Knowledge and Information Systems (2023) 65:2863–2894
https://doi.org/10.1007/s10115-023-01850-5

REGULAR PAPER

Towards intelligent database systems using clusters of SQL transactions

Arunprasad P. Marathe[1]

Received: 7 July 2022 / Revised: 13 January 2023 / Accepted: 13 February 2023 /
Published online: 16 March 2023
© The Author(s), under exclusive licence to Springer-Verlag London Ltd., part of Springer Nature 2023

Abstract
Transactions are the bread-and-butter of database management system (DBMS) industry. When you check your bank balance, pay bill, or move money from saving to chequing account, transactions are involved. That transactions are self-similar—whether you pay a utility company or credit card, it is still a 'pay bill' transaction—has been noted before. Somewhat surprisingly, that property remains largely unexploited, barring some notable exceptions. The research reported in this paper begins to build 'intelligence' into database systems by offering built-in transaction classification and clustering. The utility of such an approach is demonstrated by showing how it simplifies DBMS monitoring and troubleshooting. The well-known DBSCAN algorithm clusters online transaction processing (OLTP) transactions: this paper's contribution is in demonstrating a robust server-side feature extraction approach, rather than the previously suggested and error-prone log-mining approach. It is shown how 'DBSCAN + angular cosine distance function' finds better clusters than the previously tried combinations, and simplifies DBSCAN parameter tuning—a known nontrivial task. DBMS troubleshooting efficacy is demonstrated by identifying the root causes of several real-life performance problems: problematic transaction rollbacks; performance drifts; system-wide issues; CPU and memory bottlenecks; and so on. It is also shown that the cluster count remains unchanged irrespective of system load—a desirable but often overlooked property. The transaction clustering solution has been implemented inside the popular MySQL DBMS, although most modern relational database systems can benefit from the ideas described herein.

Keywords Intelligent database management systems · Transactions · Feature extraction · Similarity measures · Unsupervised clustering · Database monitoring · Troubleshooting · DBSCAN

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) database application. He

✉ Arunprasad P. Marathe
arun.marathe@huawei.com; ap.marathe@gmail.com

[1] Huawei Technologies Canada Co., Ltd., 19 Allstate Pkwy, Markham, ON L3R 5A4, Canada

searches several flights, 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 second transaction may have fewer `INSERT` statements because only one person is flying. The two transactions have both similarities and differences, but similarities probably outnumber differences. The self-similarity is due to their accessing the same tables in similar fashions, although the SQL statements generated may not be identical. Taking a step further, there may be a reason to believe that their performances are similar—a hypothesis that can be exploited if found true. In particular, if both of them can be put into the same cluster along with other similar transactions, and cluster performance is monitored as a whole, a simple and practically useful OLTP system abstraction may result.

A study of the various applications provided by the two popular OLTP benchmarking toolkits OLTP-Bench [1] and Sysbench [2] reveals that each application contains SQL transactions that can be neatly divided into a small number of non-overlapping *transaction clusters*. For example, the SEATS application of OLTP-Bench implements an airline reservation system and contains the following six types of activities from which six clusters may be identified:

1. Find flights ordered by departure time between two cities possibly considering nearby airports.
2. Make a new reservation.
3. Delete a reservation.
4. Find open seats on a flight.
5. Update a reservation by changing a seat.
6. Update such customer information as frequent flyer number.

An actual commercial airline reservation system will have more capabilities resulting in more clusters, but still, cluster count is likely to be small (say less than 25).[1]

Within a cluster, transactions differ in the parameter values of the SQL statements, 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. Cluster-level metrics are simple rolled-up versions of transaction-level metrics: for example, average values of transactions/sec (TPS); number of rows (read, updated, or sent to the client); locking time; and so on. In turn, transaction-level metrics are rolled-up versions of statement-level metrics for the statements within a transaction. In turn, some of the statement-level metrics—for example, rows read/written and estimated execution cost—can be computed from table-level or execution plan operator-level metrics. The existence of such a natural hierarchy may permit problem troubleshooting to proceed from coarser to finer levels.

1.1 Benefits of cluster-level monitoring

Cluster-level monitoring of an OLTP application has three advantages over transaction- or statement-level monitoring. First, rather than monitoring thousands of SQL queries or transactions, it is simpler and likely more effective to observe performances of only six clusters (in the case of SEATS). Second, the cluster count is independent of an OLTP application's load

[1] Indeed, all of the workloads in OLTP-Bench have between 1 and 10 clusters [1]. Read-write and read-only flavors of Sysbench [2] contain 10 clusters each. Although Sysbench is a synthetic benchmark, all of the benchmarks in OLTP-Bench are based on real-life applications: online auctions; banking; talent show voting; social networking; key-value store; airline reservation system; and so on.

(which is proportional to the number of active terminals). Third, the benefits of clustering multiply when an OLTP application is deployed in cloud where a few database administrators (DBA) have to monitor performance of multiple applications simultaneously [3]. A cloud DBA does not have the luxury of developing an in-depth understanding of an application and its performance profile, and needs a helping hand.

SQL transaction clustering helps a (cloud) DBA by simplifying troubleshooting of several performance problems: identification of problematic transaction rollbacks; performance bottlenecks; system-wide issues; performance drifts; and so on. The benefits also extend to non-cloud applications, and therefore, this research makes a case for database systems to become knowledge-based by providing transaction cluster-level monitoring in addition to the existing table-, index-, schema-, and user-level monitoring. The new granularity is not meant to replace the existing tools, but nicely complements them: 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.

1.2 Novelties and key contributions of this paper

This research reported herein contains three novelties.

1. For SQL transaction clustering, features are extracted server-side by modifying the MySQL code, rather than the existing state-of-the-art that relies on log-mining (which is cumbersome and error-prone).
2. A novel application of SQL transaction clustering is demonstrated: namely, DBMS system monitoring and performance troubleshooting.
3. DBSCAN parameter tuning—a known difficult task [4, 5]—is simplified. In cloud-deployed DBMS applications—increasingly common in the last 5 years—a cloud DBA may not be able to hand-tune DBSCAN parameters of the multiple applications under her supervision.

The following three subsections elaborate.

1.2.1 Server-side transaction feature extraction

SQL transaction are clustered based on inter-transaction distances. Selected attributes from a transaction form its feature vector by a process called feature extraction, and inter-transaction distances are calculated among such 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.

Server-side extensions offer four benefits over previous methods. First, regular-expression-based SQL log mining is error-prone because identifying SQL constructs requires a parser, in general. Second, commercial database systems do not provide API accesses to their parsers, and parsing code needs to be duplicated. Third, log mining requires identification of transaction boundaries, which may be difficult because transaction statements may be intermingled in the log. (In server-side extensions, the server keeps track of the entire transactions.) Fourth, if logs are mined from such a proxy server as MaxScale [6], ‘autocommit’ must be enabled which essentially breaks a transaction into single-statement pieces.

Server-side feature extraction is implemented by extending the preexisting MySQL server data structures meant for server monitoring. Similar data structures preexist in most modern DBMS engines (Oracle, SQL Server, PostgreSQL, and so on), and hence the solution suggested herein has wider applicability.

1.2.2 Clustering helps DBMS performance troubleshooting

Modern DBMS systems are some of the most complex software systems built. Monitoring their performance and troubleshooting problems has been difficult, and shows no signs of becoming simpler. The workload transactions in online transaction processing (OLTP) systems can be naturally grouped into non-overlapping clusters, and DBMS performance can be monitored at cluster-level. As this paper demonstrates, by doing that, several real-life performance problems can be troubleshot easily.

The proposed approach goes hand-in-glove with prior work on boosting OLTP system performance that executes a group of similar transactions in lock-steps so that the first transaction instance paves the L1-Instruction cache, and subsequent ones enjoy nearly miss-free executions [7]. The notion of similarity in [7] is statement- and access-path-level (commit, rollback, seek of a particular index, scan of a particular index, and so on), but in principle, can be even more effective at transaction level—for example, TPC-C's *OrderStatus* transaction. After all, two *OrderStatus* instances—especially if they are read-only, as is the case with *OrderStatus*—are likely to execute nearly identical instructions.

In the last two years, utility of database transactions in debugging has generated renewed interest. The author first noted the usefulness of SQL transaction clusters in performance troubleshooting in [8], but even the transactions by themselves (dropping 'SQL' and 'clustering' altogether) were found useful in [9] in the following sense. A client-side application can store and access its shared state on the server-side using transactions, thereby minimizing hard-to-debug concurrency bugs in distributed applications. For such bugs that do remain, correlating shared states persisted in database with debug logs using data provenance may help in debugging. The author hopes that SQL transactions continue to elicit such unusual applications.

1.2.3 DBSCAN parameter tuning

The author chose DBSCAN clustering algorithm [10] for three practical reasons. First, DBSCAN is well known in database research, and has even been used to cluster SQL transactions previously [11]. Second, an open-source Python implementation is available [12] that plugs in nicely with the rest of the Python-based transaction clustering system the author has built. Third, DBSCAN's utility claims have been questioned [4] and refuted by the DBSCAN authors themselves [5]. An independent evaluation (this research) confirms DBSCAN's suitability for clustering SQL transactions.

DBSCAN parameters need to be hand-tuned, and different applications require different values. In a subsequent paper, some of the DBSCAN authors proposed a heuristic—termed *SEKX* in this paper—for parameter tuning. The current work demonstrates that a heuristic enabled by the angular cosine distance function (*ACD*) outperforms *SEKX* by finding more accurate transaction clusters, and simplifying DBSCAN parameter tuning. Last, but not least, extensive SQL transaction clustering experiments—at a scale reported in this paper—have not been performed before although some preliminary attempts were made.

1.3 In the larger context

This research proposes a new granularity of data collection in DBMS systems: transaction-cluster-level. The overall message is that the new level is a natural extension of the existing table-, index-, query-, user-, and schema-level granularities, and can be implemented using

straightforward programming extensions to DBMS servers. This work opens up opportunities for new research as explained below. Intentionally, the directions identified are broad-based. Some specific topics are also mentioned when considering the future work in Sect. 8.

First, transaction-cluster-level performance measurement greatly simplifies troubleshooting of certain scenarios. Previous research has used transaction-cluster-level data for performance measurement and resource usage (scalability) predictions. What other uses exist for this new data granularity? Second, should transaction granularity be used more extensively by existing query tuning tools? Third, a recent trend has started to use machine learning in query optimization [13]. Can a transaction-as-a-whole be optimized using machine learning techniques the way it has been exploited to drastically reduce L1 instruction-cache misses [7]? For example, many real-life transactions are predicated based on a small number of parameters. Can deep neural networks be created based on supervised or unsupervised learning of those parameter values? Fourth, can machine learning models be trained using transaction-cluster-level performance signatures so that they can predict deviations from those signatures automatically?

