DBSherlock: A Performance Diagnostic Tool for Transactional Databases*

Dong Young Yoon Ning Niu Barzan Mozafari

University of Michigan, Ann Arbor
{dyoon, nniu, mozafari}@umich.edu

ABSTRACT

Running an online transaction processing (OLTP) system is one of the most daunting tasks required of database administrators (DBAs). As businesses rely on OLTP databases to support their mission-critical and real-time applications, poor database performance directly impacts their revenue and user experience. As a result, DBAs constantly monitor, diagnose, and rectify any performance decays.

Unfortunately, the manual process of debugging and diagnosing OLTP performance problems is extremely tedious and non-trivial. Rather than being caused by a single slow query, performance problems in OLTP databases are often due to a large number of concurrent and competing transactions adding up to compounded, non-linear effects that are difficult to isolate. Sudden changes in request volume, transactional patterns, network traffic, or data distribution can cause previously abundant resources to become scarce, and the performance to plummet.

This paper presents a practical tool for assisting DBAs in quickly and reliably diagnosing performance problems in an OLTP database. By analyzing hundreds of statistics and configurations collected over the lifetime of the system, our algorithm quickly identifies a small set of potential causes and presents them to the DBA. The root-cause established by the DBA is reincorporated into our algorithm as a new causal model to improve future diagnoses. Our experiments show that this algorithm is substantially more accurate than the state-of-the-art algorithm in finding correct explanations.

Keywords

Transactions; OLTP; performance diagnosis; anomaly detection

1. INTRODUCTION

Many enterprise applications rely on executing transactions against their database backend to store, query, and update data. As a result, databases running online transaction processing (OLTP) workloads are some of the most mission-critical software components for enterprises. Any service interruptions or performance hiccups in these databases often lead directly to revenue loss.

Thus, a major responsibility of database administrators (DBAs) in large organizations is to constantly monitor their OLTP workload for any performance failures or slowdowns, and to take appropriate actions promptly to restore performance. However, diagnosing the root cause of a performance problem is generally tedious, as it requires the DBA to consider many possibilities by manually inspecting queries and various log files over time. These challenges are exacerbated in OLTP workloads because performance problems cannot be traced back to a few demanding queries or their poor execution plans, as is often the case in analytical workloads. In fact, most transactions take only a fraction of a millisecond to complete. However, tens of thousands of concurrent transactions competing for the same resources (e.g., CPU, disk I/O, memory) can create highly non-linear and counter-intuitive effects on database performance. Minor changes in an OLTP workload can push the system into a new performance regime, quickly making previously abundant resources scarce.

However, it can be quite challenging for most DBAs to explain (or even investigate) such phenomena. Modern databases and operating systems collect massive volumes of detailed statistics and log files over time, creating an exponential number of subsets of DBMS variables and statistics that may explain a performance decay. For instance, MySQL maintains over 260 different statistics and variables (see Section 2.1) and commercial DBMSs collect thousands of granular statistics (e.g., Teradata [12]). Unfortunately, existing databases fail to provide DBAs with effective tools for analyzing performance problems using these rich datasets, aside from basic visualization and monitoring mechanisms. As a consequence, highly-skilled and highly-paid DBAs (a scarce resource themselves) spend many hours diagnosing performance problems through different conjectures and manually inspecting various queries and log files, until the root cause is found [13].

To avoid this tedious, error-prone, and adhoc procedure, we propose a performance explanation framework called `DBSherlock` that combines techniques from outlier detection and causality analysis to assist DBAs in diagnosing performance problems more easily, more accurately, and in a more principled manner. Through `DBSherlock`'s visual interface, the user (e.g., a DBA) specifies certain instances of past performance that s/he deems abnormal (and optionally, normal). `DBSherlock` then automatically analyzes large volumes of past statistics to find the most likely causes of the user-perceived anomaly, presenting them to the user along with a confidence value, either in the form of (i) concise predicates describing the combination of system configurations or workload characteristics causing the performance anomaly, or (ii) high-level diagnoses based on the existing causal models in the system. The DBA can then identify the actual cause within these few possibilities. Once the actual cause is confirmed by the DBA, his/her feedback is inte-

*`DBSherlock` is now open-sourced at http://dbseer.org.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

ISBN 978-1-4503-3531-7/16/06...$15.00

DOI: http://dx.doi.org/10.1145/2882903.2915218

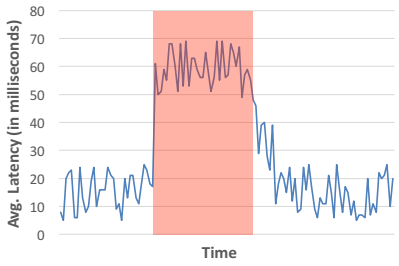

Figure 1: Identifying the root cause of a performance anomaly is non-trivial, as different causes may lead to the same performance pattern.

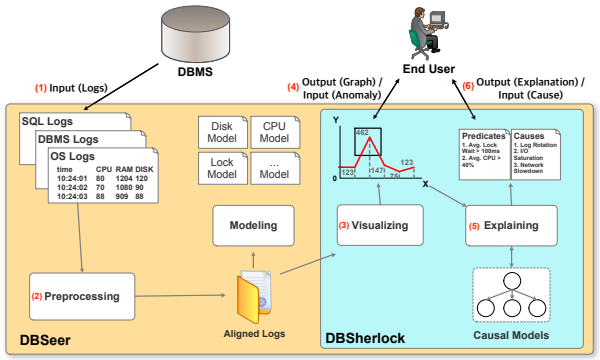

Figure 2: The workflow in DBSherlock.

grated back into DBSherlock to improve its causal models and future diagnoses.

As an example, consider Figure 1, which is a graph of average transaction latencies over time. In practice, finding the root cause of this latency spike can be quite challenging. In fact, we observe nearly the same performance plot in the following situations: (i) if the overall workload suddenly spikes, (ii) if the number of poorly written queries spikes, or (iii) if a network hiccup occurs. To establish the correct cause, the DBA has to plot several other performance metrics during the same timespan as the latency spike. This means choosing among hundreds of DBMS, OS, and network telemetry, and even inspecting several queries manually. However, DBSherlock can significantly narrow this search space by generating appropriate explanatory predicates that help the DBA distinguish between the different possible causes. In the particular example of Figure 1, DBSherlock's statistical analysis will lead to different predicates depending on the cause. When (i) has occurred, DBSherlock generates a predicate showing an increase in the number of lock waits and running DBMS threads compared to normal. In the case of (ii), DBSherlock's predicates indicate a sudden rise of next-row-read-requests as well as the CPU usage of the DBMS. Finally, (iii) leads to a predicate showing a lower than usual number of network packets sent or received during a specific time. In other words, DBSherlock's predicates help explain the root cause by bringing appropriate signals and metrics to the DBA's attention. (In Section 6, we show how DBSherlock can also provide human-readable causes in addition to raw predicates.)

Note that designing a tool for performance diagnosis is a challenging task due to the exponential number of combinations of variables and statistics that may explain the cause of a performance decay, making a naïve enumeration algorithm infeasible. Though off-the-shelf algorithms for feature selection exist, they are primarily designed to maximize a machine learning algorithm's predictive power rather than its explanatory and diagnostic power. Similarly, decision trees (e.g., PerfXplain [34]) and robust statistics (e.g., PerfAugur [41]) have been used for automatic performance explanation of map-reduce jobs and cloud services, respectively. However, such models are more likely to find secondary symptoms when the root cause of the anomaly is outside the database and not directly captured by the collected statistics. (In Section 6, we show that constructing and using causal models leads to significantly more relevant explanations.) Finally, while sensitivity-analysis-based techniques (e.g., Scorpion [46]) are highly effective in finding the individual tuples most responsible for extreme aggregate values in scientific computations, they are not applicable to performance diagnosis of OLTP workloads. This is because databases often avoid prohibitive logging overheads by maintaining aggregate statistics rather than detailed statistics for individual transactions. For instance, instead of recording each transaction's wait time for locks, MySQL and Postgres only record the total wait time for locks across all transactions. Thus, listing individual transactions is impractical (and often unhelpful for diagnosis, due to the complex interactions among concurrent transactions).

In this paper, we make the following contributions:

1. To the best of our knowledge, we propose the first algorithm specifically designed to explain and diagnose performance anomalies in highly-concurrent and complex OLTP workloads.

2. We adopt the notion of causality from the artificial intelligence (AI) literature, and treat user feedback as causal models in our diagnostic tool. We formally define a notion of *confidence* to combine predicate-based explanations and causal model predictions in a principled manner.

3. We propose a new automatic anomaly detection algorithm with a competitive accuracy to that of an expert. We also introduce a framework to prune secondary symptoms using basic forms of domain knowledge.

4. We evaluate DBSherlock against the state-of-the-art performance explanation technique, across a wide range of performance problems. DBSherlock's predicates on average achieve 28% (and up to 55%) higher F1-measures[1] than those generated by previous techniques.

Section 2 describes the high-level overview of DBSherlock. Sections 3 and 4 present our criterion and algorithm for generating predicate-based explanations, respectively. Section 5 describes our technique for incorporating domain knowledge and pruning secondary symptoms from our explanations. Section 6 explains how our system incorporates user feedback (when available) in the form of casual models in order to provide higher-level, descriptive explanations. Section 7 explains how automatic anomaly detection techniques can be combined with DBSherlock. Section 8 describes our experimental results.

2. SYSTEM OVERVIEW

DBSherlock's workflow for performance explanation and diagnosis consists of six steps, as shown in Figure 2.

1. **Data Collection.** DBSherlock collects various log files, configurations, and statistics from the DBMS and OS.

2. **Preprocessing.** The collected logs are summarized and aligned by their timestamps at fixed time intervals (e.g., every second).

[1]F1-measure (a.k.a. balanced F-score) is a commonly used measure of a test's accuracy. It considers both the precision p and the recall r of the test, and is defined as: $F_1 = 2 \cdot \frac{p \cdot r}{p+r}$.

3. **Visualization.** Through DBSherlock's graphical user interface, our end user (e.g., a DBA) can generate scatter plots of various performance statistics of the DBMS over time.

4. **Anomaly Detection.** If the end user deems any of the performance metrics of the DBMS unexpected, abnormal, or suspicious in any period of time, s/he can simply select that region of the plot and ask DBSherlock for an explanation of the observed anomaly. Alternatively, users can also rely on DBSherlock's automatic anomaly detection feature.

5. **Anomaly Explanation.** Given the user-perceived region of anomaly, DBSherlock analyzes the collected statistics and configurations over time and explains the anomaly using either descriptive predicates or actual causes.

6. **Anomaly Diagnosis and User Feedback.** Using DBSherlock's explanations as diagnostic clues, the DBA attempts to identify the root cause of the observed performance problem. Once s/he has diagnosed the actual cause, s/he provides evaluative feedback to DBSherlock. This feedback is then incorporated in DBSherlock as a causal model and used for improving future explanations.

Next, we discuss these steps in more detail: steps 1–2 in Section 2.1, steps 3–4 in Section 2.2, and steps 5–6 in Section 2.3. Then, we list the current limitations of DBSherlock in Section 2.4.

