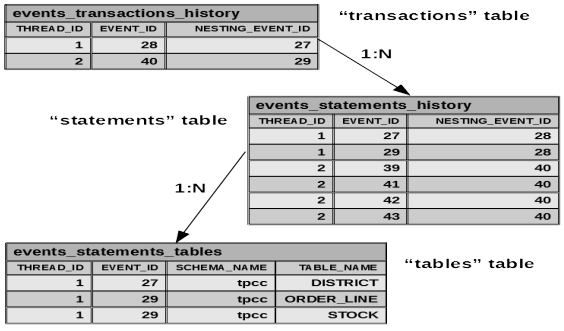


**Figure 1: Relationships between the three tables.**

Each completed transaction appears as a row in *e_t_h*, and the statements within are captured in *e_s_h* (1:N relationship). The newly added *e_s_t* table contains zero or more rows for each statement present in *e_s_h*—one for each distinct table reference in that statement (another 1:N relationship). Such statements as COMMIT and ROLLBACK do not refer to any tables, and therefore, have no presence in *e_s_t*. The columns in Fig. 1 only capture inter-table relationships. The other columns added to the three tables—not explicitly shown in Fig. 1—capture transaction-, statement-, and table-level statistics.

Using a SELECT query involving multi-way joins among *e_t_h*, *e_s_h*, and *e_s_t* tables, such information as transaction text; statement type; statement text; statement run-time; transaction

run-time; table names appearing in statement and their counts; number of rows examined; locking times; and so on is easily extracted.

Tables similar to the ones depicted in Fig. 1 already preexist (or can be easily created) in SQL Server [10], Oracle [12], and PostgreSQL [13], and therefore, server instrumentation of the kind described in this section is possible in those products.

## 4 SYSTEM ARCHITECTURE

A prototype SQL transaction clustering system has been implemented as depicted in Fig. 2. The Linux KVM virtual machine represents a hosted environment running a customer application (OLTP-Bench and Sysbench workloads during experimentation). The top MySQL instance within the Linux KVM has been instrumented as described in Section 3 to enable transaction-level data collection and feature extraction. The Windows 10 machine contains data processing components, including those performing transaction clustering and classification. (A second hosted application would require its own data processing node.)

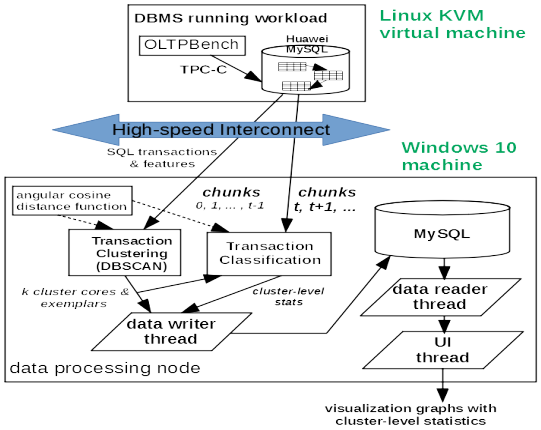


**Figure 2: Architecture of a transaction clustering system.**

A SQL query runs every 5 seconds on the Linux machine, and performs data collection. (The interval ensures minimal data collection overhead, but is a system parameter.) Each 5-second chunk includes feature and non-feature data: SQL statements; statement types; table names; table counts; statement durations; transaction duration; lock times; and rows examined by statements. An epoch-style Linux timestamp is associated with each chunk, and the resulting time-series is shipped to the data processing node where chunk-based iterators process it.

The first $t$ chunks are used to perform transaction clustering. During experimentation, the first 30 seconds worth of transactions ($t = 6$) were found adequate to determine transaction clusters, but $t$ is a system parameter. OLTP workloads do not have ad-hoc queries, and so clusters, once formed, should not change. If that assumption is violated, DBSeer's online implementation of DBSCAN can be used [5], but we leave that for future work.