A quote from a recent book on benchmarking cloud-native systems [14, p. 109] is a good way to end this introduction.

> [...] databases typically lack in flexibility when it comes to the provided analysis methods. More advanced methods like classification or clustering are not supported by many systems. Especially in exploratory analysis, such methods may be essential [...]

This research takes some preliminary steps in alleviating the situation.

2 Clusters of SQL transactions

To perform transaction-cluster-level data monitoring, SQL transactions need to be segregated into non-overlapping 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, and is described in the following three subsections.

1. Certain distinguishing attributes (called *features*) are extracted from a transaction, and an n-element *feature vector (FV)* is formed (Sect. 2.1).
2. The distance between two transactions—defined to be the distance between their feature vectors—is computed using a distance function (Sect. 2.2).
3. A clustering algorithm uses the feature vectors and the distance function to determine clusters (Sect. 2.3).

2.1 Feature vector construction

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

- Statement type: `SELECT`, `INSERT`, `UPDATE`, and `DELETE`.
- Table name(s)—possibly none—referenced in the statement in 'schema.table' format.
- Counts associated with table names indicating frequency.

In other applications, different or additional features may need to be extracted, but the overall approach described here remains valid. (Experiments in Sect. 5.8 evaluate whether transaction clustering improves when two dynamic features are added to the mix.)

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.[2]

To handle `ROLLBACK` and `COMMIT` statements, a vector element can be added that is 0 for `ROLLBACK` and 1 for `COMMIT`. If more than two such statements need to be handled—for example `BEGIN`, `END`, `SAVEPOINT`, and so on—they should be numbered 1 onward for the reason mentioned in Sect. 8. In this research, however, no such additional statements are handled.

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 OLTP-Bench [1]—shown in Listing 1. It is the 'update customer information' transaction mentioned in Sect. 1, and its feature vector is computed as follows.

- $FV_S(X_1) = [2, 1, 1, 1]$ 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 `INSERT`'s or `DELETE`'s in X_1.
- The complete feature vector for X_1 is calculated as follows.

$$\begin{aligned} FV(X_1) &= FV_S(X_1) + FV_I(X_1) + FV_U(X_1) + FV_D(X_1) \\ &= [2, 1, 1, 1] + [\,] + [1, 1] + [\,] \\ &= [2, 1, 1, 1, 1, 1] \end{aligned}$$

Consider a second transaction X_2 similar to X_1 that selects from *Customer* only once. $FV(X_2)$ will be $[1, 1, 1, 1, 1, 1]$. For all of the tables other than *Customer*, *Airport*, *Country*, and *Frequent_Flyer*, the corresponding slots in the two feature vectors are 0's and are not explicitly shown.

The slot positions in feature vectors matter when comparing two transactions. For example, imagine a third transaction X_3 similar to X_1 that does not update the *Customer* table, and selects from that table only once. Because 'update-*Customer*' occupies the last slot in the feature vector, $FV(X_3)$ will be $[1, 1, 1, 1, 1, 0]$.

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

2.2 Angular cosine distance

Angular cosine distance, henceforth *ACD*, measures the distance between two transactions using their feature vectors. For two n-dimensional vectors $\mathbf{A}$ and $\mathbf{B}$, each with indices $0, 1, \ldots n-1$, and with nonnegative values, *ACD* is defined as follows [15].

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

ACD is a distance measure or *metric* as shown in [16], and therefore, has the following properties.

1. $ACD(\mathbf{A}, \mathbf{B}) \geq 0$
2. $ACD(\mathbf{A}, \mathbf{B}) = ACD(\mathbf{B}, \mathbf{A})$
3. $ACD(\mathbf{A}, \mathbf{B}) \leq ACD(\mathbf{A}, \mathbf{C}) + ACD(\mathbf{C}, \mathbf{B})$ (triangle inequality)

Triangle inequality is an important property because it says that the closest distance between two objects is a direct route: a detour through a third object cannot be any closer. For SQL transactions, it means that two transactions that are close to a third transaction are also close to each other. If that statement does not hold, SQL transaction clusters are not meaningful.[3]

ACD also has a fourth property that is particularly useful in SQL transaction clustering: *ACD* distances are normalized.

4. $0.0 \leq ACD(\mathbf{A}, \mathbf{B}) \leq 1.0$

Because *ACD* distances are unitary, a closeness threshold $Eps \in [0.0, 1.0]$ (for example, 0.2) can be defined so that two transactions at most *Eps* apart are considered 'close'; otherwise, they are declared 'far.'[4] *ACD*—and normalized distance functions in general—also enable an easy similarity definition: $similarity = 1.0 - distance$

Thus 'close' transactions with $Eps = 0.2$ are at least 0.8 (or 80%) similar—something even a non-expert can understand. Similarity connection will play an important role in simplifying DBSCAN parameter tuning.

For the X_1 and X_2 transactions of Sect. 2.1:

$$\begin{aligned} ACD(X_1, X_2) &= ACD([2, 1, 1, 1, 1.1], [1, 1, 1, 1, 1, 1]) \\ &= 0.197 \end{aligned}$$

X_1 and X_2 are $1.0 - 0.197 = 0.803$ (80.3%) similar which seems intuitively correct. The transaction X_3 from Sect. 2.1 with $FV(X_3) = [1, 1, 1, 1, 1, 0]$ 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.

[3] A close cousin of *ACD*—the dot-product-based cosine distance—is not a metric because it does not satisfy the triangle inequality and indeed performs worse than *ACD* in clustering SQL transactions as demonstrated in Sect. 5.7.

[4] The name *Eps* is chosen because it matches a DBSCAN parameter's name as mentioned in Sect. 2.3.

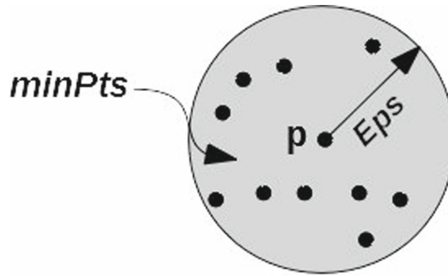


Fig. 1 DBSCAN parameters *Eps* and *minPts*

2.3 DBSCAN and its parameter tuning

DBSCAN [10] is a density-based clustering algorithm whose two parameter values *Eps* and *minPts* define density, and are illustrated in Fig. 1. If the circle (in general, hyper-sphere) centered at p with radius *Eps* has at least $minPts - 1$ other points inside it, p belongs to a dense neighborhood, and is considered a *core* point of a cluster. All of the core points belong to clusters, and if a non-core point has a core point as a neighbor within distance at most *Eps*, that non-core point also belongs to the core point's cluster.

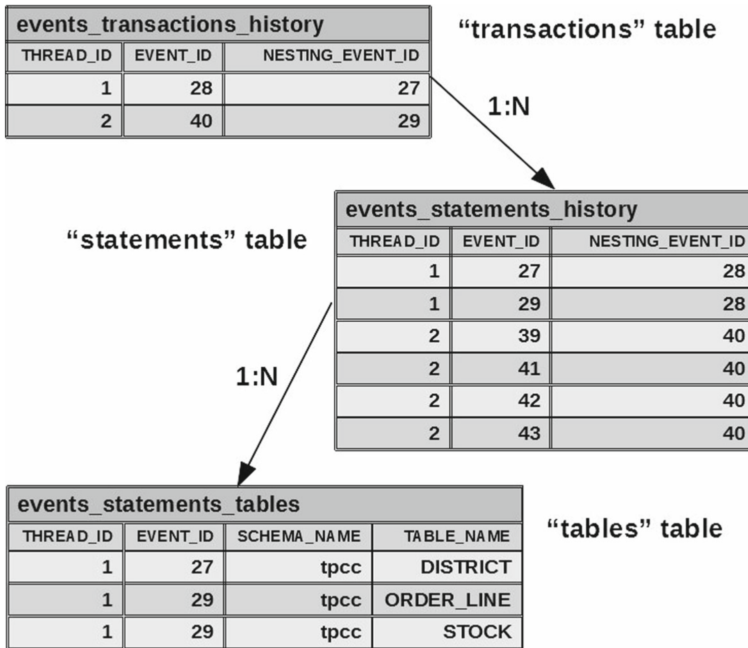
DBSCAN parameter tuning is difficult [4, 5] for two reasons.

1. *Eps* values are workload-dependent because distances themselves are. The distance value 20.0 can be 'close' for one workload, but 'very far' for another.
2. *minPts* is somewhat workload-independent in that some minimum cluster density may be required no matter what, but *minPts* too requires tuning in some cases.

Eps normalization using min–max values may help, but those values need to be determined, and determination could be error-prone because of sampling—and in cloud environments, cumbersome. Less obviously, it was found that such unnormalized distance functions as the Euclidean require dissimilar objects (transactions) to be spread apart artificially for correct cluster formations—as done in the *DBSeer* system [17] and explained in Sect. 5.5. That spread-apart factor is workload dependent, and requires its own tuning.

ACD does not eliminate hand-tuning DBSCAN parameters, but simplifies it by providing twofold help.

- *Eps* value has an intuitive meaning. For example, *Eps* = 0.2 means that *independent of workload*, transactions have to be at least 80% similar before they can belong to the same cluster. With such unnormalized distance functions as Euclidean, *Eps* values do not have that property. The same can be said for *Eps* value variations. For example, in the case of *ACD*, lowering *Eps* from 0.2 to 0.1 means that transactions now have to be at least 90% similar, but in the case of Euclidean, lowering *Eps* from 20 to 15 makes cluster membership criterion more demanding than before, but to what degree? That degree is workload-dependent.
- As demonstrated in Sect. 5.6, 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).



events_transactions_history

THREAD_ID	EVENT_ID	NESTING_EVENT_ID
1	28	27
2	40	29

events_statements_history

THREAD_ID	EVENT_ID	NESTING_EVENT_ID
1	27	28
1	29	28
2	39	40
2	41	40
2	42	40
2	43	40

events_statements_tables

THREAD_ID	EVENT_ID	SCHEMA_NAME	TABLE_NAME
1	27	tpcc	DISTRICT
1	29	tpcc	ORDER_LINE
1	29	tpcc	STOCK

Fig. 2 Relationships between the three tables

3 MySQL extensions for feature extraction from transactions

Extraction of transaction features such as the ones mentioned in Sect. 2.1 currently uses SQL log mining and takes one of two approaches: logs are obtained from such a proxy server as MaxScale [6] (as done in the *DBSeer* system [11, 18]), or directly from a DBMS (as done in [19, 20]). This research proposes a third alternative: server-side feature extraction by instrumenting a DBMS.