2.1 Data Collection and Preprocessing

We have implemented DBSherlock as a module for DBSeer [1]. DBSeer is an open-source suite of database administration tools for monitoring and predicting database performance [37, 38, 47]. We have integrated our DBSherlock into DBSeer for two reasons. First, adding performance diagnosis and explanation features will greatly benefit DBSeer's current users in gaining deeper insight into their workloads. Second, DBSherlock can simply rely on DBSeer's API for collecting and visualizing various performance statistics from MySQL and Linux (the systems used in our experiments). Here, we briefly describe the data collection and preprocessing steps performed by DBSeer and used by DBSherlock, i.e., components (1) and (2) in Figure 2.

DBSeer collects various types of performance data by passively observing the DBMS and OS *in situ* (i.e., as they are running in operation), via their standard logging features. Specifically, DBSeer collects the following data at one-second intervals [37]:

(i) Resource consumption statistics from the OS (in our case, *Linux*'s `/proc` data), e.g., per-core CPU usage, number of disk I/Os, number of network packets, number of page faults, number of allocated/free pages, and number of context switches.

(ii) Workload statistics from the DBMS (in our case, *MySQL*'s global status variables), e.g., number of logical reads, number of `SELECT`, `UPDATE`, `DELETE`, and `INSERT` commands executed, number of flushed and dirty pages, and the total lock wait-time.[2]

(iii) Timestamped query logs, containing start-time, duration, and the SQL statements executed by the system, as well as the query plans used for each query.

(iv) Configuration parameters from the OS and the DBMS, e.g., environment variables, kernel parameters, database server configurations, network settings, and (relevant) driver versions.

DBSeer further processes this data. First, it computes aggregate statistics about transactions executed during each time interval (e.g., their average and quantile latencies, total transaction counts, etc.).[3] These transaction aggregates are then aligned with the OS and DBMS statistics and configurations according to their timestamps, using the following format:

$$(\texttt{Timestamp}, \texttt{Attr}_1, \ldots, \texttt{Attr}_k)$$

where `Timestamp` marks the starting time of the 1-second interval during which these data were collected, and $\{\texttt{Attr}_1, \ldots, \texttt{Attr}_k\}$ are the attributes, comprised of the transaction aggregates and other categorical and numerical metrics collected from the database and operating system. DBSherlock uses these timestamped data for its performance explanation and diagnosis purposes.

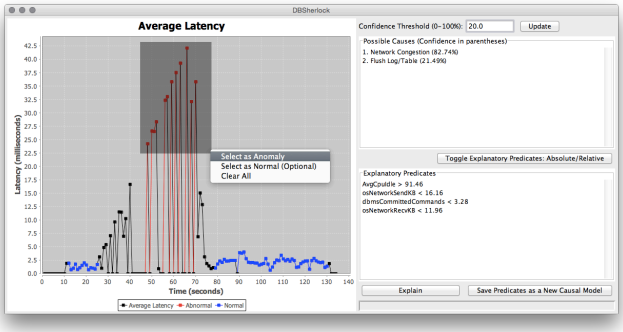

Figure 3: DBSherlock's user interface.

2.2 User Interface

DBSherlock comes with a graphical user interface, where users can plot a graph of various performance metrics over their time window of interest. This is shown as component (3) in Figure 2. For example, users might plot the average or 99% latency of transactions, number of disk I/Os, or CPU usage over the past hour, day or week. Figure 3 is an example of a scatter plot of the average latency of transactions over time. After inspecting this plot, the user can select some region(s) of the the graph where s/he finds some database metrics abnormal, suspicious, counter-intuitive, or simply worthy of an explanation. Regardless of the user's particular reason, we simply call the selected region(s) an *anomaly* (or call them *abnormal* regions). Optionally, the user can also select other areas of the graph that s/he thinks are normal (otherwise, the rest of the graph is implicitly treated as normal). After specifying the regions, the user asks DBSherlock to find likely causes or descriptive characteristics that best explain the observed *anomaly*.

When users cannot manually specify or detect the anomaly, DBSherlock relies on automatic anomaly detection (see Section 7).

2.3 System Output

Given a user-perceived anomaly, DBSherlock provides explanations in one of the following forms:

(i) predicates over different attributes of the input data; or

(ii) likely causes (and their corresponding confidence) based on existing causal models.

First, DBSherlock generates a number of predicates that identify anomalous values of some of the attributes that best explain the anomaly (Sections 3 and 4). For human readability, DBSherlock returns a conjunct of simple predicates to the user.[4] For example,

[2]To avoid performance overheads, DBSeer does not collect expensive statistics that are not maintained by default, e.g., fine-grained locking information.

[3]Since the number of transactions per second varies, we do not use individual query plans as attributes. Rather, we use their aggregate statistics, e.g., average cost estimates, number of index lookups.

[4]More complex predicates (e.g., with disjunction or negation) can easily overwhelm an average user, defeating DBSherlock's goal of being an effective tool for practitioners.

DBSherlock may explain an anomaly caused by a network slowdown by generating the following predicates:

$$\texttt{network\_send} < \texttt{10KB} \ \wedge \ \texttt{network\_recv} < \texttt{10KB} \ \wedge \ \texttt{client\_wait\_times} > \texttt{100ms} \ \wedge \ \texttt{cpu\_usage} < \texttt{5}$$

showing that there are active clients waiting without much CPU activity. (In Section 6, we show that with causal models, DBSherlock can provide even more descriptive diagnoses.) Once the user identifies the actual problem (network congestion, in this example) using these predicates as diagnostic hints, s/he can provide feedback to DBSherlock by accepting these predicates and labeling them with the actual *cause* found. This 'cause' and its corresponding predicates comprise a causal model, which will be utilized by DBSherlock for future diagnoses.

When there are any causal models in the system (i.e., from accepted and labeled predicates during previous sessions), DBSherlock calculates the *confidence* of every existing causal model for the given anomaly. This *confidence* measures a causal model's fitness for the given situation. DBSherlock then presents all existing causes in their decreasing order of confidence (as long as greater than a minimum threshold). When none of the causal models yield a sufficiently large confidence, DBSherlock does not show any causes and only shows the generated predicates to the user.

Note that DBSherlock's output in either case is only a *possible* explanation/cause of the anomaly, and it is ultimately the end user's responsibility to diagnose the *actual* root cause. The objective of DBSherlock is to provide the user with informative clues to *facilitate* fast and accurate diagnosis. In the rest of this paper, we use the terms *possible explanation* and *explanation* interchangeably, but always make a clear distinction between *possible* and *actual causes* as they are quite different from a causality perspective.

2.4 Current Limitations

The current implementation of DBSherlock has two limitations:

(i) DBSherlock finds an explanation for an anomaly if the anomaly affects at least one of the statistics available to the system.

(ii) Invariant characteristics of the system (e.g., fixed parameters or hardware specifications of the database server) are *not* considered a valid explanation of an anomaly.

It is straightforward to see the reason behind (i): if the anomaly does not manifest itself in any of the gathered statistics, DBSherlock has no means of distinguishing between abnormal and normal regions. Similarly for (ii), since the invariants of the system remain unchanged across the abnormal and normal regions, they cannot be used to distinguish the two. However, such invariants may have ultimately contributed to the anomaly in question. For instance, with a small buffer pool, dirty pages are flushed to disk frequently. Thus, when the number of concurrent transactions spikes, the pages may be flushed even more frequently. The increase in disk IOs may then affect transaction latencies. In such a case, DBSherlock reports the workload spike as the explanation for the increased latencies. While one might argue that small memory was the root cause of the problem, DBSherlock does not treat the memory size as a cause, as it is unchanged before and after the anomaly. Here, the workload spike can distinguish the two regions (see Section 3) and is hence returned as a cause to the user (i.e., with the justification that the memory was sufficient for the normal workload).

However, DBSherlock's reported cause can still be quite helpful even in the cases above. For example, even when presented with workload spike as an explanation of the performance slowdown, an experienced DBA may still rectify the problem by modifying system invariants (e.g., provisioning a larger memory or faster disk) or throttling the additional load.

3. PREDICATE GENERATION CRITERION

Given an abnormal region A, a normal region N, and input data T, we aim to generate a conjunct of predicates, where each predicate Pred is in one of the following forms: $\texttt{Attr}_i$ < x, $\texttt{Attr}_i$ > x, or x < $\texttt{Attr}_i$ < y when $\texttt{Attr}_i$ is numeric, and $\texttt{Attr}_i \in \{\texttt{x}_1, \ldots, \texttt{x}_c\}$ when $\texttt{Attr}_i$ is categorical. Intuitively, we desire a predicate that segregates the input tuples in A well from those in N. We formally define this quality as the separation power of a predicate.

Separation power of a predicate. Let T_N and T_A be the input tuples in the normal and abnormal regions, respectively. Also, let Pred(T) be the input tuples satisfying predicate Pred. Then the separation power (SP) of a predicate Pred is defined as:

$$\texttt{SP(Pred)} = \frac{|\texttt{Pred}(T_A)|}{|T_A|} - \frac{|\texttt{Pred}(T_N)|}{|T_N|} \quad (1)$$

In other words, a predicate's separation power is the ratio of the tuples in the abnormal region satisfying the predicate, subtracted by the ratio of the tuples in the normal region satisfying the predicate. A predicate with higher separation power is more capable of distinguishing (i.e., separating) the input tuples in the abnormal region from those in the normal one. Thus, DBSherlock's goal is to filter out individual attributes with low separation power.[5]

Identifying predicates with high separation power is challenging. First, one cannot find a predicate of high separation power by simply comparing the values of an attribute in the raw dataset.This is because real-world datasets and OS logs are noisy and attribute values often fluctuate regardless of the anomaly. Second, due to human error, users may not specify the boundaries of the abnormal regions with perfect precision. The user may also overlook smaller areas of anomaly, misleading DBSherlock to treat them as normal regions. These sources of error compound the problem of noisy datasets. Third, one cannot easily conclude that predicates with high separation power are the actual cause of an anomaly. They may simply be correlated with, or be symptoms themselves of the anomaly, and hence, lead to incorrect diagnoses. The following section describes our algorithm for efficiently finding predicates of highest separation power, while accounting for the first two sources of error. We deal with the third type of error in Section 5.

4. ALGORITHM

Our algorithm takes the aligned tuples as input (described in Section 2.1), which are separated between the abnormal and the normal regions (other tuples are ignored by DBSherlock).

Figure 4 illustrates a high-level overview of our predicate generation algorithm. The majority of our attributes are numeric (i.e., statistics), which are significantly noisier than our categorical attributes. As a result, our algorithm uses two additional steps for numeric attributes (Steps 3 and 4). (In our discussion, we highlight the differences for categorical attributes when applicable.) The first step is to discretize the domain of each attribute into a number of partitions (Step 1). Based on the user-specified abnormal and normal regions, DBSherlock labels each partition of an attribute as Abnormal, Normal, or Empty (Step 2 in Figure 4). Next, for numeric attributes, DBSherlock filters out some of the Abnormal and Normal partitions, which are mingled at this point, to find a predicate with high separation power (Step 3 in Figure 4). If the previous step is successful, the algorithm then fills the gap between the two separated sets of partitions and generates the candidate predicate accordingly (Steps 4 and 5 in Figure 4). The formal pseudo code of

[5]This strategy is similar to *single-variable classifiers* in machine learning literature, whereby variables' individual predictive power is used for feature selection [27].