Once clusters form, the rest of the streaming transactions are simply classified. The average distance (computed using *ACD*) from a transaction $X$ to a cluster's exemplars is calculated, and $X$ is assigned to the cluster for which that average distance is minimum, as long as that minimum value is no more than a threshold (say 0.2). If the threshold is exceeded, $X$ is declared an outlier.

Simple roll-up operations compute cluster-level statistics from transaction statistics as long as transactions are not outliers. If necessary, data about outliers can be captured and processed similarly.

## 5 CLUSTERING EXPERIMENTS

These experiments demonstrate that *ACD*-based DBSCAN is effective at finding transaction clusters, and is not very sensitive to *Eps* and *minPts* parameter values. Workload consists of OLTP-Bench [2] and Sysbench [6]. Out of OLTP-Bench's 15 workloads, the author was able to run 11.[2] Clustering effectiveness requires expected cluster counts which OLTP-Bench already provides, and were manually determined for Sysbench. In Tables 1 and 2, expected cluster counts are indicated in brackets after the workload names. Actual cluster counts are not always integral because each value is an average of 5 runs, and DBSCAN sometimes produces slightly different cluster counts for different samples.

### 5.1 *ACD*-based DBSCAN

*ACD*-based DBSCAN with (*Eps*, *minPts*) = (0.2, 10)—henceforth, $ACD(0.2, 10)$—is an excellent starting point for clustering database transactions. Table 1 captures $ACD(0.2, 10)$'s performance against a baseline provided by the well-known Euclidean distance function. In the Euclidean baseline, *Eps* and *minPts* values are set using a heuristic provided by the DBSCAN authors in a subsequent paper [14]. We will term the resulting baseline *SEKX* after the author names. The heuristic itself works as follows. Let the *dimensionality* of a workload—defined to be maximum feature vector length—be *DIM*. Then:

- Heuristic value of $Eps = (2 * DIM) - 1$
- Heuristic value of $minPts = (2 * DIM)$

$ACD(0.2, 10)$ handily outperforms *SEKX* in 8 out of 13 workloads, and equally important, is never worse than *SEKX* in the remaining 5 workloads. For 9 workloads, *SEKX* puts all of the transactions into single clusters, and hence is ineffective.

The following observations—cross-referenced in the 'Comments' column of Table 1—provide reasons why $ACD(0.2, 10)$'s cluster counts are slightly off in a few cases.

(1) Epinions cluster count is off by 1 because a transaction type selects from *Review* and *Trust* tables separately, and another type selects from their joined version. Both end up in the same cluster because the feature vector construction in Section 2.1 currently does not distinguish them.
(2) YCSB cluster count is off by 1 because two of the transaction types have point and range selects, but are otherwise identical, and that difference is currently not captured as a feature.
(3) AuctionMark and SEATS produced the correct cluster counts with $ACD(0.15, 10)$ and $ACD(0.15, 15)$, respectively.

Observations 1 and 2 suggest possible features that can be extracted, and added to the feature vector.

### 5.2 Sensitivity analysis of *ACD*

When used along with *ACD*, DBSCAN is not very sensitive to various *Eps* and *minPts* values. Table 2 captures *ACD* results for 6 sets of parameters. Approximately, cluster membership criterion becomes more stringent as one reads across a row, and hence cluster counts can be expected to diminish from left to right. As can be seen, *ACD* is not very sensitive to parameter values for most of

[2]Wikipedia turns out to be hard to cluster because two of the transaction types have frequencies of only 0.07% each, and are rarely present in samples. One can ignore 'Wikipedia' results, but they are included for completeness.