Server-side feature extraction, as implemented in MySQL, uses data from the three in-memory tables shown in Fig. 2 that contain transaction-level, statement-level, and table-level information. Only the *events_statements_tables* table is new; the other two preexist. New columns were added to all of the tables, however. The three tables will henceforth be referred to by the acronyms *e_t_h*, *e_s_h*, and *e_s_t*. They belong to a larger class of such tables—in a special schema named ‘performance_schema’—that capture information about running workloads. This real-time performance data capture feature, also called *performance schema*, is enabled by default in MySQL [21].

Each completed transaction appears as a row in the *e_t_h* table whose key is (THREAD_ID, EVENT_ID). One or more statements belonging to a transaction are captured in *e_s_h*, and accordingly there is a 1:N relationship between the two tables. In Fig. 2, the transaction with EVENT_ID 40 in *e_t_h* contains four statements appearing as four of the rows in *e_s_h* using NESTING_EVENT_ID as the linkage column. 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. Accordingly, there is a 1:N relationship between the bottom two tables. Such statements as `COMMIT` and `ROLLBACK` do not refer to any tables,

and therefore, have no presence in *e_s_t*. For example, the two-way join

"`SELECT`..`FROM` ORDER_LINE, STOCK `WHERE`.."

of TPC-C appears as two of the rows in *e_s_t* with EVENT_ID of 29. The columns in Fig. 2 only capture the inter-table relationships. The other columns added to the three tables—not explicitly shown in Fig. 2—capture transaction-, statement-, and table-level statistics.

A statistic's scope suggests a natural place for its persistence in Fig. 2. Such statement-level statistics as rows sent to client, durations, and lock wait times are best persisted as columns of the "statements" table *e_s_h* (or its equivalent in other DBMS's). Such table-level statistics as a table's frequency in a statement, rows read, and rows updated is best kept in such tables as *e_s_t*. Whether to keep fine-grained or rolled up data (or both) can be a design decision. If rolled-up data is not kept, SQL's `GROUP BY` and other similar clauses can compute various aggregations on the-fly. Keeping only coarse-level data will minimize performance impact.

Using a `SELECT` query involving multi-way joins among *e_t_h*, *e_s_h*, and *e_s_t* tables, such information as the 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. The query itself can be nontrivial to write, but needs to be written only once. One tricky aspect of such data capture is that the 1:N relationships shown in Fig. 2 are not enforced by MySQL for performance reasons, and therefore, dangling tuples may result. Such tuples should either be identified and filtered out by the query itself (as is done in this research), or should be deleted in a post-processing step.

The statistical data gathered in this research are only for illustration, and depending on need, a variety of other statistics can be extracted: point-selection versus range-selection; determination of '`SELECT`... `FOR UPDATE`' (a SQL language feature found in MySQL and Oracle); query complexity based on subquery count or join-width; the number of statements not using a good index; estimated execution costs; estimated row counts; and so on.

Although the server extensions described are MySQL-specific, the idea has wider applicability. Tables similar to those depicted in Fig. 2 preexist (or can be easily created) in other products too: in SQL Server, Oracle, and PostgreSQL, they are called dynamic management views [22], dynamic performance tables [23], and statistics collectors [24], respectively. 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. 3. The Linux KVM virtual machine represents a hosted environment running a customer application. The Windows 10 machine contains data processing components, including those performing transaction clustering and classification. A high-speed network separates the two machines, and because of that, no data processing overhead is put on the cloud VM. DBMS workload is generated using the OLTP-Bench [1] (or Sysbench [2]) toolkit running locally on the top machine.

The top MySQL instance has been instrumented as described in Sect. 3 to enable transaction-level data collection and feature extraction. The bottom data processing node then rolls up such data to derive transaction cluster-level metrics. A SQL query runs on the top MySQL instance every 5 s, and performs data collection. (The interval ensures minimal data collection overhead, but is a system parameter.) From each 5-s chunk, the following fea-

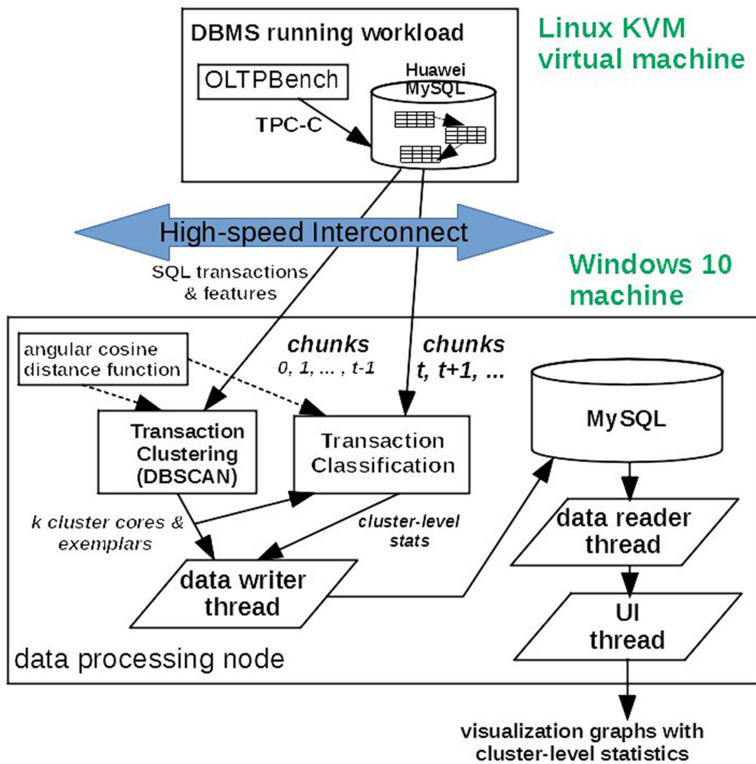


Fig. 3 Architecture of a transaction clustering system

ture and non-feature information is extracted: SQL statements; statement types (SELECT, INSERT, COMMIT, and so on); table names; table counts; statement durations; transaction duration; lock times; and rows examined by statements.

Each 5-s chunk contains data about all of the transactions that finished *within* the last 5 s, and an epoch-style Linux timestamp is associated with that data resulting in a streaming time-series that is shipped to the data processing node where chunk-based iterators process it in pipelined fashion. After each data collection event, the three tables shown in Fig. 2 are truncated to avoid rereading the same data during the next interval.[5]

In the rest of this section, the two data processing components of Fig. 3 are described: transaction clustering and transaction classification.

4.1 Transaction clustering

Chunk processing begins by consuming the first t chunks to perform transaction clustering. During experimentation, the first 30 s worth of transactions ($t = 6$) were found adequate to determine transaction clusters, but t is a system parameter. The DBSCAN algorithm [10]—enhanced with the *ACD* distance function—performs transaction clustering [12].

After the initial t chunks have been used to determine transaction clusters, the subsequent chunks are processed by the 'Transaction Classification' module. OLTP workloads do not

[5] The tables are implemented using circular in-memory queues, and therefore, do not grow beyond limits, but truncation ensures data fidelity.

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 [17], but we leave that for future work.

4.2 Transaction classification

Transaction classification algorithm needs k cluster cores and exemplars therein to classify a transaction X as follows. Distances from X to all of the exemplars of a cluster are computed using the angular cosine distance function, and averaged. For the k cluster cores, k average distances are computed, and X is assigned to the cluster corresponding to the minimum average distance, as long as that distance is not more than a threshold value (for example, 0.2). If the threshold is exceeded, X is not close enough to any of the clusters and is declared an outlier.

5 SQL transaction clustering experiments

DBMS performance monitoring at transaction-cluster-level is a central theme of this paper, and accurate cluster formations are required. Previous attempts at SQL transaction clustering have been small-scale [11, 18], and cluster quality has not been given much attention. This research attempts to remedy that situation.

The experimentation contained in this section provides the answers to a series of questions.

1. How much overhead does server-side feature extraction add? (Sect. 5.3.)
2. How effective is the *ACD*-based DBSCAN algorithm in forming accurate transaction clusters? This is answered by comparing *ACD* with a baseline heuristic suggested by the DBSCAN authors themselves. (Sect. 5.4.)
3. The *DBSeer* system attempted SQL transaction clustering using a Euclidean-based distance function. Does *ACD* find better clusters than the Euclidean-based distance function? (Sect. 5.5.)
4. Sensitivity analysis of *ACD*-based DBSCAN shows that it is not crucial for a DBA to get the values of *Eps* and *minPts* just right. A good starting point of (*Eps*, *minPts*) = (0.2, 10) forms acceptable clusters for many workloads. (Sect. 5.6.)
5. Why is it necessary to use the angular cosine distance function? What is the drawback of the well-known dot-product-based cosine distance function? (Sect. 5.7.)
6. Do better clusters form when more transaction features are extracted? (Sect. 5.8.)

5.1 Experimental setup

The system depicted in Fig. 3 is used in all of the experiments, but is not optimized for performance because this research's focus is on performance monitoring, troubleshooting, and debugging. The top Linux machine is a KVM virtual machine with 8 CPU cores (2.5 GHz), and 16 GB memory. The Windows 10 data processing node runs Enterprise Edition of Windows, and contains an *i7-8700* 3.20 GHz CPU with 16 GB memory. In all of the experiments, both of the machines were lightly loaded.

5.2 Workload

The workload consists of the applications in the OLTP-Bench [1] and Sysbench [2] toolkits. Out of the OLTP-Bench's 15 workloads, the author was able to run 11.[6] Two flavors of Sysbench benchmark were run: read-only and read-write, indicated in the results using 'sysbench_ro' and 'sysbench_rw,' respectively. Thus, all of the experiments ran on 13 workloads.

An open-source DBSCAN implementation from 'Scikit-learn' [12] performs transaction clustering. It was instrumented to use *ACD* and the other distance functions tried. (By default, it uses the Euclidean distance function.) The author did not consider other clustering algorithms because although DBSCAN has been used for transaction clustering previously [11, 18], its suitability has not been established because the previous clustering experiments were small-scale, and did not use a variety of workloads.