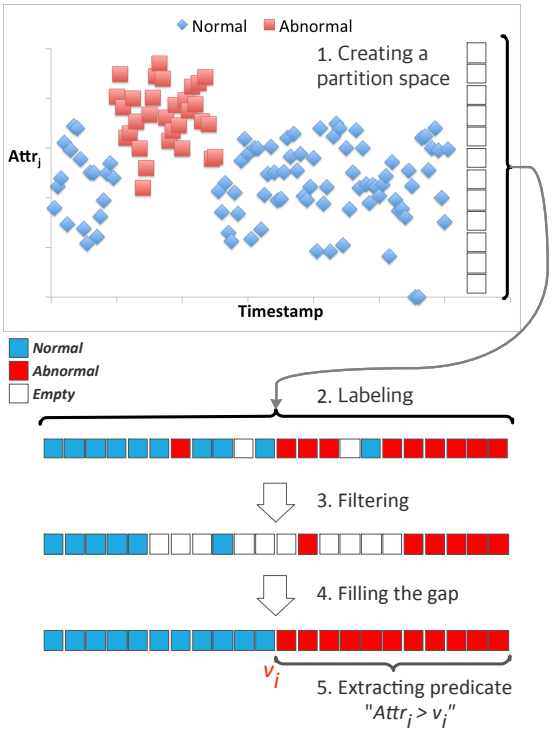

Figure 4: Example of finding a predicate over an attribute in the partition space. (Steps 3 & 4 are only applied to numeric attributes.)

these steps is presented in Algorithm 1. In the rest of this section, we explain each of these steps in detail.

4.1 Creating a Partition Space

DBSherlock starts by creating a discretized domain for each attribute, called a *partition space*.

For each numeric attribute $\texttt{Attr}_i$, we create R equi-width partitions $\texttt{P} = \{\texttt{P}_1, \ldots, \texttt{P}_R\}$ that range from $\texttt{Min}(\texttt{Attr}_i)$ to $\texttt{Max}(\texttt{Attr}_i)$. The width of each partition is

$$\frac{\texttt{Max}(\texttt{Attr}_i) - \texttt{Min}(\texttt{Attr}_i)}{R}$$

We denote the lower and upper bounds of partition $\texttt{P}_j$ as $\texttt{lb}(\texttt{P}_j)$ and $\texttt{ub}(\texttt{P}_j)$, respectively. $\texttt{P}_j$ contains any value val of $\texttt{Attr}_i$ where $\texttt{lb}(\texttt{P}_j) \leq \texttt{val} < \texttt{ub}(\texttt{P}_j)$.

For example, if the values of an attribute range from 0 to 100 and we discretize them into buckets of size 20, their partition space will be {[0,20), [20,40), [40,60), [60,80), [80,100)}. We use equi-width partitions to preserve and map the distribution of input tuples in the abnormal and normal regions into the partition space, as shown in Figure 4. A secondary goal of our discretization step is to reduce the influence of having more tuples with normal values than with abnormal values. Thus, our discretization enables us to focus on the distribution of an attribute's value across the two regions.

Here, the number of partitions, R, is an important parameter, which decides the trade-off between our algorithm's computation time (see Section 4.6) and the ability of the individual partitions to distinguish normal from abnormal tuples. By default, DBSherlock uses 1,000 partitions (R=1,000) for numeric attributes, which is large enough to separate abnormal and normal tuples, yet small enough to optimize computation time. (We study the effect of different values of R on our predicate accuracy and run time in Appendix D.)

Algorithm 1: Predicate Generation.

Inputs: T: input tuples (with k attributes)
A: abnormal region
N: normal region
R: number of partitions
θ: normalized difference threshold
δ: anomaly distance multiplier
Output: π: list of predicates with high separation power

$\pi \leftarrow \emptyset$ **// start with no predicate**
foreach $\texttt{Attr}_i \in \{\texttt{Attr}_1, \ldots, \texttt{Attr}_k\}$ **do**
  if $\texttt{Attr}_i$ is numeric **then**
    $P \leftarrow$ create a partition space for $Attr_i$ with R partitions
    $P_L \leftarrow$ label P based on tuples T and regions A, N
    $P_{L,Filtered} \leftarrow$ filter P_L
    $P^* \leftarrow$ fill the gap in $P_{L,Filtered}$ based on δ
    $\text{Norm}(\texttt{Attr}_i) \leftarrow$ Normalization of $\texttt{Attr}_i$ into $[0, 1]$ values
    $\mu_A \leftarrow$ Average of $\text{Norm}(\texttt{Attr}_i)$ for tuples in A
    $\mu_N \leftarrow$ Average of $\text{Norm}(\texttt{Attr}_i)$ for tuples in N
    $d \leftarrow |\mu_A - \mu_N|$
    if P^* contains a single block of consecutive abnormal partitions ***and*** $d > \theta$ **then**
      Pred $\leftarrow$ extract a candidate predicate from P^*
      $\pi \leftarrow \pi \cup$ Pred **// add Pred into the list**
    end
  else if $\texttt{Attr}_i$ is categorical **then**
    $P \leftarrow$ create a partition space for $Attr_i$ with $|Unique(\texttt{Attr}_i)|$ partitions
    $P_L \leftarrow$ label P based on tuples T and regions A, N
    if P_L has at least one abnormal partition **then**
      Pred $\leftarrow$ extract a candidate predicate from P_L
      $\pi \leftarrow \pi \cup$ Pred **// add Pred into the list**
    end
  end
end
return π

For each categorical attribute $\texttt{Attr}_i$, we create $|Unique(\texttt{Attr}_i)|$ number of partitions. Here, $|Unique(\texttt{Attr}_i)|$ is the number of unique values found in our dataset for $\texttt{Attr}_i$, i.e., one partition per each value of the attribute. We use $\texttt{C}_j$ to denote the (categorical) value represented by partition $\texttt{P}_j$. Unlike numeric attributes, the order of partitions for categorical attributes is unimportant.

4.2 Partition Labeling

Once the partition space is created, the next step is to mark each partition with one of three labels: {Empty, Normal, Abnormal}. For a numeric attribute $\texttt{Attr}_i$, an input tuple belongs to partition $\texttt{P}_j$, if the tuple's value for $\texttt{Attr}_i$ lies within $\texttt{P}_j$'s boundaries. If every tuple belonging to $\texttt{P}_j$ lies in the abnormal region specified by the user, $\texttt{P}_j$ is labeled as Abnormal. Conversely, if every tuple belonging to $\texttt{P}_j$ lies in the normal region, $\texttt{P}_j$ is labeled as Normal. Otherwise, the partition label is left Empty. (See Figure 4.)

For a categorical attribute, an input tuple belongs to a partition $\texttt{P}_j$, if the tuple's value for $\texttt{Attr}_i$ equals the category value of the partition, i.e., $\texttt{Attr}_i = \texttt{C}_j$. Since our categorical attributes are less noisy, we use a simpler labeling strategy. Let $\texttt{P}_j\texttt{(A)}$ and $\texttt{P}_j\texttt{(N)}$ be the number of tuples belonging to $\texttt{P}_j$ in the abnormal and normal

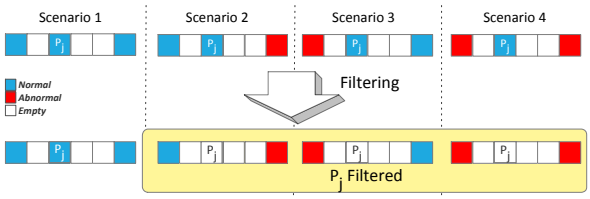

Figure 5: Different scenarios of filtering P_j in the partition space.

regions, respectively. P_j is labeled as Abnormal if $P_j(A) > P_j(N)$. Similarly, P_j is labeled as Normal if $P_j(A) < P_j(N)$. Otherwise, the partition label is left with an Empty label. For categorical attributes, our algorithm extracts a candidate predicate from the partition space right after the labeling step (see Section 4.5), skipping the next two steps.

4.3 Partition Filtering

After labeling, we filter some of the Normal and Abnormal partitions by replacing their labels with an Empty one. The filtering step is only applied to our numeric attributes, which are quite noisy. During this step, a partition P_j's label is replaced with Empty, if its original label is different from either of its two non-Empty adjacent partitions (i.e., the closest non-Empty partitions on the left and right side of P_j in the partition space). Figure 5 demonstrates various cases where a partition P_j is filtered out. Note that the only case where P_j remains unchanged is when both of its non-Empty adjacent partitions have the same label as P_j itself (shown as Scenario 1 in Figure 5). If we only have a single Normal or Abnormal partition to begin with, we deem it significant and do not filter it. Once all non-Empty partitions are processed, their labels are changed to Empty *simultaneously*. We do not perform this procedure incrementally to prevent a situation where partitions continuously filter each other out. If filtering is performed incrementally, the two partitions at each end of the partition space will also get filtered in Scenarios 2 and 3 in Figure 5.

Our filtering strategy aims to separate the Normal and Abnormal partitions that are originally mixed across the partition space (e.g., due to the noise in the data or user errors). If there is a predicate on attribute $\texttt{Attr}_i$ that has high separation power, the Normal and Abnormal partitions are very likely to form well-separated clusters after the filtering step. This step mitigates some of the negative effects of noisy data or user's error, which could otherwise exclude a predicate with high separation power from the output. This idea is visually shown in Figure 4.

4.4 Filling the Gaps

This step is only applied to numeric attributes. After the filtering step, there will be larger blocks of consecutive Abnormal and Normal partitions, separated by Empty partitions. These Empty partitions were either initially Empty or were filtered out during the filtering step. Our algorithm fills these gaps before the predicate generation by labeling these Empty partitions as either Normal or Abnormal as follows.

We compare the distance of each Empty partition P_j to its two adjacent non-Empty partitions. If both adjacent partitions have the same label or if P_j has only one adjacent non-Empty partition (i.e., $j{=}1$ or $j{=}R$), P_j will receive the same label as its adjacent non-Empty partition(s) at the end of this step. If P_j's two adjacent non-Empty partitions have different labels, we calculate P_j's distance to each partition and assign it the label of the closer partition.

There is a special case where only Abnormal partitions remain after the filtering step. In this case, if we naïvely fill the gaps, every partition will become Abnormal and the algorithm will not find any predicates for the attribute. To handle this special case, we calculate the average value of the attribute over the input tuples in the normal region and label the partition that contains this average value as Normal regardless of its previous label. Then we fill the gaps according to the previously described procedure. (Without any Normal partitions, we will not be able to determine the direction of the predicate in the next step of the algorithm, i.e., whether $\texttt{Attr}_i < v$ or $\texttt{Attr}_i > v$.)