**Table 1: *ACD* (0.2, 10) versus *SEKX* ($2 * DIM - 1$, $2 * DIM$)**

| Workload & expected cluster count | DIM | Avg. cluster count | | Comments |
|---|---|---|---|---|
| | | *ACD* | *SEKX* | |
| AuctionMark (9) | 9 | 9.4 | 1 | Observation 3 |
| Epinions (9) | 2 | 8 | 1 | Observation 1 |
| SEATS (6) | 8 | 7 | 1 | Observation 3 |
| SIBench (2) | 1 | 2 | 2 | |
| SmallBank (6) | 5 | 6 | 1 | |
| sysbench_ro (10) | 1 | 10 | 10 | |
| sysbench_rw (10) | 5 | 10 | 10 | |
| TATP (7) | 3 | 7 | 1 | |
| TPC-C (5) | 10 | 5 | 1 | |
| Twitter (5) | 2 | 5 | 1 | |
| Voter (1) | 4 | 1 | 1 | |
| Wikipedia (5) | 7 | 2 | 1 | Footnote 2 |
| YCSB (6) | 2 | 5 | 1 | Observation 2 |

the workloads. Even when it is (AuctionMark and sysbench_rw; and to a lesser extent SEATS and TPC-C), graceful degradation is observed. Therefore, it is not crucial for a DBA to get the values of *Eps* and *minPts* spot on, and any reasonable values should perform respectably well. 'Sysbench_rw' is tricky to cluster for its highly symmetrical transactions—all of the transactions are roughly equidistant from all of the other transactions—but two *ACD* configurations perform well.

**Table 2: *ACD* with different *Eps* and *minPts* values.**

| Workload & expected cluster count | Avg. cluster count | | | | | |
|---|---|---|---|---|---|---|
| | *ACD* (.20,10) | *ACD* (.15,10) | *ACD* (.15,15) | *ACD* (.10,10) | *ACD* (.10,15) | *ACD* (.05,20) |
| AuctionMark (9) | 9.4 | 9.4 | 5.4 | 8.2 | 5.4 | 4 |
| Epinions (9) | 8 | 8 | 8 | 8 | 8 | 8 |
| SEATS (6) | 7 | 7.4 | 6 | 8.4 | 7 | 5.2 |
| SIBench (2) | 2 | 2 | 2 | 2 | 2 | 2 |
| SmallBank (6) | 6 | 6 | 6 | 6 | 6 | 6 |
| sysbench_ro (10) | 10 | 10 | 10 | 10 | 10 | 10 |
| sysbench_rw (10) | 10 | 9 | 0.6 | 0 | 0 | 0 |
| TATP (7) | 7 | 7 | 7 | 7 | 7 | 7 |
| TPC-C (5) | 5 | 6 | 5.8 | 6 | 5.8 | 4.8 |
| Twitter (5) | 5 | 5 | 5 | 5 | 5 | 5 |
| Voter (1) | 1 | 1 | 1 | 1 | 1 | 1 |
| Wikipedia (5) | 2 | 2 | 2 | 2 | 2 | 2 |
| YCSB (6) | 5 | 5 | 5 | 5 | 5 | 5 |

# 6 PERFORMANCE TROUBLESHOOTING USING TRANSACTION CLUSTERING

Experiments in this section demonstrate how cluster-level performance monitoring simplify debugging of several real-life problems faced by cloud DBA's.

## 6.1 Cluster signatures

OLTP-Bench's SmallBank workload simulates some operations of a bank, and produces the cluster signatures shown in Fig. 3 with 10-terminal workload, and scale-factor of 1. The three sub-plots capture the average values of the three transaction-cluster-level metrics: lock time (in ms); number of rows examined; and TPS

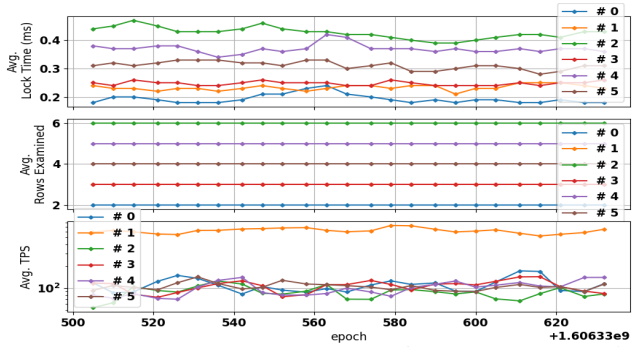


**Figure 3: SmallBank's cluster signatures (10-terminal workload).**

(transactions/sec). Each sub-plot contains six lines—one for each of the transaction clusters identified.[3]