5.2.1 Expected cluster counts

Determination of clustering effectiveness requires expected cluster counts to which actual counts can be compared. Expected cluster counts were determined as follows.

- OLTP-Bench already provides the expected cluster counts for the workloads therein [1].
- For Sysbench, the counts were determined manually by studying the transactions themselves.

In Tables 1, 2, 3, and 4, expected cluster counts are indicated in brackets after the workload names. For example, 9 clusters are expected for the AuctionMark workload. Actual cluster counts in some of the experiments are not always integral because each value is an average of 5 runs produced using different samples, and DBSCAN sometimes produces slightly different cluster counts for different samples. When evaluating various distance functions, fairness is ensured by reusing the same transaction samples.

5.3 Performance impact of server-side feature extraction

In the first experiment, an attempt is made to measure the performance impact of the instrumentation described in Sect. 3. MySQL's performance schema is enabled by default and contributes one portion of the overhead. The other portion is due to the server extensions shown in Fig. 2. Ideally, the two overheads would be separately measurable, but the server-extension overhead was so small that we could not measure it. The combined overhead was 3–7% based on the average requests/s values of the TPC-C benchmark queries locally submitted to Huawei MySQL using OLTP-Bench, and thus transaction clustering can be an 'always-on' feature in many situations. Overhead can be reduced further by modifying MySQL code so that only selected portions of 'performance schema' are enabled, but we leave that for future work.

5.4 Clustering effectiveness of the *ACD*-based DBSCAN

ACD-based DBSCAN with (*Eps*, *minPts*) = (0.2, 10)—henceforth, abbreviated to *ACD* (0.2, 10)—is an excellent starting point for clustering database transactions as the experiment in this section demonstrates.

[6] CH-benCHmark, JPAB, and ResourceStreser workloads produced run-time errors. LinkBench was excluded because it (and JPAB) have been removed from a rewrite of OLTP-Bench currently under development [25].

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

Workload and 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 4
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)[a]	7	2	1	
YCSB (6)	2	5	1	Observation 2

[a] Wikipedia application turns out to be hard to cluster by all of the distance functions for a non-obvious reason. The five types of transactions in Wikipedia have very skewed frequencies, with two types of transactions contributing 98.94% of the total. Two other types of transactions occur with frequencies of only 0.07% each, and are rarely present in the clustering samples. Changing transaction frequencies caused OLTP-Bench errors (reported to the benchmark authors) and, therefore, were left at their original values. The reader can ignore the Wikipedia results, but they are included for completeness

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 [26]. We will term the resulting baseline *SEKX* after the author names.

5.4.1 The *SEKX* heuristic

The *SEKX* heuristic works as follows. Let the *dimensionality* of a workload—defined to be the maximum feature vector length—be *DIM*. Then:

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

The *DIM* column of Table 1 captures a workload's dimensionality, defined as above. Accordingly, for example, when AuctionMark is clustered using *SEKX*-based DBSCAN, the parameter settings are: $Eps = 17$ and $minPts = 18$.

5.4.2 Performance of *ACD*(0.2, 10) versus the *SEKX* baseline

The 'Avg. cluster count' columns of Table 1 compare clustering effectiveness of the ACD versus the *SEKX* baseline. As mentioned in Sect. 5.2, average cluster counts in this and subsequent tables are averages over 5 runs, each using its own transaction samples, and hence are not always integral. As can be seen, $ACD(0.2, 10)$ 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 the reasons why $ACD(0.2, 10)$'s cluster counts are slightly off in a few cases.

Observation 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 Sect. 2.1 currently does not distinguish them.

Observation 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.

Observation 3. AuctionMark produced the correct cluster count with $ACD(0.15, 10)$.

Observation 4. SEATS produced the correct cluster count with $ACD(0.15, 15)$.

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

The author has also compared *ACD* with baselines provided by other distance functions, including some *ACD* variations.

- Jaccard distance by treating a SQL query as a bag of words
- A combination of Jaccard and Levenshtein distances
- An *ACD* variation in which the distance is defined to be: $\text{avg}(FV_S, FV_I, FV_U, FV_D)$ by taking the average distance of the four sub-vectors FV_S, FV_I, FV_U, and FV_D mentioned in Sect. 2.1.
- Another *ACD* variation in which the distance is defined to be: $\max(FV_S, FV_I, FV_U, FV_D)$.

ACD outperformed them all.

5.5 *ACD*-based DBSCAN versus *DBSeer*-based DBSCAN

The *DBSeer* system had previously used DBSCAN to cluster database transactions while using an enhanced-Euclidean-based distance function for inter-transaction distances [17]. How that function's clustering effectiveness compares with *ACD*'s is investigated in this experiment.

In *DBSeer*, similar transactions are those that access the same tables in similar fashions. Dissimilar transactions are those in which there is at least one table that one transaction accesses (via `SELECT`, `INSERT`, `UPDATE`, or `DELETE`), and the other one does not.

- Similar transactions use the usual Euclidean distance function. For example, for transactions X_1 and X_2 of Sect. 2.1, the Euclidean distance computation yields: $DBSeer([2, 1, 1, 1, 1], [1, 1, 1, 1, 1]) = 1.0$.
- For dissimilar transactions such as X_1 and X_3 of Sect. 2.1, a modified formula is used that multiplies 'dissimilar' squares by a spread-apart factor (set to 10000.0)—viz. $10000 \cdot (1-0)^2$—and yields:

$$\begin{aligned} DBSeer(X_1, X_3) &= DBSeer([2, 1, 1, 1, 1], [1, 1, 1, 1, 0]) \\ &= 100.0 \end{aligned}$$

DBSeer's enhanced-Euclidean distance function requires hand-tuning of DBSCAN parameters, but to enable comparison with *ACD*, a pair of parameter values that gave an overall decent performance across all of the workloads was determined: (*Eps*, *minPts*) =

Table 2 *ACD* versus *DBSeer*'s enhanced-Euclidean distance function

Workload and expected cluster count	Avg. cluster count			
	ACD (0.2, 10)	*DBSeer* (10, 10)	*DBSeer* (250, 10)	*DBSeer* (250, 20)
AuctionMark (9)	9.4	9.6	3	3
Epinions (9)	8	8	1	1
SEATS (6)	7	8.2	1	1
SIBench (2)	2	2	1	1
SmallBank (6)	6	6	1	1
sysbench_ro (10)	10	10	10	10
sysbench_rw (10)	10	0	10	10
TATP (7)	7	7	1	1
TPC-C (5)	5	5	3	3
Twitter (5)	5	5	1	1
Voter (1)	1	1	1	1
Wikipedia (5)	2	2	2	2
YCSB (6)	5	5	1	1

(10, 10). The results are captured in columns 2 and 3 of Table 2. Once again, the abbreviation *DBSeer*(10, 10) means '*DBSeer*-based DBSCAN algorithm with (*Eps*, *minPts*) = (10, 10).'

Ignoring the 'sysbench_rw' workload, *DBSeer* is somewhat worse than *ACD*, but is decent. *DBSeer* could not cluster 'sysbench_rw,' however, and thus *ACD* wins the comparison.

'Sysbench_rw' is an interesting case. It is hard to cluster not only by *DBSeer* but also by several other distance functions that the author has tried. The reason has to do with that workload's highly symmetrical transactions: all of the transactions are roughly equidistant from all of the other transactions. Unsurprisingly, such objects are hard to cluster.

For scientific curiosity, we ran a parameter sweep for *DBSeer*'s enhanced-Euclidean function so that *DBSeer*-based DBSCAN could cluster 'sysbench_rw.' *Eps* ranged from 25 to 250 in increments of 25, and *minPts* varied between 10 and 20. In all, 20 pairs were examined, and out of those, three candidate pairs emerged: (225, 10), (250, 10), and (250, 20) with average cluster counts of 10.8, 10, and 10, respectively, for 'sysbench_rw.' The last two pairs were chosen to run the entire suit of benchmarks, and the results appear in columns 4 and 5 of Table 2.

As can be seen, both of those pairs can cluster the two Sysbench workloads well, but cannot cluster most of the other workloads in that all of the transactions are put in single clusters. Tuning the DBSCAN parameters using *DBSeer*'s enhanced-Euclidean distance function is indeed difficult. Another approach is to try to tune the spread-apart factor (10000.0), but that just trades tuning of one parameter for another.

5.6 Sensitivity analysis of the *ACD*-based DBSCAN

This experiment attempts to answer the following question: When using the *ACD*-based DBSCAN algorithm, how crucial is it for a DBA to get the values of *Eps* and *minPts* just right? For example, one DBA might want transactions to be put in the same cluster only if they are at least 90% similar. Another DBA might want denser clusters by requiring that

Table 3 Sensitivity analysis of *ACD* with different *Eps* and *minPts* values

Workload and 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

a transaction's neighborhood contain at least 20 similar transactions. It turns out that the *ACD*-based DBSCAN is not very sensitive to various the *Eps* and *minPts* values for most of the workloads.

Table 3 captures *ACD*-based DBSCAN's performance for 6 sets of parameters. Approximately, cluster membership criterion becomes more demanding as one reads across a row, and hence cluster counts can be expected to decrease from left to right. For example, the last column requires two transactions to be at least 95% similar, and their neighborhoods to contain at least 20 similar transactions—extremely challenging conditions. As can be seen, *ACD*-based DBSCAN is not very sensitive to parameter values for most of 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 decent values should perform respectably well. For the tricky 'sysbench_rw' workload, two *ACD* configurations perform well.

The average cluster counts in Table 3 should also be compared with '*DBSeer*' cluster counts in Table 2 (columns 3, 4, and 5) to conclude that *ACD*-based DBSCAN is considerably less sensitive to parameter values than *DBSeer*-based DBSCAN is.

5.7 Why not use the dot-product-based cosine distance?

Some readers may wonder why the angular cosine distance function is used, and not its well-known cousin: the cosine distance based on dot product. The experiment in this section provides an answer.