To control the behavior of our algorithm, we also introduce a parameter δ, called the *anomaly distance multiplier*. When the Empty partition P_j is processed by the above procedure, we multiply its distance to its adjacent Abnormal partition by δ. Thus, $\delta{>}1$ will cause more Empty partitions to be labeled as Normal while $\delta{<}1$ results in more Empty partitions being labeled as Abnormal. In other words, with parameter δ, we can tune our predicates: $\delta{<}1$ for more general predicates (i.e., more likely to flag tuples as abnormal) and $\delta{>}1$ for more specific ones (i.e., less likely to flag tuples as abnormal). By default, DBSherlock uses $\delta = 10$. (We study the effect of different values of δ on our predicates in Appendix D.)

4.5 Extracting Predicates from Partitions

This step is applied to both numeric and categorical attributes using slightly different procedures. For numeric attributes, the previous filtering step allows us to find attributes that have a predicate with high separation power, but there is still a possibility that some of these attributes are not related to the actual cause of the anomaly. To mitigate this problem, we perform the following procedure. First, we normalize each numeric attribute $\texttt{Attr}_i$ by subtracting its minimum value from its original values $\texttt{val}_i$ and dividing them by the attribute's range:

$$\texttt{Norm}(\texttt{val}_i) = \frac{\texttt{val}_i - \texttt{Min}(\texttt{Attr}_i)}{\texttt{Max}(\texttt{Attr}_i) - \texttt{Min}(\texttt{Attr}_i)} \tag{2}$$

This results in the values of an attribute to range in $[0, 1]$. Let μ_A and μ_N be the average values of $\text{Norm}(\texttt{Attr}_i)$ for tuples in the abnormal and normal regions, respectively. DBSherlock extracts a candidate predicate from $\texttt{Attr}_i$'s partition space only if $|\mu_A - \mu_N| > \theta$, where θ is a parameter called the *normalized difference threshold*. The user can tune this threshold to adjust the selectivity of DBSherlock in finding predicates (we study the effect of this parameter in Appendix D).

After performing these normalization and thresholding procedures, we can extract candidate predicates from the partition space, as follows. As noted in section 3, we only seek predicates of the form $\texttt{Attr}_i < x$, $\texttt{Attr}_i > x$, and $x < \texttt{Attr}_i < y$. In the partition space, these types of predicates correspond to a single block of consecutive Abnormal partitions. Therefore, we extract a candidate predicate Pred for an attribute $\texttt{Attr}_i$ if and only if there is a single block of consecutive Abnormal partitions.

For categorical attributes, our procedure for extracting a candidate predicate is much simpler. DBSherlock traverses the partition space of such attributes and extracts each category value C_j if its partition P_j is labeled as Abnormal. A predicate for a categorical attribute is of the form $\texttt{Attr}_i \in \{c_1, \ldots, c_l\}$, where l is the number of partitions labeled as Abnormal and c_i's are their corresponding category values.

4.6 Time Complexity

For each attribute, our predicate generation algorithm scans the input tuples to label each partition. Then, it iterates over these partitions twice in the subsequent steps (i.e., 'Filtering' and 'Filling the gap' in Figure 4). Thus, the time complexity of DBSherlock's predicate generation algorithm is $O(k(X{+}R))$, where R is the number of

partitions,[6] X is the number of input tuples, and k is the number of attributes. In Appendix D, we study the effectiveness and run-time of our algorithm for different values of these parameters.

5. INCORPORATING DOMAIN KNOWLEDGE

Our algorithm extracts predicates that have a high diagnostic power (see Section 8). However, some of these predicates may be secondary symptoms of the root cause, which if removed, can make the diagnosis even easier. This is because the fact that $\texttt{Pred}_i$ **implies** an anomaly, we cannot conclude that it also **causes** it. In fact, there could be another predicate, say $\texttt{Pred}_j$, causing both $\texttt{Pred}_i$ and the anomaly.

Thus, to further improve the accuracy of our predicates and prune secondary symptoms, DBSherlock allows for incorporating domain knowledge of attributes' semantics into the system. However, note that this mechanism is an optional feature, and as we show in our experiments, DBSherlock produces highly accurate explanations even without any domain knowledge (see Section 8.3). Also, DBSherlock is bootstrapped with domain knowledge only once for each specific version of OS or DBMS. In other words, DBAs do not need to modify this, as the semantics of DBMS and OS variables do not depend on the workload, e.g., OS CPU Usage always has the same meaning regardless of the specific workload.

Every piece of domain knowledge is encoded as a rule: $\texttt{Attr}_i \rightarrow \texttt{Attr}_j$. Each rule must satisfy the following conditions:

i. If predicates $\texttt{Pred}_i$ and $\texttt{Pred}_j$ (corresponding to attributes $\texttt{Attr}_i$ and $\texttt{Attr}_j$, respectively) are both extracted, $\texttt{Pred}_j$ is likely to be a secondary symptom of $\texttt{Pred}_i$.

ii. $\texttt{Attr}_i \rightarrow \texttt{Attr}_j$ and $\texttt{Attr}_j \rightarrow \texttt{Attr}_i$ cannot coexist.

For instance, if $\texttt{Attr}_i$ is the 'DBMS CPU Usage' and $\texttt{Attr}_j$ is the 'OS CPU Usage', then $\texttt{Attr}_i \rightarrow \texttt{Attr}_j$ is a valid rule since DBMS CPU usage effects OS CPU usage, but not vice versa.

However, given a rule $\texttt{Attr}_i \rightarrow \texttt{Attr}_j$, and the corresponding predicates $\texttt{Pred}_i$ and $\texttt{Pred}_j$, whether $\texttt{Pred}_j$ will be filtered out will require further analysis, since the domain knowledge may not be a perfect reflection of the reality either. For instance, the rule 'DBMS CPU Usage' $\rightarrow$ 'OS CPU Usage' may occasionally break. For example, there might be other attributes, such as 'Number of Processes' or 'Number of Threads' that are not utilized by the DBMS, but may affect 'OS CPU Usage'.

As a solution, DBSherlock tests the independence between $\texttt{Attr}_i$ and $\texttt{Attr}_j$ based on their continuous (or categorical) values. For continuous attributes, we discretize the two attributes $\texttt{Attr}_i$ and $\texttt{Attr}_j$ with γ equi-width bins for each attribute. We then construct a two-dimensional joint histogram from the input data, estimating the joint probability distribution of the two attributes. For categorical attributes, a joint histogram is constructed from the input data. For testing independence, we use the joint probability distribution of the two attributes to calculate their *mutual information*.

We denote the mutual information of two attributes $\texttt{Attr}_i$ and $\texttt{Attr}_j$ by $\texttt{MI}(\texttt{Attr}_i, \texttt{Attr}_j)$, defined as:

$$\texttt{MI}(\texttt{Attr}_i, \texttt{Attr}_j) = H(\texttt{Attr}_i) + H(\texttt{Attr}_j) - H(\texttt{Attr}_i, \texttt{Attr}_j)$$

where $H(\texttt{Attr}_i)$ is the entropy of the attribute $\texttt{Attr}_i$ and $H(\texttt{Attr}_i, \texttt{Attr}_j)$ is the joint entropy of the two attributes [18]. An independence factor $\kappa(\texttt{Attr}_i, \texttt{Attr}_j)$ of the two attributes is then calculated as follows:

$$\kappa(\texttt{Attr}_i, \texttt{Attr}_j) = \frac{\texttt{MI}(\texttt{Attr}_i, \texttt{Attr}_j)^2}{H(\texttt{Attr}_i)H(\texttt{Attr}_j)}$$

[6] A similar analysis applies if the number of partitions differ.

The value of κ will be 0, if the two attributes are independent and approaches 1 with higher dependence. We perform the independence test by comparing the value of κ with a threshold κ_t (by default, we use $\kappa_t = 0.15$), and the two attributes pass the test if $\kappa < \kappa_t$. If the two attributes do not pass the independence test, we conclude that the rule $\texttt{Attr}_i \rightarrow \texttt{Attr}_j$ is indeed valid, and $\texttt{Pred}_j$ is merely a secondary symptom of $\texttt{Pred}_i$ and filter out $\texttt{Pred}_j$. If the two attributes pass the independence test, we conclude that the rule $\texttt{Attr}_i \rightarrow \texttt{Attr}_j$ does not apply, and leave both predicates in the output.

For MySQL on Linux, the following four rules are sufficient to encode such relationships:

1. DBMS CPU Usage $\rightarrow$ OS CPU Usage
2. OS Allocated Pages $\rightarrow$ OS Free Pages
3. OS Used Swap Space $\rightarrow$ OS Free Swap Space
4. OS CPU Usage $\rightarrow$ OS CPU Idle

The first rule encodes the subset relationship. The last three rules encode the fact that one attribute is always a constant value minus the other attribute, and is thus uninteresting. In Section 8.6, we show that even without these rules, DBSherlock's accuracy drops by only 2–3%. We evaluate the effectiveness of this approach in pruning secondary symptoms in Appendix F.

6. INCORPORATING CAUSAL MODELS

Previous work on performance explanation [34] has only focused on generating explanations in the form of predicates. DBSherlock improves on this functionality by generating substantially more accurate predicates (20-55% higher F1-measure; see Section 8.4). However, a primary objective of DBSherlock is to go beyond raw predicates, and offer explanations that are more human-readable and descriptive. For example, the cause of a performance hiccup could be a network congestion due to a malfunctioning network router. Initially, the user will rely on DBSherlock's generated predicates as diagnostic clues to identify the actual cause of the performance problem more easily. However, once the root cause is diagnosed, s/he can notify DBSherlock as to what the actual cause was. DBSherlock then relates the generated predicates to the actual cause and saves them in the form of a causal model. This model will be consulted in future diagnoses to provide a human-readable explanation (i.e., 'malfunctioning router') for similar situations.

To utilize the user feedback, DBSherlock uses a simplified version of the *causal model* proposed in the seminal work of Halpern and Pearl [28]. Our causal model consists of two parts: *cause variable* and *effect predicates*. The *cause variable* is a binary, exogenous variable[7] labeled by the end user. When the *cause variable* is set to *true*, it activates all of its *effect predicates*. For example, Figure 6 is a causal model with 'Log Rotation' as the *cause variable* and three *effect predicates*. According to this model, if there is an event of 'Log Rotation' (i.e. *cause variable* is *true*) then these three *effect predicates* will also be true.

The following example describes how such causal models are constructed and used in DBSherlock. Consider a scenario where the user selects an abnormal region for which DBSherlock returns the following predicates:

$$\texttt{CpuWait} > 50\% \ \wedge \ \texttt{Latency} > 100\text{ms} \ \wedge \ \texttt{DiskWrite} > 5\text{MB/s}$$

[7] An exogenous variable is a variable whose values are determined by factors outside the model [28].

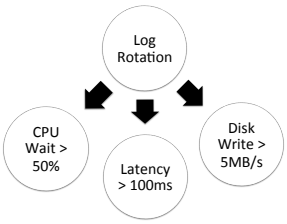

Figure 6: An example of a causal model in DBSherlock.