From SmallBank cluster signatures, a cloud DBA can learn several things about the workload. First, six clusters means that there are six types of transactions in SmallBank—as OLTP-Bench confirms. Second, cluster signatures are discernible. Third, small signature variations exist because each data point aggregates 5 seconds worth of transactions (which themselves execute in a multi-tasking environment). Fourth, all of the transactions in a given cluster examine the same number of rows—a SmallBank peculiarity. Fifth, cluster exemplars reveal that the only cluster with read-only transactions is cluster 1—explaining its its highest TPS values. Sixth, when 100 terminals generate workload, the same six clusters form (graphs not included to save space), thereby confirming that clusters are load independent—a practically useful promise of clustering.

## 6.2 Identification of transaction rollbacks

Transaction rollbacks are normal DBMS occurrences, but sometimes their frequencies become problematic. If rollbacks are limited to a transaction type, cluster-level monitoring helps because unexpected additional clusters form.

The TPC-C benchmark [20] has five well-known transaction types. Rollbacks are demonstrated using the *Payment* transaction by modifying its code such that after submitting 2 out of its 7 statements, it rolls back with 20% probability—simulating a problematic high-frequency rollback. To make example even more realistic, a second rollback—simulating a normal and rare DBMS occurrence—happens after the sixth statement with 0.1% probability (overall probability $0.8 \times 0.1 = 0.08\%$). Because the problematic rollbacks are numerous, they should form their own cluster, whereas the normal rollbacks should not.

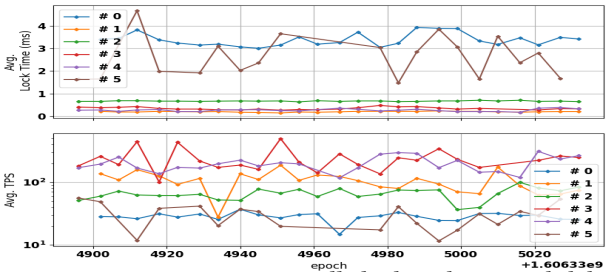


**Figure 4: *Payment* transaction rolls back with 20% probability causing an unexpected sixth cluster (Cluster 1) to appear.**

[3]The second subplot contains only five lines because clusters 1 and 3 examine 3 rows each, and the plotting software cannot distinguish two overlapping lines.

The modified TPC-C benchmark produces the results shown in Fig. 4. Usually, TPC-C produces five clusters, but six are found. A sample exemplar from Cluster 1 reveals the 'incomplete' *Payment* transaction with a telltale `rollback` issued after two statements.

```
UPDATE WAREHOUSE SET W_YTD = W_YTD + 1704.68994140625
    WHERE W_ID = 2
SELECT W_STREET_1, W_STREET_2, W_CITY, W_STATE, W_ZIP,
    W_NAME FROM WAREHOUSE WHERE W_ID = 2
rollback
```

The normal (rare) DBMS rollbacks (0.08% probability) do not form a cluster because they do not meet DBSCAN's density requirement mentioned in Section 2.3. High-frequency rollbacks are difficult to identify if cluster-level statistics are not kept. Applications often resubmit transactions in case of rollbacks, and users only notice and wonder about degraded performance. An incident report would cause DBA's or programmers to dig through voluminous logs to even begin suspecting a culprit.

## 6.3 Performance drift

Performance drift refers to a situation in which one (or just a few) cluster's performance drifts from its norm. Many situations can cause performance drifts. Here is a typical one: A DBA might forget to reinstate an index that (s)he deliberately dropped during a bulk load operation.