The cosine of two nonzero vectors **A** and **B** can be computed using the dot-product formula [15]:

$$\mathbf{A} \cdot \mathbf{B} = \|\mathbf{A}\| \|\mathbf{B}\| \ \cos(\theta)$$

The dot-product-based cosine similarity is $\cos(\theta)$, and the distance is $(1 - \cos(\theta))$. Therefore, the dot-product-based cosine distance—henceforth *DCD*—can be calculated as follows:

$$DCD(\mathbf{A}, \mathbf{B}) = 1 - \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = 1 - \left(\frac{\sum_{i=0}^{n-1} \mathbf{A}_i \mathbf{B}_i}{\sqrt{\sum_{i=0}^{n-1} \mathbf{A}_i^2} \sqrt{\sum_{i=0}^{n-1} \mathbf{B}_i^2}} \right) \tag{2}$$

Although there are similarities between Eqs. 2 and 1, it is somewhat surprising that unlike the angular cosine distance, *DCD* is *not* a metric. In particular, the triangle inequality does not hold for the triplet [1, 0], [0, 1], and [0.7071, 0.7071]:

$$DCD([1,0],[0.7071,0.7071]) + DCD([0.7071,0.7071],[0,1]) < DCD([1,0],[0,1])$$

In SQL transaction clustering, is it important for a distance function to be a metric? It is, as the experiment in this section demonstrates. When the experiment in Sect. 5.4 is repeated by comparing *ACD*(0.2, 10) and *DCD*(0.2, 10) distance functions, the data in Table 4 results. To understand *DCD*(0.2, 10)'s behavior in more detail, two more columns are added that capture the numbers of unclassified transactions after clustering finishes. Once again, because each cell value captures the average of five runs, the numbers of unclassified transactions are not always integral.

Notice that *DCD*(0.2, 10) is never better than *ACD*(0.2, 10), but is worse for three workloads: AuctionMark, SEATS, and SmallBank. For SEATS, *ACD*(0.2, 10) produces a cluster more than the expected count, whereas *DCD*(0.2, 10) produces one fewer cluster. We declare *DCD*(0.2, 10) to be worse because fewer clusters means that two or more of them have been aggregated, and their performances cannot be studied separately, whereas more clusters means that a given transaction type is divided into two or more fine-grained categories.

The last two columns of Table 4 indicate that *DCD* consistently leaves fewer unclassified transactions. In other words, the problem is not that it cannot classify certain transactions, but that it wrongly classifies them in several cases. SmallBank provides an extreme case: all of the transactions form a single large cluster. (Sect. 6.1 describes the six transaction types in SmallBank.)

In short, metric distance functions should be preferred whenever possible although the problems caused by non-metric distance function may not be apparent for several workloads. Indeed, as noted in Sect. 7.4, researchers have not always been precise or correct in using the distance functions in clustering SQL transactions. Because no previous study has performed a thorough investigation on suitable distance functions for SQL clustering, the relative merits of the various distance functions have not been brought to light—something that the experiments in this research begin to address.[7]

[7] As an interesting aside emphasizing the importance of metric distance functions, the author was attempting to generalize an elegant algorithm due to Lingas [27] that computed the relative neighborhood graphs (RNG). Lingas' algorithm computed the RNG for a set of points using the Euclidean distance; the generalization was to compute the RNG for points with associated weights (disks). Generalizing the Euclidean distance to compute inter-disk distances makes it non-metric. A counterexample was found in [28] that suggested that a straightforward generalization of Lingas' algorithm would be nearly impossible—the status quo after more than 27 years.

Table 4 Angular cosine versus dot-product-based cosine: performance of *ACD*(0.2, 10) versus *DCD*(0.2, 10)

Workload and expected cluster count	Avg. cluster count		# of unclassified transactions	
	ACD	*DCD*	*ACD*	*DCD*
AuctionMark (9)	9.4	7	63.2	27.0
Epinions (9)	8	8	0.0	0.0
SEATS (6)	7	5	33.6	5.4
SIBench (2)	2	2	0.0	0.0
SmallBank (6)	6	1	0.6	0.6
sysbench_ro (10)	10	10	0.0	0.0
sysbench_rw (10)	10	10	0.6	0.6
TATP (7)	7	7	0.0	0.0
TPC-C (5)	5	5	3.2	3.2
Twitter (5)	5	5	0.0	0.0
Voter (1)	1	1	3.6	3.6
Wikipedia (5)	2	2	6.4	1.4
YCSB (6)	5	5	0.0	0.0

5.8 Extraction of more features

The experiment in Sect. 5.4 suggested that Epinions and YCSB workloads would benefit from two static features (joined versus individual tables and point versus range selects), but for diversity, two dynamic features of transactions are considered: (1) the total number of rows examined by the statements within a transaction; and (2) the average lock time for the statements within a transaction. Accordingly, the feature vector is lengthened by 2. Contrary to expectation, these two dynamic features did not improve clustering performance, and therefore, detailed results are not included. Nevertheless, experimental work advised us to make the following three remarks.

Remark 1 Feature values require normalization. A typical value for average lock time is 0.03 ms; for row count, typical values are 8, 20 and 130. As indicated in Sect. 2.1, existing feature values are small integers, and without normalization, row count dominates other features.

Remark 2 Even after normalization, clustering performance is still slightly worse than the vanilla *ACD* variation which shows that extracting more features does not necessarily translate to better cluster formations.

Remark 3 When all three combinations of ‘row count’ and ‘average lock time’ are tried as features, no clear winner emerges, indicating that these features should be considered non-primary or non-essential for the 13 workloads.

6 Performance troubleshooting enabled by SQL transaction clustering

The experiments in this section demonstrate how cluster-level performance monitoring simplifies troubleshooting of many real-life problems faced by DBA’s and system administrators. In particular, the experiments help address the following questions, thereby taking some steps in making the database system more intelligent.

1. What can the cluster signatures tell us? (Sect. 6.1.)
2. How do cluster signatures behave under heavy system loads? (Sect. 6.2.)
3. How can we find the performance sweet spot of a DBMS application? (Sect. 6.3.)
4. How can we identify problematic transaction rollbacks? Such rollbacks can negatively impact an application's performance, and are difficult to troubleshoot using existing techniques. (Sect. 6.4.)
5. How can we identify a performance drift whereby a performance metric departs considerably from its baseline value? (Sect. 6.5.)
6. How can we identify such system-wide performance problems as a failed network card or a failed disk? (Sect. 6.6.)
7. What resource—for example, CPU or memory—is most likely to become a bottleneck in a DBMS application? (Sect. 6.7.)

6.1 Transaction cluster signatures

OLTP-Bench's SmallBank workload simulates some operations of a bank and is chosen to illustrate the concept of cluster signatures. Intentionally, a relatively unknown workload is chosen—although OLTP-Bench contains the well-known TPC-C benchmark—because a DBA in a cloud environment may not understand a customer application to the same extent that a dedicated DBA in non-cloud environment might. SmallBank contains six transaction types briefly described below.

1. *Balance* computes the sum of savings and checking balances for a customer, and is the only read-only transaction in SmallBank.
2. *DepositChecking* increases the checking balance by some amount, and can optionally rollback.
3. *TransactSaving* makes a deposit or withdrawal on the savings account, and can optionally rollback.
4. *Amalgamate* moves all of the funds from one customer to another customer's checking account.
5. *WriteCheck* represents writing a check against an account. If the account does not have the necessary funds, a penalty of 1 for overdrawing is applied.
6. *SendPayment* (OLTP-Bench's addition) sends payment from one checking account to another.

The SmallBank cluster signatures with 10-terminal workload at the scale-factor of 1 are shown in Fig. 4. The three sub-plots in Fig. 4 capture the average values of the three transaction-cluster-level metrics: lock time (in ms); number of rows examined; and TPS (transactions/s). Each sub-plot contains six lines—one for each of the transaction clusters identified.[8] Each plotted point represents many transaction-level values rolled up to a cluster-level value, and conveys summary information from the previous 5 s.

From the SmallBank cluster signatures, a DBA can make several 'intelligent' observations.

- Six clusters mean that there are six types of transactions in SmallBank—as enumerated above.
- Cluster signatures are discernible, and therefore, transaction-cluster-level data aggregation and analysis seems worthwhile. For the top two sub-plots, the signatures are even non-overlapping. In the third sub-plot, 5 signatures are clustered around 100 TPS, and

[8] 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.

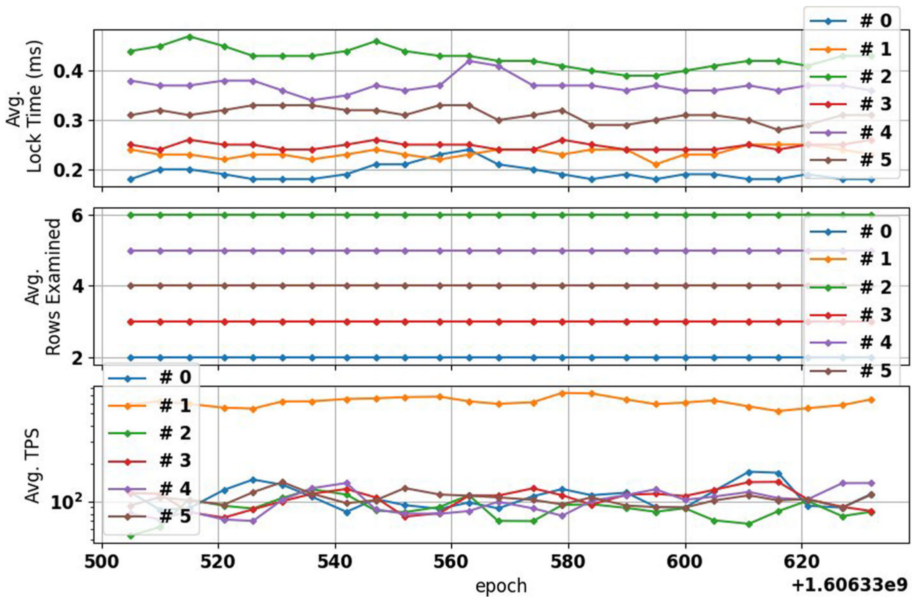


Fig. 4 SmallBank's cluster signatures (10-terminal workload)

unsurprisingly, the highest TPS values (cluster 1: 600–700) are for the read-only *Balance* transaction (and as can be expected, has the second lowest average lock time).

- Most signatures contain small variations for at least three reasons. First, transactions in a cluster are similar but not identical. Second, transactions execute cooperatively in a multitasking environment, and even two identical transactions do not get the same resources. Third, each data point aggregates 5 s worth of data. Nevertheless, the metrics are stable because the workload and its hosting system are stable. (Some of the future experiments will introduce systematic workload and system variations to recreate some real-life scenarios.)
- All of the transactions in a given cluster examine the same number of rows—a SmallBank peculiarity.
- 'Avg. lock time' and 'Avg. rows examined' graphs are stacked in the same order—understandable because usually, the more rows a transaction examines, the more locking time it needs.