Also, suppose that the user is able to diagnose the actual cause of the problem with the help of these predicates; assume that by inspecting the recent system logs she establishes that the cause was the rotation of the redo log file.[8] Once this feedback is received from the user, DBSherlock can link this cause with these predicates as shown in Figure 6. After that, for every diagnosis inquiry in the future, DBSherlock calculates the *confidence* of this causal model and 'Log Rotation' is reported as a possible cause if its confidence is higher than a threshold λ. By default, DBSherlock displays only those causes whose confidence is higher than λ=20%. However, the user can modify λ as a knob (i.e., using a sliding bar) to interactively view fewer or more causes.

Over time, additional causal models might be added in the system as a result of inspecting new performance problems. When multiple causal models are available in the system, DBSherlock consults all of them (i.e., computes their confidence) and returns those models whose confidence is higher than λ to the user, presented in their decreasing order of confidence. When none of the causes offered by the existing models are deemed helpful by the user, she always has the option of asking DBSherlock to simply show the original predicates instead. Also, when none of the causal models achieve a confidence higher than λ (e.g., when the given anomaly has not been previously observed in the system), DBSherlock only displays the generated predicates. Again, once the cause is diagnosed and shared with the system, DBSherlock creates a new causal model to be used in the future. (See Figure 2 in Section 2.)

We evaluate the accuracy of the generated explanations in Section 8. Next, we explain how DBSherlock computes the confidence of each causal model (Section 6.1), and how it merges multiple models to improve their explanatory power (Section 6.2).

6.1 Confidence of a Causal Model

In DBSherlock we define the confidence of a causal model as the average separation power of its *effect predicates* in the partition space. Note that, unlike equation (1), here we use the partition space instead of the input tuples to reduce the effect of the noise in real-world data (see Section 4.1). Formally:

Confidence of a causal model. Let $\{\texttt{Pred}_1, \ldots, \texttt{Pred}_n\}$ be the *effect predicates* of a given causal model $\mathcal{M}$, where $\texttt{Pred}_i$ is defined over $\texttt{Attr}_i$. Also, let $P_{i,N}$ and $P_{i,A}$ be the partitions labeled as Normal and Abnormal in the partition space of $\texttt{Attr}_i$, respectively. We define the *confidence* $\mathcal{C}_\mathcal{M}$ of the causal model $\mathcal{M}$ as:

$$\mathcal{C}_\mathcal{M} = \frac{\sum_{i=1}^{n} \frac{|\texttt{Pred}_i(P_{i,A})|}{|P_{i,A}|} - \frac{|\texttt{Pred}_i(P_{i,N})|}{|P_{i,N}|}}{n} \quad (3)$$

where $\texttt{Pred}(P)$ is the set of partitions in P that satisfy predicate Pred.

The idea behind this definition is to estimate the likelihood of a *cause variable* being true given the normal and abnormal partitions, based on the assumption that if the model's *cause variable* is true, then its *effect predicates* are also likely to exhibit high separation power in their partition spaces.

[8] In MySQL, log rotations can cause performance hiccups when the adaptive flushing option is disabled.

6.2 Merging Causal Models

Among the *effect predicates* of a causal model, some predicates may have less or no relevance to the actual cause, e.g., some predicates could simply be a side-effect of the actual cause. Also, since the *effect predicates* of a single causal model reflect the specific values observed in a particular instance of an anomaly, they may not be applicable to other instances of the same cause. In DBSherlock, multiple causal models might be created for the same cause while analyzing different anomalies over time. DBSherlock can improve such causal models by merging them into a single one.

Merging causal models eliminates some of the unnecessary and less relevant *effect predicates*, while enabling relevant *effect predicates* to apply to different anomaly instances caused by the same cause. We merge two causal models by:

1. keeping only those *effect predicates* that are on attributes common to both models; and
2. merging two predicates on the same attribute into a single predicate that includes the boundaries (or categories) of both.

Suppose that we have two causal models with the same cause: $\mathcal{M}_1$ with the *effect predicates* {A > 10, B > 100, C > 20, E ∈ {'xx', 'yy', 'zz'}}, and $\mathcal{M}_2$ with the *effect predicates* {A > 15, C > 15, D < 250, E ∈ {'xx', 'zz'}}. To merge $\mathcal{M}_1$ and $\mathcal{M}_2$, we only keep {A > 10, C > 20, E ∈ {'xx', 'yy', 'zz'}} from $\mathcal{M}_1$ and {A > 15, C > 15, E ∈ {'xx', 'zz'}} from $\mathcal{M}_2$ since attributes A, C and E are common to both models.

Next, we compare the two predicates on the same attribute and merge them such that the merged predicate includes both. Here, merging {A > 10} and {A > 15} leads to {A > 10} and merging {C > 20} and {C > 15} leads to {C > 15}. Likewise, merging {E ∈ {'xx', 'yy', 'zz'}} and {E ∈ {'xx', 'zz'}} leads to {E ∈ {'xx', 'zz'}}. Thus, in this example, the *effect predicates* of the merged causal model will be {A > 10, C > 15, E ∈ {'xx', 'yy', 'zz'}}. Note that numeric predicates with different directions (e.g., {A > 10} and {A < 30}) are considered inconsistent. Such predicates are not merged and will be discarded.

We study the effect of merging causal models in Section 8.5, where we show that the merged causal models are on average 30% more accurate than the original models.

7. AUTOMATIC ANOMALY DETECTION

Sometimes, an anomaly may not be visually obvious to a human user inspecting a performance plot. In such situations, users may mistakenly specify a normal region as abnormal and vice versa. To aid with these cases, DBSherlock also provides an option for automatic anomaly detection. Thus, users can either (i) rely on DBSherlock to find and suggest anomalies to them, or (ii) continue to manually find anomalies but compare them with those found by DBSherlock for reassurance.

There is much work on outlier detection in different contexts [15, 19, 33, 40, 41, 42, 43, 44, 45, 46, 48]. In DBSherlock, we introduce an algorithm for the automatic detection of the anomaly regions. Our algorithm utilizes the DBSCAN clustering algorithm [25] and works as follows.

First, we normalize each attribute $\texttt{Attr}_i$, which is equivalent to the normalization step in our predicate generation algorithm (Equation (2) in Section 4.5). We then choose relevant attributes to detect possible anomalies, which are characterized by a subsequence in

the time series with an abrupt change in the values. For attributes that we cannot identify such a behavior, we exclude them from our analysis as they are likely to have an insignificant separation power. We quantify this behavior and call it a *potential power* of an attribute, denoted as $PP(\texttt{Attr}_i)$.

To calculate $PP(\texttt{Attr}_i)$, we first define a sliding window $w(\tau)$ as a subsequence of size τ in the time series. We also denote the median of $\texttt{Attr}_i$ as $\texttt{Median(Attr}_i\texttt{)}$ and denote the median of the values within a sliding window $w(\tau)$ as $\texttt{Median(Attr}_i\texttt{, } w\texttt{)}$. Then $PP(\texttt{Attr}_i)$ is calculated as follows:

$$PP(\texttt{Attr}_i) = \max_{w \in W} |\texttt{Median(Attr}_i\texttt{)} - \texttt{Median(Attr}_i\texttt{, } w\texttt{)}| \tag{4}$$

where W represents the set of all possible sliding windows of size τ. Equation (4) uses a median filter to calculate the maximum absolute difference between the overall median and the median of values in each window. We only include attributes with a potential power greater than a threshold $PP_t \in [0, 1]$. (DBSherlock uses $\tau = 20$ and $PP_t = 0.3$ as default values.)

We use DBSCAN to build clusters with the selected attributes from the previous step. DBSCAN takes two parameters, ϵ and $minPts$. For our algorithm, we fix $minPts$ to 3 and use the *k-dist* function to build a list L_k of the distances of the *k-th nearest neighbors*, as suggested in [25], to determine ϵ. We have empirically found $\epsilon = \max(L_k)/4$ to perform well in DBSherlock.

Given the clusters formed by DBSCAN, our algorithm returns the points in all clusters whose sizes are less than 20% of the total number of data items. This is under the assumption that the abnormal region is relatively smaller than the normal region. In Appendix E, we evaluate DBSherlock's accuracy when using automatic anomaly detection against a manually selected anomaly and another anomaly detection algorithm, PerfAugur [41].

8. EVALUATION

In this section, we empirically evaluate the effectiveness of DBSherlock. The goals of our experiments are to show that:

(i) Our causal models produce accurate explanations (Section 8.3).

(ii) Even without causal models, the raw predicates generated by DBSherlock are more accurate than those generated by the state-of-the-art explanation framework (Section 8.4).

(iii) Our idea of merging causal models improves the quality of our explanations significantly (Sections 8.5).

(iv) Incorporating domain knowledge allows DBSherlock to achieve higher accuracy (Section 8.6).

(v) DBSherlock is able to explain compound situations where multiple anomalies arise simultaneously (Section 8.7).

(vi) Using our predicates, users can diagnose the actual cause of performance anomalies much more accurately (Section 8.8).

We have included additional experiments in the Appendix for interested readers.

8.1 Experiment Setup

To collect log data with different types of anomalies, we ran different mixtures of the TPC-C benchmark [5] on Microsoft Azure [2] virtual machine instances. In all our experiments, we have used two Microsoft Azure A3-tier instances, each with 4 CPU cores of 2.1Ghz (AMD Opteron 4171H) and 7GB of RAM running Ubuntu 14.04. We employed one of the two A3 instances to run MySQL 5.6.20 and the other to simulate clients (using OLTPBenchmark framework [3, 21]). For stress-based experiments, we also used a tool called stress-ng [4] which can artificially stress the system by taking up excessive CPU, I/O and network resources when needed. Each individual experiment (called a *dataset* in this paper) consisted of two minutes of normal activity plus one or more abnormal situations (of varying length). We ran our experiments using both TPC-C and TPC-E. Due to space constraints and the similarity of the results, here we only report our TPC-C results and defer TPC-E results to Appendix A. The default setting used in our TPC-C workload was a scale factor of 500 (i.e., 50GB) with 128 terminals. We also experimented with different scale factors (from 16 to 500) and number of terminals (from 16 to 128). The results were consistent across these different settings, and thus we only report our results using the default setting described above. In each dataset, we intentionally created various abnormal situations on the server, as described next.

8.2 Test Cases

To test our algorithm, we created 10 different classes of anomalies to represent some of the important types of real-world problems that can deteriorate the performance of a database. During the two-minute run of the normal workload in each dataset, we invoked the actual cause of an anomaly with different start times and durations. For each type of anomaly, we collected 11 different datasets by varying the duration when possible (e.g., stressing system resources) or its start time (i.e., the time when the cause of an anomaly is triggered) when the actual duration was impossible to control (e.g., running *mysqldump*). The duration or start time of the anomalies ranged from 30 to 80 seconds with the increment of 5 seconds, yielding 11 datasets (i.e., 30, 35, $\cdots$, 80) for each type of anomaly (a total of 110 datasets). For each dataset, we manually selected a region of anomaly via visual inspection; the region left unselected automatically became the normal region.

Table 1 lists the types and descriptions of the different classes of anomalies that we tested within our experiments. These anomalies are designed to reflect a wide range of realistic scenarios that can negatively impact the performance of a transactional database.