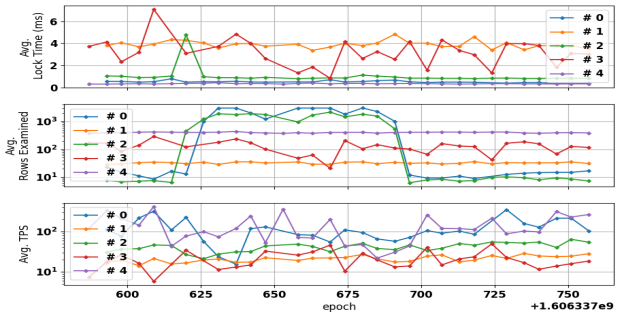


**Figure 5: An index made invisible during [615, 685].**

To simulate a performance drift, a secondary index (used by two out of the five TPC-C transactions) on the *CUSTOMER* table is made invisible[4] to the query optimizer during a portion of the run. When the index is made unavailable, the query optimizer has to use table scans instead of index seeks. As can be seen in Fig. 5, average row counts show dramatic increases (drifts) for clusters 0 and 2 during the interval [615, 685].[5]

When the performance of only one cluster drifts, objects related to only that cluster (e.g., tables, indexes, statistics) are good starting points for debugging. Without cluster-level statistics, such a diagnosis may require considerably more work.

## 6.4 System-wide performance problem

System-wide performance problems are caused by such things as a failed network card, operating system reboot, failed disk, and runaway process hogging CPU's. If all of the clusters experience simultaneous degraded performances, a system-wide issue may be the cause. One such situation is created using a CPU-hogging program that spawns as many processes as the number of CPU cores on the computer (8), and then making them run infinite 'while' loops—thereby creating a CPU bottleneck.

[4]MySQL command used was: `ALTER TABLE CUSTOMER ALTER INDEX IDX_CUSTOMER_NAME INVISIBLE;`

[5]As an aside, if an invisible index makes no difference to any cluster's performance, it might be safe to drop—the reason why that feature was added to MySQL.

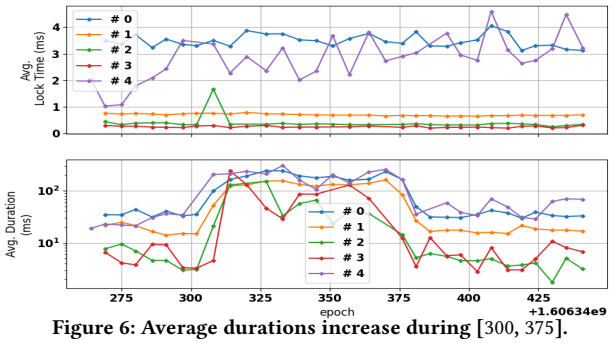


**Figure 6: Average durations increase during [300, 375].**

In the resulting graphs shown in Fig. 6, during the interval [300, 375], average durations of all of the clusters show unmistakable jumps. After about 375, when the offending program is killed, all five average durations return to their baseline values. Interestingly, average lock times are largely unaffected, indicating that for the few transactions that did manage to execute during CPU saturation, lock time did not take a hit. Such observations should provide the DBA a good starting point to formulate a hypothesis before beginning a detailed investigation.

## 6.5 Bottleneck analysis

Cloud applications run on pre-provisioned VM's. Because application behaviour is relatively unknown, a bottleneck may develop—in CPU, memory, disk I/O, network I/O, and so on. Furthermore, bottlenecks may vary by transaction types. Non-cloud DBA's are used to monitoring such operating system-level performance counters as *vmstat*, *iostat*, and *netstat* in Linux for bottleneck identification, but cluster-level statistics offer a complementary method that can provide additional help.

To study whether a VM has sufficient memory, its memory is reduced on the fly from 16 GB to 3 GB while TPC-C workload runs. Such a drastic change in memory allocation is only for demonstration: typical changes should be much smaller.