The most expensive transaction type—the one with the lowest TPS value—is Cluster 2, representing the *Amalgamate* transaction, whose sample exemplar is produced in Listing 2. The ability to look at cluster exemplars can be useful already. For example, the *Amalgamate* transaction looks suspicious: if funds are moved from one customer's account to another customer's account, should not the second customer's account balance increase, and not decrease (unless, of course, if the bank permits negative balances)?

Listing 2 An exemplar of the *Amalgamate* transaction type

```
SELECT * FROM ACCOUNTS WHERE custid = 586790
SELECT * FROM ACCOUNTS WHERE custid = 728976
SELECT bal FROM SAVINGS WHERE custid = 586790
SELECT bal FROM CHECKING WHERE custid = 728976
UPDATE CHECKING   SET bal = 0.0  WHERE custid = 586790
UPDATE SAVINGS   SET bal = bal - 52840.0  WHERE custid = 728976
commit
```

Cluster signatures can act as baseline performance profiles, and may allow easy detection of any future deviations (possibly even automatically using automatic anomaly detection [29]). The rest of the experiments in this section will introduce systematic variations to either the workload or the hosting system to observe how signatures change. To save space, some graphs will contain fewer than three metrics, and in those cases, the reader can assume that the omitted metric conveys no additional information, or is unaffected by the variation studied.

6.2 Unchanged cluster counts under heavy system load

The system load has a direct impact on a DBMS application's performance. Many metrics—for example, average response time and throughput—degrade when the system load increases. Performance troubleshooting becomes more difficult as the system load increases because the various system logs grow at faster rates, and data sizes increase. One of the promises of keeping cluster-level metrics is that the cluster count remains unchanged irrespective of the system load.

When SmallBank runs with 100 simultaneous terminals rather than 10 as in Sect. 6.1, the cluster count remains 6. (The graphs are omitted to save space.) A system under heavy load will run into performance bottlenecks more easily than the same system under light load, and cluster-level data capture does not 'solve' that problem. Nevertheless, cluster-level data capture can simplify determination of a performance 'sweet spot'—a load value below which the system is likely to run smoothly—as the next experiment suggests.

6.3 Performance sweet spot

By using a variation of the experiment demonstrated in Sect. 6.2, the sweet spot of a DBMS system performance can be determined. In particular, by systematically increasing an OLTP application's load (measured using the number of terminals connected to it), an approximate number can be determined beyond which the performance degradation is no longer graceful because at least some of the cluster-level metrics worsen by large amounts. At that point, throttling mechanisms can kick in to ensure that the load does not increase beyond that sweet spot. The details are omitted for brevity.

6.4 Identification of transaction rollbacks

Transaction rollbacks are normal; some are even expected. For example, SmallBank's *DepositChecking* and *TransactSaving* can occasionally rollback by design (Sect. 6.1). Sometimes, however, rollback frequencies become problematically high, and may require remedial

actions. If rollbacks are limited to a transaction type, cluster-level monitoring helps because unexpected additional clusters form: those for the rolled-back ‘incomplete’ transactions.

TPC-C contains five transaction types [30], which the system correctly identifies as five clusters. (The graphs are omitted for brevity.) Transaction 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 $0.8 \times 0.1 = 0.08\%$). The problematic rollbacks are numerous, and should form their own cluster, whereas the normal ones should not.

The modified *Payment* transaction’s logic is illustrated in Listing 3.

Listing 3 Modified *Payment* transaction with a high-frequency and a normal rollback. The probability calculations are omitted for brevity.

```
UPDATE WAREHOUSE SET W_YTD = W_YTD + 562.6300048828125 WHERE W_ID = 1
SELECT W_STREET_1, W_STREET_2, W_CITY, W_STATE, W_ZIP, W_NAME FROM
    WAREHOUSE WHERE W_ID = 1
<-- high-frequency rollback with 20% probability -->
UPDATE DISTRICT SET D_YTD = D_YTD + 562.6300048828125 WHERE D_W_ID = 1
    AND D_ID = 8
SELECT D_STREET_1, D_STREET_2, D_CITY, D_STATE, D_ZIP, D_NAME FROM
    DISTRICT WHERE D_W_ID = 1 AND D_ID = 8
SELECT C_FIRST, C_MIDDLE, C_LAST, C_STREET_1, C_STREET_2, C_CITY,
    C_STATE, C_ZIP, C_PHONE, C_CREDIT, C_CREDIT_LIM, C_DISCOUNT,
    C_BALANCE, C_YTD_PAYMENT, C_PAYMENT_CNT, C_SINCE FROM CUSTOMER
    WHERE C_W_ID = 1 AND C_D_ID = 8 AND C_ID = 298
UPDATE CUSTOMER SET C_BALANCE = -4496.89013671875, C_YTD_PAYMENT =
    47285.52734375, C_PAYMENT_CNT = 19 WHERE C_W_ID = 1 AND C_D_ID = 8
    AND C_ID = 298
<-- normal rollback with 0.08% probability -->
INSERT INTO HISTORY (H_C_D_ID, H_C_W_ID, H_C_ID, H_D_ID, H_W_ID,
    H_DATE, H_AMOUNT, H_DATA) VALUES (8,1,298,8,1,'2020-02-27
    10:51:37.255',562.6300048828125,'iphxnusrihx')
commit
```

The modified TPC-C benchmark produces the results shown in Fig. 5 in which six clusters form, rather than the expected five. A sample exemplar from Cluster 1 appears in Listing 4, and reveals the ‘incomplete’ *Payment* transaction with a telltale `rollback` issued after two statements.

Listing 4 ‘Incomplete’ *Payment* transaction with a telltale `rollback`

```
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
```

Because the ‘incomplete’ *Payment* transaction contains only two statements, its ‘Avg. Lock Time’ is the least among the six in Fig. 5. The normal (rare) DBMS rollbacks (0.08% probability) do not form a cluster because they do not meet DBSCAN’s density requirement mentioned in Sect. 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.

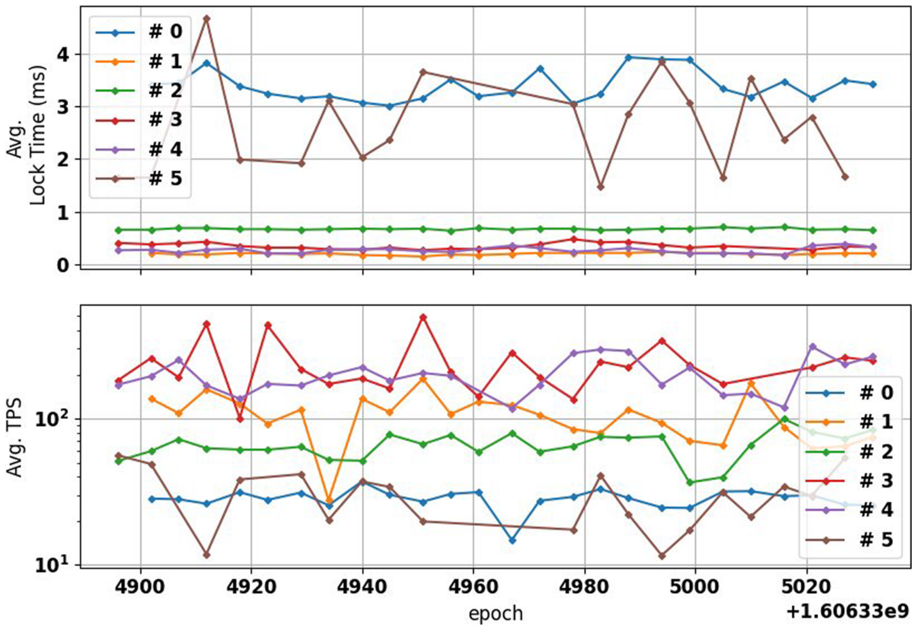


Fig. 5 When *Payment* rolls back with 20% probability, an unexpected sixth cluster (Cluster 1) appears

6.5 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. (Dropping and recreating indexes on either side speed up a bulk load operation.)

To simulate a performance drift, a secondary index on the (C_W_ID, C_D_ID, C_LAST, C_FIRST) columns of TPC-C's CUSTOMER table is dropped while workload is running. MySQL can make an index invisible, making that access method unavailable to the query optimizer. During this experiment, the secondary index is made invisible for a portion of the run.

Two out of the five TPC-C transactions use that index as an access method to navigate to other tables: the read-only *OrderStatus* transaction, and the read-write *Payment* transaction. When the index is made unavailable, the query optimizer has to use table scans instead of index seeks. As can be seen in Fig. 6, average row counts show dramatic increases (drifts) for clusters 0 (*OrderStatus*) and 2 (*Payment*) during the interval [615, 685] during which the index was made invisible. Sure enough, as soon the invisible index is reinstated at the timestamp 685, the optimizer begins using it, and the two 'rows examined' values return to their baseline values soon afterward.

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.

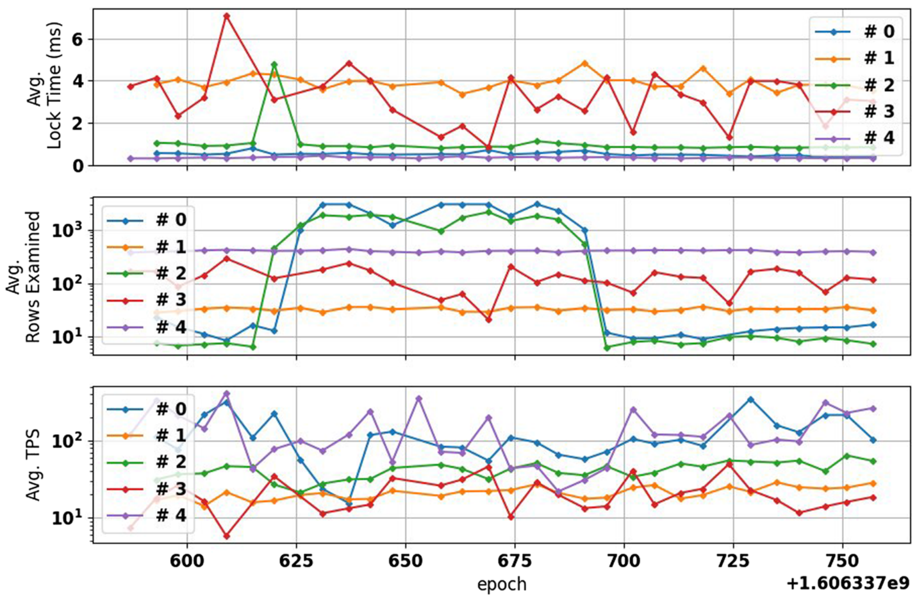


Fig. 6 An index made invisible during [615, 685] causes two transactions to access many more rows

6.6 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.