8.3 Accuracy of Single Causal Models

Our goal in this section is to evaluate the effectiveness of our causal models in producing correct explanations. It is quite common that an anomaly from a certain cause is only observable a few times over the lifetime of a database operation. This makes log samples of such anomalies quite scarce in many cases (e.g., disk failure) and thus necessitates that our framework identifies the correct cause even when our causal model is created from a single dataset. Thus, in each test case we only used a single dataset to construct a causal model with θ=0.2 (which is the normalized difference threshold, see Section 4.5). This is the default value of θ in DBSherlock chosen to aggressively filter out attributes with insignificant behavior in the anomaly region. We applied the constructed causal model on all the remaining 109 datasets to obtain its confidence in each test case. We repeated this process until every dataset was chosen to construct a causal model.

With our algorithm, the correct causal models achieve the highest confidence in all 10 test cases (i.e., the correct cause was shown as the most likely cause to the user). The margin (i.e., positive difference) of confidence between the correct model and the highest among incorrect models is on average 13.5%. In other words, not only does the correct model achieve the highest confidence (and is shown to the user as the most likely cause), but its confidence is also well separated from the highest-ranked incorrect model. (In Section 8.5, we show that our model merging technique improves this margin even further.) Figure 7 shows the margin of confidence of

Type of anomaly	Description
Poorly Written Query	Execute a poorly written JOIN query, which would run efficiently if written properly.
Poor Physical Design	Create an unnecessary index on tables where mostly INSERT statements are executed.
Workload Spike	Greatly increase the rate of transactions and the number of clients simulated by OLTPBenchmark (128 additional terminals with transaction rate of 50,000).
I/O Saturation	Invoke stress-ng, which spawns multiple processes that spin on write()/unlink()/sync() system calls.
Database Backup	Run *mysqldump* on the TPC-C database instance to dump the table to the client machine over the network.
Table Restore	Dump the pre-dumped *history* table back into the database instance.
CPU Saturation	Invoke stress-ng, which spawns multiple processes calling poll() system calls to stress CPU resources.
Flush Log/Table	Flush all tables and logs by invoking *mysqladmin* commands ('flush-logs' and 'refresh').
Network Congestion	Simulate network congestion by adding an artificial 300-milliseconds delay to every traffic over the network via Linux's *tc* (Traffic Control) command.
Lock Contention	Change the transaction mix to execute *NewOrder* transactions only on a single warehouse and district.

Table 1: Ten types of performance anomalies used in our experiments.

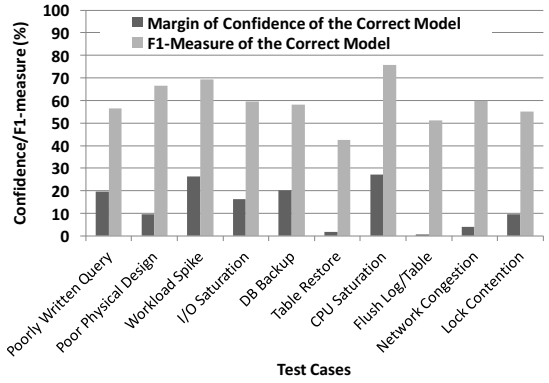

Figure 7: The margin of confidence and the average F1-measure of the correct causal model for different anomalies.

the correct causal model, which compares the average confidence of the correct causal model to the highest confidence among all other (incorrect) models for different types of anomalies.

Here, anomalies caused by 'Table Restore' and 'Flush Log/Table' proved most illusive, as they yielded the lowest confidence among the causal models. This was because the two anomalies shared the common characteristic that the DBMS performed too many disk I/Os. However, even in this case, `DBSherlock` could correctly distinguish the correct cause from the incorrect ones.

Overall, this challenging experiment is an extremely encouraging result showing that `DBSherlock` is capable of generating the correct explanation even with a single dataset as a training sample and in the presence of 9 other competing models.

8.4 DBSherlock Predicates versus PerfXplain

We compared the accuracy of our predicates with predicates generated by the state-of-the-art performance explanation framework, PerfXplain [34]. Since PerfXplain is designed to work with MapReduce logs, we had to re-implement PerfXplain's algorithm to fit into our context. Originally, PerfXplain operates on pairs of MapReduce jobs. Instead, we modified it to use pairs of our input tuples. We used the following query for PerfXplain:

`EXPECTED avg_latency_difference = insignificant`
`OBSERVED avg_latency_difference = significant`

where two average latencies are deemed significant if their difference is at least 50% of the smaller value. We chose the same parameters for PerfXplain as suggested in [34] (i.e., we used 2,000 samples and a weight value of 0.8 for its scoring rules). We also varied the number of predicates from 1 to 10 and chose 2, which yielded the best results for PerfXplain. With 11 datasets for each case, we used 10 datasets to generate predicates and the accuracy of generated predicates was tested on the remaining dataset. Figure 9 demonstrates the average precision, recall and F1-measure in comparison. Our predicates achieved better accuracy than PerfXplain in nearly all cases (except for recall on one test case). Most notably, `DBSherlock` improves on PerfXplain's F1-measure by upto 55% (28% on average). This shows that performance explanation and diagnosis for OLTP workloads requires drastically different techniques than those developed for OLAP and map-reduce workloads.

8.5 Effectiveness of Merged Causal Models

When multiple causal models are available (from diagnosing different datasets), `DBSherlock` tries to merge them as much as possible in order to further improve the relevance and accuracy of the generated explanations. To evaluate the effectiveness of our merging strategy, we conducted a series of experiments, each using multiple datasets as training samples. We randomly assigned about 50% of the datasets from each type of anomaly (i.e., 5 out of 11 datasets) to construct and merge causal models for each type. Merged causal models were then used to calculate the confidence on the remaining 6 datasets. The process was repeated 50 times, resulting in 300 instances of explanations for each test case. We used a lower value of θ, namely $\theta = 0.05$, for our merged causal models (in contrast to 0.20 used for our single causal models). With a lower value of θ, we can maximize the effect of merging causal models by having more predicates for each causal model at the start.

The results of these experiments are summarized in Figure 8a, showing that merging significantly increases the average margin of confidence against the second-highest confidence in all test cases.

To compare the accuracy of each explanation, we counted the number of cases where the correct cause was included in the top-k possible causes shown to a user. As shown in Figure 8b, `DBSherlock` presented the correct cause as the top explanation in almost every instance. In other words, k=1 would always be sufficient for achieving an accuracy greater than 98%. If we allow `DBSherlock` to list the top-2 possible explanations, then it identifies the correct cause in 99% of the cases.

We also studied the effectiveness of merging causal models with respect to the number of datasets used in constructing each causal model. As shown in Figure 8c, the accuracy of the merged causal models increases with more datasets. The accuracy quickly reaches 95% with only two datasets if only the top cause is returned, and it reaches 99% if the top two causes are returned. This experiment highlights that `DBSherlock` only requires a few manual diagnoses of an anomaly to construct highly accurate causal models.

In summary, the merging of causal models greatly improves the accuracy and quality of our explanations, generating predicates that

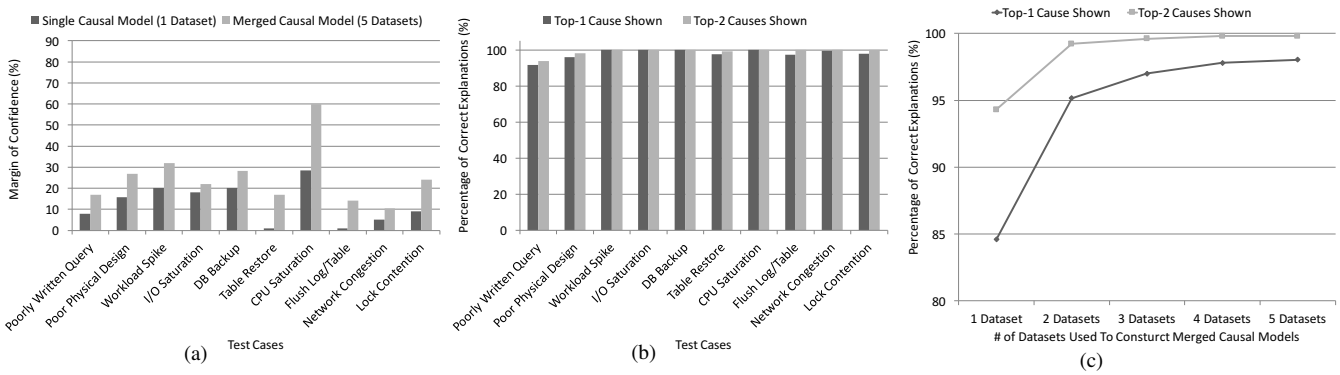

Figure 8: (a) The margin of confidence for single versus merged causal models, (b) the ratio of correct explanations for merged causal models (if the top-k possible causes are shown to the user), and (c) the effect of the number of datasets (i.e., number of manual diagnoses required) on the accuracy of the casual model.

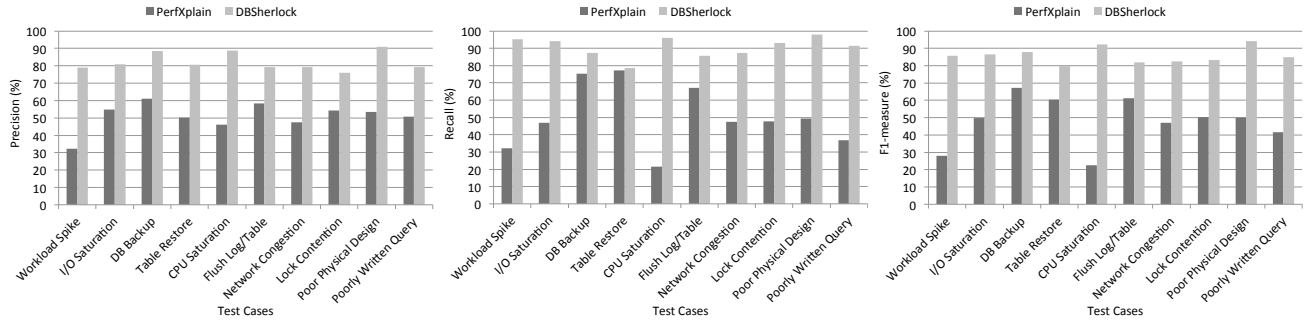

Figure 9: Average precision, recall and F1-measure of predicates generated by DBSherlock and PerfXplain.

are more relevant to the cause. Only a few datasets of the same anomaly are needed to construct a merged causal model that achieves an accuracy greater than 95%.

8.6 Effect of Incorporating Domain Knowledge

To study the effect of incorporating domain knowledge, we incorporated the four rules introduced in Section 5 into DBSherlock, and constructed single causal models with and without these rules, similar to the setup of Section 8.3.

	Accuracy if shown top-1 cause	**Accuracy if shown top-2 causes**
With Domain Knowledge	85.3%	94.8%
Without Domain Knowledge	82.7%	93.2%

Table 2: Ratio of correct causes with & without domain knowledge.

Table 2 reports the accuracy of single causal models with and without domain knowledge. Domain knowledge removed predicates that were a secondary symptom, and thus less relevant for the correct diagnosis of a given anomaly, improving the accuracy of causal models by 2.6% if shown the top-1 cause and 1.6% if shown the top-2 causes. On the other hand, this experiment shows that, in practice, DBSherlock works surprisingly well even without any domain knowledge. (To fully evaluate the effect of domain knowledge in removing secondary symptoms, we report additional experiments in Appendix F using synthetic data.)