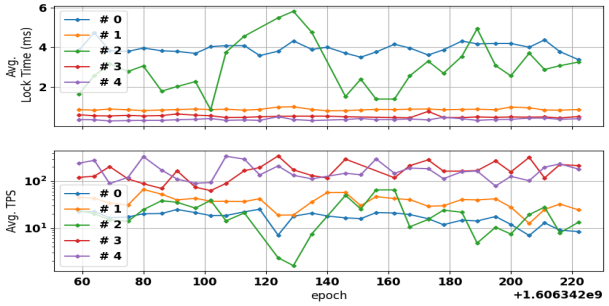


**Figure 7: Memory reduces from 16 GB to 3 GB at timestamp 120.**

In the results captured in Fig. 7, memory reduction happens at timestamp 120 onward. The average TPS values before and after that interval show no discernible changes. There is a noticeable drop at 120 as the operating system seems to adjust to the new memory setting, but soon, normal service resumes. The 'Avg. lock time' metric is also mostly unaffected, and therefore, one can conclude that this VM is well-provisioned for memory.

In the next variation, CPU is constrained. Changing CPU count in KVM requires a machine restart, and therefore, an approach similar to the one in Section 6.4 is taken, except that 7 out of the

8 cores are kept busy running infinite 'while' loops. The results appear in Fig. 8.

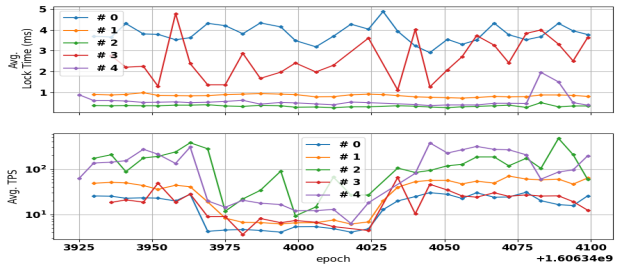


**Figure 8: CPU saturation during the interval [3960, 4025].**

The CPU bottleneck spans the interval [3960, 4025] during which reduced 'Avg. TPS' values are visible. The highest 'Avg. TPS' values are for clusters 2 and 4 (read-only transactions *Stock-Level* and *Order-Status*, respectively). Outside of the CPU bottleneck, those values are somewhat close, but during the bottleneck, *Order-Status* transaction's performance takes a bigger hit (Y-axis is log-scale) suggesting *Order-Status* is much more sensitive to CPU than *Stock-level* is in this environment. If *Order-Status* is deemed important (say because customers check statuses their orders often), it may make sense to over-provision for CPU rather than for memory if a choice is to be made between the two.

## 7 RELATED WORK

Identifying transaction clusters from SQL text arriving at a database server is important for two reasons. First, applications deployed in cloud environments are often web-applications [17] using object-relational mappings to submit SQL queries, and do not use stored procedures. Second, as noted by Stonebraker *et al.* [19], because of SQL's 'one language fits all' approach, transaction code may use a mix of stored procedures, prepared statements, and Java/C++/C# code.

Clustering itself is a broad and well-studied topic [22]. SQL query clustering and classification has been studied under two granularities: query-level and transaction-level. SQL query features previously tried include terms in `SELECT`, `JOIN`, `FROM`, `GROUP BY`, and `ORDER BY` clauses, table names, column names, normalized estimated execution costs [7, 9]; and features have been converted into vectors, graphs, or sets [7, 18]. As a general observation, fewer features suffice in self-similar OLTP workloads; ad-hoc workloads require more features. Such distance functions as cosine, Jaccard, and Hamming have been tried for clustering SQL queries [1, 9, 18], although only the Euclidean has been tried at the transaction level before [5]. Before this research, feature extraction has mined SQL text from DBMS logs [9, 18], or MaxScale proxy server [11]; server-side feature extraction is novel.

## 8 CONCLUSIONS AND FUTURE WORK

This research makes a case that in addition to user, table, index, and schema level monitoring provided, DBMS's should start to provide transaction-cluster-level monitoring. In applications deployed in the cloud, and for OLTP workloads, that additional level simplifies debugging of performance problems: unexpected transaction rollbacks, performance drifts, bottleneck identifications, and so on. Angular cosine distance-based DBSCAN is an improvement over Euclidean-based DBSCAN with *SEKX* heuristic [14] (better clusters and simplified DBSCAN parameter tuning).