In the resulting graphs shown in Fig. 7, 'Avg. duration' replaces 'Avg. TPS' as a metric. (The two are inversely related.) CPU saturation happens in the interval [300, 375] during which the 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 DBA's a good starting point to formulate a hypothesis before starting a detailed investigation.

6.7 Bottleneck analysis

Cloud applications run on pre-provisioned VM's such as KVM [31]. Because application behavior 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.

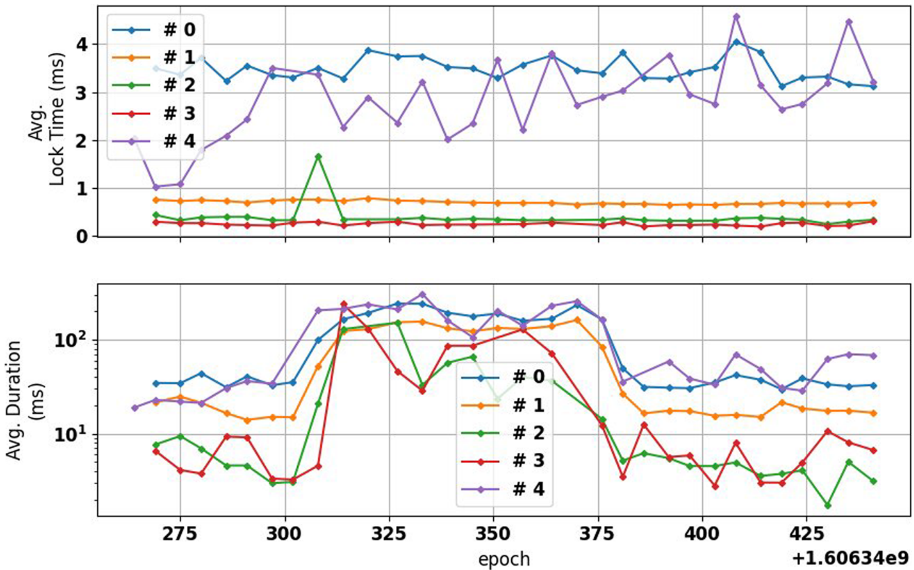


Fig. 7 Average durations increase during [300, 375]

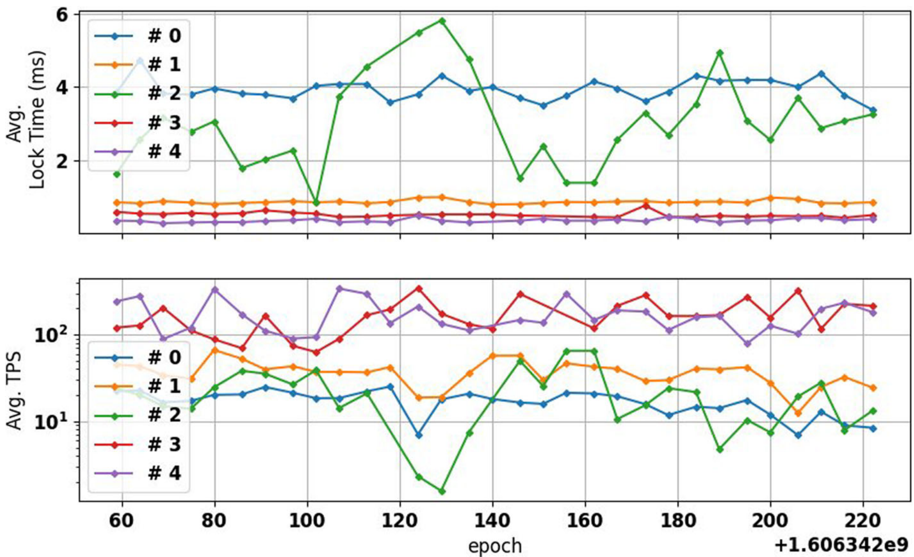


Fig. 8 Memory reduces from 16 to 3 GB at timestamp 120 onward

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.

In the results captured in Fig. 8, 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,

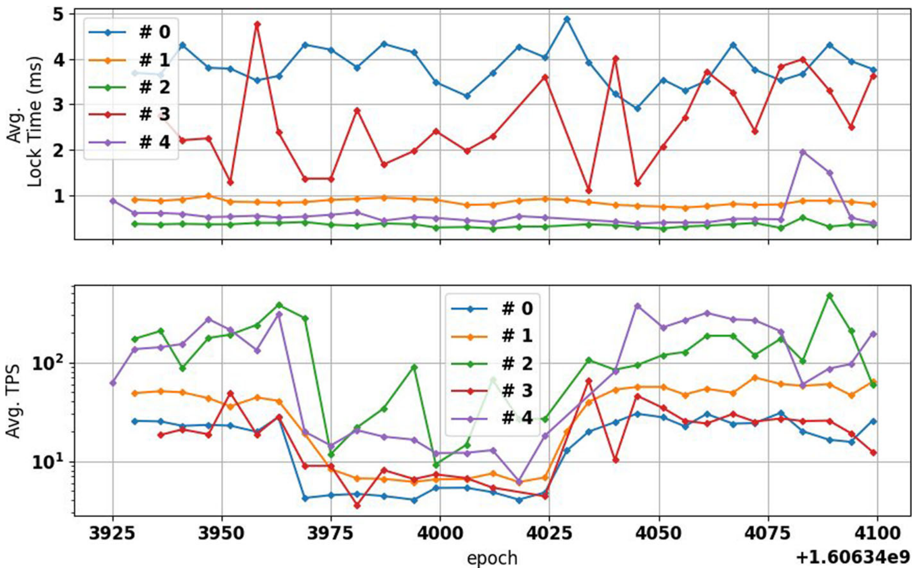


Fig. 9 CPU saturation during the interval [3960, 4025]

but soon, normal service resumes. The 'Avg. lock time' metric is also mostly unaffected except for one cluster, 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 Sect. 6.6 is taken, except that 7 out of the 8 cores are kept busy running infinite 'while' loops. The results appear in Fig. 9.

The CPU bottleneck spans the interval [3960, 4025] during which reduced 'Avg. TPS' values are visible. Highest 'Avg. TPS' values are for clusters 2 and 4 (read-only transactions *StockLevel* and *OrderStatus*, respectively). Outside of the CPU bottleneck, those values are somewhat close, but during the bottleneck, *OrderStatus* transaction's performance takes a bigger hit (Y-axis is log-scale) suggesting that *OrderStatus* is much more sensitive to CPU than *StockLevel* is in this environment. (The observation is consistent across many runs.) If *OrderStatus* is deemed important (say because customers check statuses of 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. The 'what-if' analysis of the type enabled by transaction clustering is somewhat reminiscent of such analysis provided by DBMS workload tuning tools.

7 Related work

The research reported in this paper proposes a new granularity for studying a DBMS system's performance: clusters of similar (but not identical) SQL transactions. Accordingly, such topics as clustering methodologies, distance functions, feature vectors, and so on relevant to this work. For ease of exposition, the related work is divided into many subsections.

7.1 Need for identifying SQL transaction clusters on the server side

Identifying transaction clusters from SQL text arriving at a database server is important for two reasons. First, modern applications deployed in cloud environments are often Web applications [3] in which such frameworks as Ruby-on-Rails [32] and Django [33] use object-relational mappings to submit SQL queries, and do not use stored procedures—which would otherwise form good clusters. Second, as noted by Stonebraker et al. [34], because of SQL's 'one language fits all' approach, transaction code may use a mix of stored procedures; prepared statements; and SQL embeddings in such languages as Java, C++, and C#, necessitating identification of transaction clusters from SQL text arriving at a database server.

7.2 SQL query clustering

Clustering itself is a broad and well-studied topic [35]. Of most relevance to this research are the scenarios when SQL queries are classified or clustered, and in those cases two granularities have been considered: individual query level and transaction level. The former has received a lot more attention than the latter. As noted in [20], SQL query similarity has been the basis for such varied applications as data prefetch in OLAP, SQL-autocomplete, view selection optimization in warehouses, workload analysis, database parameter tuning, and so on. Just as an example, query clusters were found to be a good abstraction when tuning database parameter values using feature vectors conceptually similar to the ones in Sect. 2.1, but including normalized estimated operation costs [36]. Before this research, SQL transaction clusters were considered only once in the context of performance and resource modeling of DBMS applications [18]. Extensive clustering experiments using a variety of OLTP workloads have not been performed previously, although some small-scale attempts were made [11, 18].

7.3 SQL query feature extraction

No matter whether SQL queries or transactions are clustered, SQL query features drive cluster formation. 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 [19, 36], and features have been converted into vectors, graphs, or sets [20, 36]. A variation in [20] suggests SQL query normalization using well-known rewrite rules before feature extraction. The features considered in this paper can be used for distance (and similarity) calculations between two SQL queries although at that granularity, more features should probably be added to vectors. When attempting more features on OLTP workloads, care should be taken to ensure that cluster count stays within reasonable limits because otherwise, benefits of clustering begin to diminish.

7.4 Distance functions

Distance functions and similarity scores play central roles in clustering because inter-object distances are required. Such distance functions as cosine, Jaccard, and Hamming have been tried to cluster SQL queries [19, 20, 37], although only the Euclidean has been tried at the transaction level before [17]. In [37], cosine distance is used to find similarity between a SQL query and its context. The definition does not use angular cosine distance though, and the

dot-product-based definition is not a metric as noted in Sect. 2.2. In [19], cosine distance is used to compute distance between two SQL queries, but the authors did not indicate which flavor of it was used. One takeaway from this paper on that topic is that such normalized distance functions as *ACD* outperform such unnormalized distance functions as Euclidean for SQL transaction clustering.

7.5 Feature extraction methodologies

Before this research, feature extraction has mined SQL text from DBMS logs [19, 20], or MaxScale proxy server [18]. As mentioned in Sect. 1, mining SQL logs is inexact and error-prone because of the language's complexity. (Commercial DBMS engines have not provided API access to their parsers.)