8.7 Explaining Compound Situations

It is not uncommon in a transactional database that multiple problems occur simultaneously. These compound situations add a new level of difficulty to diagnostic systems. We ran an experiment to address the framework's capability in such compound situations. We created six cases, where two or three anomalies happen at the same time during the two-minute run of our normal workload. For this experiment, causal models were constructed for each individual test case by merging causal models from every dataset. Explanations were then generated for the compound test cases.

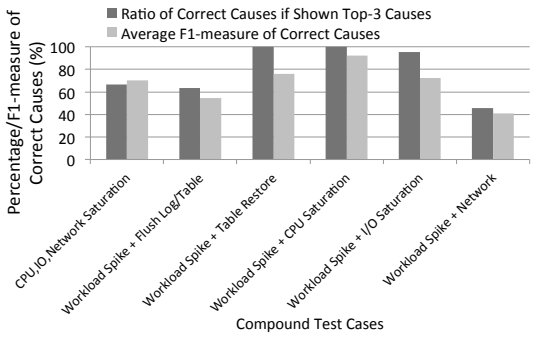

Figure 10: Ratio of the correct causes and their average F1-measure for compound situations (when top-3 causes are shown to the user).

Figure 10 demonstrates the ratio and the average F1-measure of correct causes (when three possible causes were offered to the user). On average, our explanation contained more than two-thirds of the correct causes, except for 'Workload Spike + Network Congestion'. For this dataset, DBSherlock missed the 'Workload Spike' and only returned 'Network Congestion' as the correct cause. This was because 'Network Congestion' had reduced the impact of 'Workload Spike' on the system (by slowing down the rate of incoming

Background	# of participants	Avg. # of correct answers (out of 10)
Baseline (No Predicates)	N/A	2.5
Preliminary DB Knowledge	20	7.5
DB Usage Experience	15	7.8
DB Research or DBA Experience	13	7.8

Table 3: The summary of the user study.

queries), and hence made it difficult for DBSherlock to identify their simultaneous occurrence.

8.8 User Study

We also performed a user study to evaluate the ability of our generated predicates in helping users diagnose the actual cause of performance problems. We asked various people who had some experience with databases to participate in a web-based questionnaire. We categorized the participants into three levels of competency based on their experience: *Preliminary DB Knowledge* (e.g., SQL knowledge or undergraduate course on databases), *DB Usage Experience*, and *DB Research or DBA Experience*. We used a few trivial questions to filter out spammers from genuine participants, leaving us with a total of 20 participants in our study. The questionnaire consisted of 10 multiple-choice questions. Each question had one correct cause and three randomly chosen incorrect causes. In each question, we presented a graph of average latency to our participants, with a pre-specified anomaly region and DBSherlock's generated predicates explaining the anomaly.

Table 3 shows the summary of the user study. Here, the first row represents the random baseline, i.e., where no predicates are presented to the user. Participants with preliminary database knowledge were able to identify the correct cause in 75% of the cases. Participants with practical database usage or above identified the correct cause in 78% of the cases. This promising result shows that the predicates generated by DBSherlock can help the end user correctly diagnose anomalies in practice.

9. RELATED WORK

Our work incorporates recent research in the fields of causality, performance diagnosis, and outlier detection.

Causality. Our work draws on the notion of causality proposed by Halpern and Pearl [28, 29]. We apply a simplified version of their causal model to introduce the notion of causality in our explanations. In the database literature, the notion of causality is brought together with data provenance [14, 17]. Meliou et al. [36] adapt the notion of causality to explain the cause of answers and non-answers in the output of database queries. In the context of probabilistic databases, Kanagal et al. [32] define the notion of an input tuple's *influence* on a query result. Scorpion [46] explains outliers in aggregate results of a query by unifying the concepts of *causality* and *influence*. Our notions of *normal* and *anomaly* are similar to Scorpion's *hold-out* and *outlier* sets, respectively.

Performance diagnosis. There have been many applications of performance diagnosis in databases [22], such as tuning query performance [8, 30], diagnosing databases that run on storage area networks [11], or parameter tuning [23]. Benoit [9] and Dias et al. [20] propose tools for automatic diagnosis of performance problems in commercial databases. However, [9] requires DBAs to provide a set of manual rules and [20] relies on detailed internal performance measurements from the DBMS (e.g., time spent in various modules of Oracle to process a query). More importantly, these tools lack explanatory features to answer 'why' a performance problem has occurred. In contrast, DBSherlock produces accurate explanations even without manual rules and using only aggregate statistics. Also, previous work has not accounted for the interaction of the DBMS with the machine that it is running on. DBSherlock gives an explanation based on every statistic it can gather both inside and outside the DBMS.

In the context of MapReduce, there is work on automatic tuning of MapReduce programs [7, 31]. Here, the most relevant work is PerfXplain [34], which generates predicate-based explanations. PerfXplain helps debug MapReduce jobs, answering questions such as 'Why Job A is faster than Job B?'. DBSherlock is designed for OLTP workloads, its predicates are more accurate than those of PerfXplain (see Section 8.4), and can incorporate causal models.

There has been much work on automated performance diagnosis in other areas. Gmach et al. [26] use a *fuzzy controller* to remedy exceptional situations in an enterprise application. Their controller, however, requires the rules to be hard-coded and pre-defined by experts. In contrast, DBSherlock allows for causal models to be added, merged, and refined as new anomalies occur in the future. Further, while DBSherlock allows for incorporating domain knowledge, it can provide accurate explanations even without domain knowledge (see Section 8.6). Mahimkar et al. [35] propose a tool for troubleshooting IPTV networks, but assume that large correlation and regression coefficients among pairs of attributes imply causality. DBSherlock does not make this assumption.

Outlier detection. DBSherlock uses a simple but effective outlier detection strategy to autonomously monitor database performance (Section 7). Allowing users to choose from additional outlier detection algorithms (e.g., [15, 19, 33, 40, 41, 42, 43, 44, 45, 46, 48]) will make an interesting future work.

10. CONCLUSION

Performance diagnosis of database workloads is one of the most challenging tasks DBAs face on a daily basis. Besides basic visualization and logging mechanisms, current databases provide little help in automating this adhoc, tedious, and error-prone task. In this paper, we presented DBSherlock, a framework that explains performance anomalies in the context of a complex OLTP environment. A user can select a region in a performance graph, which s/he thinks is abnormal, and ask DBSherlock to provide a diagnostic explanation for the observed anomaly. DBSherlock explains the anomaly in the form of predicates and possible causes produced by causal models. These explanations aid our users in diagnosing the correct cause of the performance problems more easily and more accurately. We also demonstrated that the confidence of our causal models can be increased via merging multiple causal models sharing the same cause. Our extensive experiments show that our algorithm is highly effective in identifying the correct explanations and is more accurate than the state-of-the art algorithm. As a much needed tool for coping with the increasing complexity of today's DBMS, DBSherlock is released as an open-source module in our workload management toolkit [1].

An important future work is to enable automatic actions for rectifying simple forms of performance anomaly (e.g., throttling certain tenants or triggering a migration), once they are detected and diagnosed with high confidence. We also plan to extend DBSherlock to go beyond creating causal models upon successful diagnoses, by documenting and storing the actions taken by the DBA to use as a suggestion for future occurrences of the same anomaly. Finally, DBSherlock's ideas might also be applicable to analytical workloads, e.g., in explaining performance problems caused by workload drifts [39].

Acknowledgements. This work is in part funded by National Science Foundation (IIS-1553169).

11. REFERENCES

[1] DBSeer. http://www.dbseer.org.

[2] Microsoft Azure. https://azure.microsoft.com/.

[3] OLTPBenchmark. http://oltpbenchmark.com/.

[4] stress-ng. http://tinyurl.com/pw59xs3.

[5] TPC-C benchmark. http://www.tpc.org/tpcc/.

[6] TPC-E benchmark. http://www.tpc.org/tpce/.

[7] S. Babu. Towards automatic optimization of mapreduce programs. In *SoCC*, 2010.

[8] P. Belknap, B. Dageville, K. Dias, and K. Yagoub. Self-tuning for sql performance in oracle database 11g. In *ICDE*, 2009.

[9] D. G. Benoit. Automatic diagnosis of performance problems in database management systems. In *ICAC*, 2005.

[10] K. A. Bollen. Structural equations with latent variables. 1989.

[11] N. Borisov, S. Uttamchandani, R. Routray, and A. Singh. Why did my query slow down. In *CIDR*, 2009.

[12] D. P. Brown, A. Richards, and D. Galeazzi. Teradata active system management, 2008.

[13] D. P. Brown and P. Sinclair. White paper: Real-time diagnostic tools for teradata's parallel query optimizer. Technical report, Teradata Solutions Group, 2000.

[14] P. Buneman, S. Khanna, and T. Wang-Chiew. Why and where: A characterization of data provenance. In *ICDT*. 2001.

[15] L. Cao, Q. Wang, and E. A. Rundensteiner. Interactive outlier exploration in big data streams. *PVLDB*, 7, 2014.

[16] S. Chen, A. Ailamaki, M. Athanassoulis, P. B. Gibbons, R. Johnson, I. Pandis, and R. Stoica. Tpc-e vs. tpc-c: Characterizing the new tpc-e benchmark via an i/o comparison study. *ACM SIGMOD Record*, 39(3):5–10, 2011.

[17] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 2007.

[18] T. M. Cover and J. A. Thomas. Entropy, relative entropy and mutual information. *Elements of Information Theory*, pages 12–49, 1991.

[19] A. Deligiannakis, V. Stoumpos, Y. Kotidis, V. Vassalos, and A. Delis. Outlier-aware data aggregation in sensor networks. In *ICDE*, 2008.

[20] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood. Automatic performance diagnosis & tuning in oracle. In *CIDR*, 2005.

[21] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[22] S. Duan, S. Babu, and K. Munagala. Fa: A system for automating failure diagnosis. In *ICDE*, 2009.

[23] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with ituned. *PVLDB*, 2, 2009.

[24] O. D. Duncan. Introduction to structural equation models. 1975.

[25] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[26] D. Gmach, S. Krompass, A. Scholz, M. Wimmer, and A. Kemper. Adaptive quality of service management for enterprise services. *ACM Transactions on the Web (TWEB)*, 2(1):8, 2008.

[27] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *JMLR*, 3, 2003.

[28] J. Y. Halpern and J. Pearl. Causes and explanations: a structural-model approach. part i: causes. In *UAI*, 2001.

[29] J. Y. Halpern and J. Pearl. Causes and explanations: a structural-model approach. part ii: explanations. In *IJCAI*, 2001.

[30] H. Herodotou and S. Babu. Xplus: a sql-tuning-aware query optimizer. *PVLDB*, 3, 2010.

[31] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for mapreduce programs. *PVLDB*, 4, 2011.

[32] B. Kanagal, J. Li, and A. Deshpande. Sensitivity analysis and explanations for robust query evaluation in probabilistic databases. In *SIGMOD*, 2011.