Future work may investigate the following features for transaction clustering: column names to possibly identify index issues; predicate types (point queries vs. range queries) and counts; access paths used; isolation level; number of sorts; join tables; and so on. Multiple cluster-level signatures may help because an application may have distinct 'peak' and 'off-peak' behaviours. Whether *ACD* is suitable with such clustering algorithms as BIRCH [24] and $k$-means [8] remains to be seen. Server-side feature extraction can be attempted in other modern database systems using minor extensions to preexisting scaffoldings.

## ACKNOWLEDGMENTS

Matthew Van Dijk implemented the server-side extensions needed for transaction feature extraction. Anonymous reviewers provided valuable feedback.

## REFERENCES

[1] Rakesh Agrawal, Ralf Rantzau, and Evimaria Terzi. 2006. Context-sensitive ranking. In *Proceedings of the SIGMOD 2006 Conference*. 383–394.
[2] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288.
[3] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*. 226–231.
[4] Junhao Gan and Yufei Tao. 2015. DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation. In *Proceedings of the SIGMOD 2015 Conference*. 519–530.
[5] GitHub. 2020. *DBSeer*. Retrieved September 17, 2020 from https://github.com/barzan/dbseer
[6] GitHub. 2020. *sysbench*. Retrieved August 5, 2020 from https://github.com/akopytov/sysbench
[7] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* 12, 12 (2019), 2118–2130.
[8] Stuart P. Lloyd. 1982. Least squares quantization in PCM. *IEEE Trans. Inf. Theory* 28, 2 (1982), 129–136.
[9] Vitor Hirota Makiyama, Jordan Raddick, and Rafael D. C. Santos. 2015. Text Mining Applied to SQL Queries: A Case Study for the SDSS SkyServer. In *SIMBig*.
[10] Microsoft. 2019. *System Dynamic Managemenet Views*. Retrieved August 4, 2020 from https://docs.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/system-dynamic-management-views?view=sql-server-ver15
[11] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. 2013. Performance and resource modeling in highly-concurrent OLTP workloads. In *Proceedings of the SIGMOD 2013 Conference*. 301–312.
[12] Oracle. 2020. *About Dynamic Performance Views*. Retrieved September 22, 2020 from https://docs.oracle.com/cd/B19306_01/server.102/b14237/dynviews_1001.htm#i1398692
[13] PostgreSQL. 2020. *The Statistics Collector*. Retrieved August 4, 2020 from https://www.postgresql.org/docs/9.6/monitoring-stats.html
[14] Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. 1998. Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and Its Applications. *Data Min. Knowl. Discov.* 2, 2 (1998), 169–194.
[15] Erich Schubert, Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Trans. Database Syst.* 42, 3 (2017), 19:1–19:21.
[16] Scikit Learn. 2019. *DBSCAN*. Retrieved August 2, 2020 from https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html
[17] Alexandre Verbitski *et al.* 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the SIGMOD 2017 Conference*. ACM, 1041–1052.
[18] Gökhan Kul *et al.* 2018. Similarity Metrics for SQL Query Clustering. *IEEE Trans. Knowl. Data Eng.* 30, 12 (2018), 2408–2420.
[19] Michael Stonebraker *et al.* 2007. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the VLDB 2007 Conference*. 1150–1160.
[20] Transaction Processing Performance Council 1992. *TPC-C*. Retrieved September 22, 2020 from http://www.tpc.org/tpcc/
[21] Wikipedia. 2019. *Cosine similarity*. Retrieved August 11, 2020 from https://en.wikipedia.org/wiki/Cosine_similarity
[22] Dongkuan Xu and Yingjie Tian. 2015. A Comprehensive Survey of Clustering Algorithms. *Annals of Data Science* 2 (2015), 165–193.
[23] Dong Young Yoon, Ning Niu, and Barzan Mozafari. 2016. DBSherlock: A Performance Diagnostic Tool for Transactional Databases. In *Proceedings of the SIGMOD 2016 Conference*. 1599–1614.
[24] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. 1997. BIRCH: A New Data Clustering Algorithm and Its Applications. *Data Min. Knowl. Discov.* 1, 2 (1997), 141–182.