For the past 15 years at least, modern DBMS systems have been exposing dynamic data from running servers as sets of relations on which SQL queries can be posed [21–24]. This approach has proven to be practical for three reasons. First, data of various granularity can be cheaply maintained in server data structures because no transaction semantics need to be enforced. Second, a row-set needs to be materialized from such data structures only on demand when a query arrives. Third, SQL queries on such row-sets enable powerful information extraction. This research has demonstrated that in future, intelligent database systems can provide clustering and classification functionalities using the same or similar data sets.

8 Conclusions and future work

DBMS administration work is getting increasingly complex because of the workload diversity, and DBA's having to monitor multiple applications in cloud environments. This research lends a helping hand by adding some intelligence to complicated software systems. In particular, by clustering the transactions, a 'birds-eye' view of the system is provided that makes performance troubleshooting for such scenarios as unexpected transaction rollbacks, performance drifts, bottleneck identifications, and so on. Others have started to investigate how machine learning can help query optimization [13], and this research is complementary to that thread of work.

Previous research had suggested a possibility that DBSCAN might be a suitable algorithm for SQL transaction clustering, but no substantial evidence for its suitability was gathered. This research demonstrates that angular cosine distance-based DBSCAN is suitable for the clustering task, and is an improvement over the Euclidean-based DBSCAN with either *SEKX* [26] or *DBSeer* [17] heuristics: better clusters form, and DBSCAN parameter tuning is simplified. As an added benefit, *ACD*-based DBSCAN is not very sensitive to the DBSCAN's parameter values, and therefore, any reasonable values for *Eps* and *minPts* will likely form acceptable transaction clusters. Another finding demonstrates that angular cosine distance function finds better clusters than the well-known dot-product-based cosines.

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. The server-side framework suggested in this paper allows storage of feature values in table-, statement-, or transaction-level relations as appropriate. Usefulness of cluster-level signatures as established in Sect. 6.1 suggests that some cluster-level signatures—for example lock times

and row counts—can also be kept at statement-level. Doing that will enable troubleshooting to drill-down from a transaction to the statements within. Multiple cluster-level signatures may help because an application may have distinct 'peak' and 'off-peak' behaviors.

Future work can investigate whether *ACD* is suitable with such clustering algorithms as BIRCH [38], *k*-means [39] and SOTA [40].

Last, but not least, server-side feature extraction can be attempted in other modern database systems using minor extensions to preexisting scaffoldings, and then transaction-cluster-level data monitoring can be provided out-of-the-box. A possibility also exists to perform clustering on the server side based on regularly and randomly collected transaction samples. The advantage of client-side data processing as advocated in this paper, of course, is that no clustering overhead is put on the DBMS server, but a major redesign on the DBMS server may make server-side clustering possible in future.

Acknowledgements Matthew Van Dijk implemented the server-side feature extraction framework.

References

1. Difallah DE, Pavlo A, Curino C, Cudré-Mauroux P (2013) OLTP-bench: an extensible testbed for benchmarking relational databases. PVLDB 7(4):277–288
2. GitHub: sysbench (2020). https://github.com/akopytov/sysbench
3. Verbitski A, Gupta A, Saha D, Brahmadesam M, Gupta K, Mittal R, Krishnamurthy S, Maurice S, Kharatishvili T, Bao X (2017) Amazon aurora: design considerations for high throughput cloud-native relational databases. In: Proceedings of the 2017 ACM international conference on management of data, SIGMOD conference 2017, Chicago, IL, USA, pp 1041–1052. https://doi.org/10.1145/3035918.3056101. https://doi.org/10.1145/3035918.3056101
4. Gan J, Tao Y (2015) DBSCAN revisited: mis-claim, un-fixability, and approximation. In: Proceedings of the ACM SIGMOD conference, pp 519–530. https://doi.org/10.1145/2723372.2737792
5. Schubert E, Sander J, Ester M, Kriegel H, Xu X (2017) DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. ACM Trans Database Syst 42(3):19–11921
6. MariaDB (2020) MariaDB MaxScale. https://mariadb.com/kb/en/maxscale/
7. Harizopoulos S, Ailamaki A (2004) STEPS towards cache-resident transaction processing. In: (e)Proceedings of the thirtieth international conference on very large data bases, VLDB 2004, Toronto, Canada, pp 660–671. https://doi.org/10.1016/B978-012088469-8.50059-0. http://www.vldb.org/conf/2004/RS18P1.PDF
8. Marathe AP (2021) DBMS performance troubleshooting in cloud computing using transaction clustering. In: Proceedings of the EDBT 2021 Conference, pp. 463–468 (2021). https://doi.org/10.5441/002/edbt.2021.52
9. Li Q, Kraft P, Cafarella M, Demiralp c, Graefe G, Kozyrakis C, Stonebraker M, Suresh L, Zaharia M (2023) Transactions make debugging easy. In: Proceedings of the CIDR 2023 conference, Amsterdam, The Netherlands, Jan. 8–11
10. Ester M, Kriegel H, Sander J, Xu X (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), pp 226–231
11. Yoon DY, Niu N, Mozafari B (2016) DBSherlock: a performance diagnostic tool for transactional databases. In: Proceedings of the 2016 international conference on management of data, pp 1599–1614 . https://doi.org/10.1145/2882903.2915218
12. Scikit Learn: DBSCAN (2019). https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html
13. Marcus RC, Negi P, Mao H, Zhang C, Alizadeh M, Kraska T, Papaemmanouil O, Tatbul N (2019) Neo: a learned query optimizer. Proc VLDB Endow 12(11):1705–1718. https://doi.org/10.14778/3342263.3342644
14. Bermbach D, Wittern E, Tai S (2017) Cloud service benchmarking-measuring quality of cloud services from a client perspective. Springer, New York City
15. Wikipedia: Cosine similarity (2019). https://en.wikipedia.org/wiki/Cosine_similarity

16. Leskovec J, Rajaraman A, Ullman JD (2014) Mining of massive datasets, 2nd edn. Cambridge University Press, Cambridge
17. GitHub: DBSeer (2020). https://github.com/barzan/dbseer
18. Mozafari B, Curino C, Jindal A, Madden S (2013) Performance and resource modeling in highly-concurrent OLTP workloads. In: Proceedings of the ACM SIGMOD international conference on management of data, SIGMOD 2013, New York, NY, USA, pp 301–312. https://doi.org/10.1145/2463676.2467800
19. Makiyama VH, Raddick J, Santos RDC (2015) Text mining applied to SQL queries: a case study for the SDSS skyserver, vol 1478. CEUR-WS.org, Aachen, Germany, pp 66–72. http://ceur-ws.org/Vol-1478/paper7.pdf
20. Kul G, Luong DTA, Xie T, Chandola V, Kennedy O, Upadhyaya SJ (2018) Similarity metrics for SQL query clustering. IEEE Trans Knowl Data Eng 30(12):2408–2420. https://doi.org/10.1109/TKDE.2018.2831214
21. MySQL: Chapter 26 MySQL Performance Schema (2020). https://dev.mysql.com/doc/refman/8.0/en/performance-schema.html
22. Microsoft: System Dynamic Managemenet Views (2019). https://docs.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/system-dynamic-management-views?view=sql-server-ver15
23. Oracle: About Dynamic Performance Views (2020). https://docs.oracle.com/cd/B19306_01/server.102/b14237/dynviews_1001.htm#i1398692
24. PostgreSQL: The Statistics Collector (2020). https://www.postgresql.org/docs/9.6/monitoring-stats.html
25. GitHub: OLTP-Bench II (2020). https://github.com/timveil-cockroach/oltpbench
26. Sander J, Ester M, Kriegel H, Xu X (1998) Density-based clustering in spatial databases: the algorithm GDBSCAN and its applications. Data Min Knowl Discov 2(2):169–194
27. Lingas A (1994) A linear-time construction of the relative neighborhood graph from the Delaunay triangulation. Comput Geom 4:199–208. https://doi.org/10.1016/0925-7721(94)90018-3
28. Marathe AP (1995) The weighted relative neighbourhood graph. Master's thesis, York University, Toronto, Ontario, Canada
29. Marathe AP (2020) LRZ convolution: an algorithm for automatic anomaly detection in time-series data. In: Proceedings of the SSDBM 2020 conference, pp 1–1112
30. TPC-C (1992). http://www.tpc.org/tpcc/
31. Wikipedia: Kernel-based Virtual Machine (2020). https://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine
32. Wikipedia: Ruby on Rails (2020). https://en.wikipedia.org/wiki/Ruby_on_Rails
33. Wikipedia: Django (web framework) (2020). https://en.wikipedia.org/wiki/Django_(web_framework)
34. Stonebraker M, Madden S, Abadi DJ, Harizopoulos S, Hachem N, Helland P (2007) The end of an architectural era (it's time for a complete rewrite). In: Proceedings of the 33rd international conference on very large data bases. University of Vienna, Austria, pp 1150–1160. http://www.vldb.org/conf/2007/papers/industrial/p1150-stonebraker.pdf
35. Xu D, Tian Y (2015) A comprehensive survey of clustering algorithms. Ann Data Sci 2:165–193
36. Li G, Zhou X, Li S, Gao B (2019) Qtune: a query-aware database tuning system with deep reinforcement learning. Proc VLDB Endow 12(12):2118–2130
37. Agrawal R, Rantzau R, Terzi E (2006) Context-sensitive ranking. In: Proceedings of the ACM SIGMOD conference, pp. 383–394. https://doi.org/10.1145/1142473.1142517
38. Zhang T, Ramakrishnan R, Livny M (1997) BIRCH: a new data clustering algorithm and its applications. Data Min Knowl Discov 1(2):141–182
39. Lloyd SP (1982) Least squares quantization in PCM. IEEE Trans Inf Theory 28(2):129–136
40. Dopazo J, Carazo JM (1997) Phylogenetic reconstruction using an unsupervised growing neural network that adopts the topology of a phylogenetic tree. J Mol Evol 44:226–233

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Arunprasad P. Marathe is a Canadian citizen of Indian origin. Born in Mumbai, Maharashtra, India, he grew up in the adjoining state of Gujarat. He obtained his bachelor's, master's, and doctorate degrees from M. S. University of Baroda (1991), York University, Toronto (1995), and University of Waterloo (2001), respectively, all in computer science. His research interests include database management systems, distributed systems, software engineering, algorithms, scientific computing, theoretical computer science, compilers, and statistics. He has developed database systems on the server-side (SQL Server, DB2, and MySQL at Microsoft, IBM, and Huawei, respectively) and client-side (using SQL Server at VersaBank). He worn a "best teacher" award—as voted by students—at the M. S. University of Baroda where he worked as a lecturer between 1991 and 1993.