[33] E. Keogh, J. Lin, A. W. Fu, and H. Van Herle. Finding unusual medical time-series subsequences: Algorithms and applications. *Information Technology in Biomedicine, IEEE Transactions on*, 10(3):429–439, 2006.

[34] N. Khoussainova, M. Balazinska, and D. Suciu. PerfXplain: Debugging mapreduce job performance. *PVLDB*, 2012.

[35] A. A. Mahimkar, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and Q. Zhao. Towards automated performance diagnosis in a large IPTV network. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 231–242. ACM, 2009.

[36] A. Meliou, W. Gatterbauer, and D. Suciu. Bringing provenance to its full potential using causal reasoning. In *TaPP*, 2011.

[37] B. Mozafari, C. Curino, A. Jindal, and S. Madden. Performance and resource modeling in highly-concurrent OLTP workloads. In *SIGMOD*, 2013.

[38] B. Mozafari, C. Curino, and S. Madden. DBSeer: Resource and performance prediction for building a next generation database cloud. In *CIDR*, 2013.

[39] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. CliffGuard: A principled framework for finding robust database designs. In *SIGMOD*, 2015.

[40] A. Rogge-Solti and G. Kasneci. Temporal anomaly detection in business processes. In *Business Process Management*. 2014.

[41] S. Roy, A. C. König, I. Dvorkin, and M. Kumar. PerfAugur: Robust diagnostics for performance anomalies in cloud services. In *ICDE*, 2015.

[42] S. Sarawagi. Explaining differences in multidimensional aggregates. In *VLDB*, 1999.

[43] Y. Tan, X. Gu, and H. Wang. Adaptive system anomaly prediction for large-scale hosting infrastructures. In *PODC*, 2010.

[44] L. Wei, N. Kumar, V. N. Lolla, E. J. Keogh, S. Lonardi, and C. A. Ratanamahatana. Assumption-free anomaly detection in time series. In *SSDBM*, volume 5, pages 237–242, 2005.

[45] L. Wei, W. Qian, A. Zhou, W. Jin, and X. Jeffrey. Hot: Hypergraph-based outlier test for categorical data. In *AKDDM*. 2003.

[46] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 6, 2013.

[47] D. Y. Yoon, B. Mozafari, and D. P. Brown. DBSeer: Pain-free database administration through workload intelligence. *PVLDB*, 2015.

[48] J. X. Yu, W. Qian, H. Lu, and A. Zhou. Finding centric local outliers in categorical/numerical spaces. *KAIS*, 9, 2006.

APPENDIX

Our appendix provides supplementary experiments for interested readers. Appendix A demonstrates the performance of `DBSherlock` on a different workload (i.e., TPC-E). Appendix B shows the effectiveness of merged causal models and discusses the possible issue of over-fitting. Appendix C demonstrates the robustness of `DBSherlock` against input errors. Appendix D describes the effect of tuning parameters on our predicate generation algorithm. Appendix E evaluates `DBSherlock`'s performance, when using automatic anomaly detection. Appendix F demonstrates the effectiveness of domain knowledge in pruning secondary symptoms.

A. ACCURACY FOR OTHER WORKLOADS

To confirm whether `DBSherlock`'s capability of producing accurate explanations extends to other workloads besides TPC-C, we conducted additional experiments using the TPC-E benchmark [6]. Here, we used TPC-E with 3,000 customers, resulting in a 50GB data size. We generated the datasets and constructed merged causal models using a similar setup as in Section 8.5 (i.e., 5 datasets to construct merged causal models and 300 instances of explanations).

Type of Workload	**Accuracy if shown top-1 cause**	**Accuracy if shown top-2 causes**
TPC-C	98.0%	99.7%
TPC-E	92.5%	99.6%

Table 4: `DBSherlock`'s accuracy for TPC-C and TPC-E workloads.

Table 4 compares the percentage of correct answers when the top-1 or top-2 causes are returned to the user for both TPC-C and TPC-E. When only the top-1 cause was shown to the user, the accuracy for the TPC-E workload slightly dropped to 92.5% on average. This was mainly due to DBSherlock's lower accuracy for 'Poor Physical Design' with TPC-E, as the effects of 'Poor Physical Design' and 'Lock Contention' on the system were not as significant as they were with TPC-C. The was due to the fact that TPC-E is much more read-intensive than TPC-C [16]. Nonetheless, DBSherlock still achieved an impressive accuracy of 99% on average with the TPC-E workload when the top-2 causes were shown to the user.

B. OVER-FITTING AND MERGED CAUSAL MODELS

To verify if adding more datasets could further improve our merged models, we also ran a leave-one-out cross validation experiment. With 11 datasets for each case, we constructed causal models from 10 datasets and merged them. The final causal model then calculated confidence on the remaining dataset of each test case. Overall, the average confidence of the correct model increased slightly as shown in Figure 11a, but at the same time, the average margin of confidence decreased in some test cases as shown in Figure 11b.

The decrease in the average margin of confidence in some test cases suggests that merging more models than necessary can be ineffective. In other words, our proposed technique for merging causal models continues to widen the scope of relevant predicates while filtering irrelevant ones out. Once every irrelevant predicate has been filtered out, merging more models is not as effective. This is similar to the over-fitting phenomenon in machine learning. Nonetheless, DBSherlock still succeeded in returning the correct cause among its top two explanations in every instance (except for 'Network Congestion'), as demonstrated in Figure 11c.

C. RARE ANOMALIES AND ROBUSTNESS AGAINST INPUT ERRORS

As explained in Section 2.2, the user selects the abnormal regions manually after visual inspection of the performance plots. (We have used the same method in our experiments—Section 8.2.) However, users may not specify the regions with perfect precision. To understand how robust DBSherlock is to input errors caused by human mistakes, we conducted the following experiment. Using the same setup as Appendix B, we extended the boundaries of the original anomaly region by 10% in one experiment and shortened them by 10% in another. We also ran a third experiment where we randomly chose only two seconds of the original abnormal region as our input anomaly. The goal of this test case was to evaluate DBSherlock's effectiveness in diagnosing anomalies that are rare or only last a few seconds. For each dataset, we repeated each experiment 10 times and averaged the result.

Width of Abnormal Region	Accuracy if shown top-1 cause	Accuracy if shown top-2 causes
Original	94.6%	99.1%
10% Longer	95.5%	100%
10% Shorter	95.5%	97.3%
Two Seconds	74.6%	86.4%

Table 5: DBSherlock's robustness against rare and imperfect inputs.

Table 5 reports the percentage of correct answers when the top-1 or 2 causes are returned to the user. The accuracy did not change significantly when the span of the abnormal region was shorter or longer than the original one by 10%. More surprisingly, DBSherlock achieved reasonable accuracy even when the abnormal region was only two seconds long (e.g., the top-2 causes contained the correct explanation in 85% of the cases). These experiments show that DBSherlock remains effective even when the abnormal regions are not perfectly aligned with the actual anomaly or only last a very short period.

D. DIFFERENT PARAMETERS/STEPS IN DBSHERLOCK

We conducted various experiments to study the effect of the individual steps and configurable parameters of our predicate generation algorithm on its accuracy.

To evaluate the different steps of our algorithm (from Section 4), we compared it against its simpler variants by omitting some of the steps each time. Since steps 1, 2 and 5 (i.e., *Creating a Partition Space*, *Partition Labeling* and *Extracting Predicates*) form the skeleton of our algorithm and cannot be excluded easily, we omitted the other two steps (i.e., *Partition Filtering* and *Filling the Gaps*). Table 6 reports the average margin of confidence and accuracy of each variant. Skipping either of the *Partition Filtering* or *Filling the Gaps* steps lowers the accuracy of our algorithm significantly (down to 0–10%). Without both, our algorithm fails to find any relevant predicates for explaining the given anomaly. This experiment underlines the significant contribution of these two steps towards our algorithm's overall accuracy.

Algorithms	Overall avg. margin of confidence	Accuracy if shown top-1 cause
Original (all 5 steps)	37.4	94.6%
Without *Filling the Gaps*	9.3	10.1%
Without *Partition Filtering*	0.7	0%
Without *Filling the Gaps* & *Partition Filtering*	0	0%

Table 6: Contributions of the different steps of our predicate generation algorithm to the overall accuracy.

Our predicate generation algorithm has three configurable parameters: the number of partitions (R), the anomaly distance multiplier (δ) and the normalized difference threshold (θ). We conducted experiments to study the effect of these parameters on the generated explanations. We ran our algorithm on every dataset with different values of each parameter and averaged its confidence, computation time, and number of generated predicates. For each experiment, we used 10 datasets to construct merged causal models and calculated their confidence on the remaining dataset. We used the default values of $\{R, \delta, \theta\} = \{250, 10, 0.2\}$.

We varied R using the following values {125, 250, 500, 1000, 2000}. As shown in Figure 12a, $R >$1000 increased the computation time significantly, without much improvement in confidence. We also varied δ using the following values {0.1, 0.5, 1, 5, 10}. As shown in Figure 12b, and as expected, $\delta >$1 favored more specific predicates and led to higher confidence.

Lastly, we varied the value of θ using the following values {0.01, 0.05, 0.1, 0.2, 0.4}. As shown in Figure 12c, increasing the value of θ reduced the number of generated predicates but increased their confidence slightly. However, the confidence dropped significantly

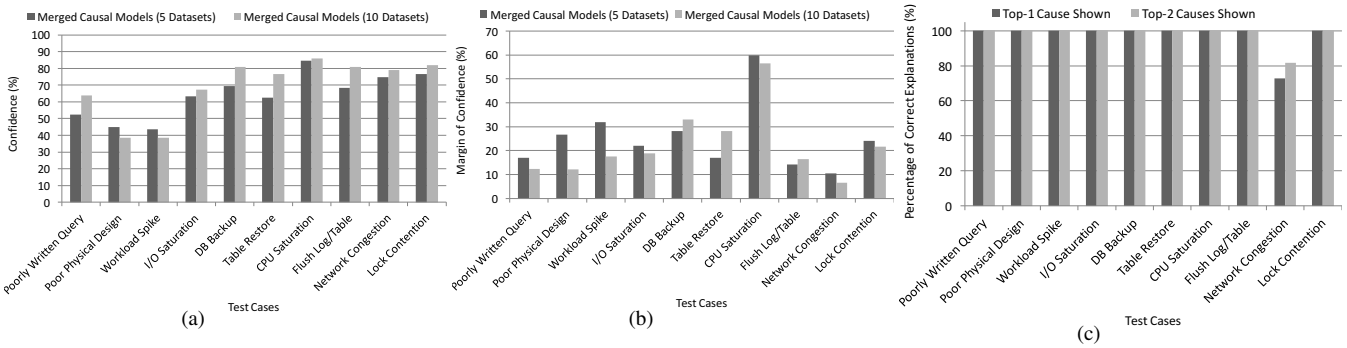

Figure 11: Evaluation of a merged causal model with 10 datasets in terms of (a) confidence, compared to a merged causal model with 5 datasets, (b) margin of confidence, compared to a merged causal model with 5 datasets, and (c) accuracy if top-k causes are returned.

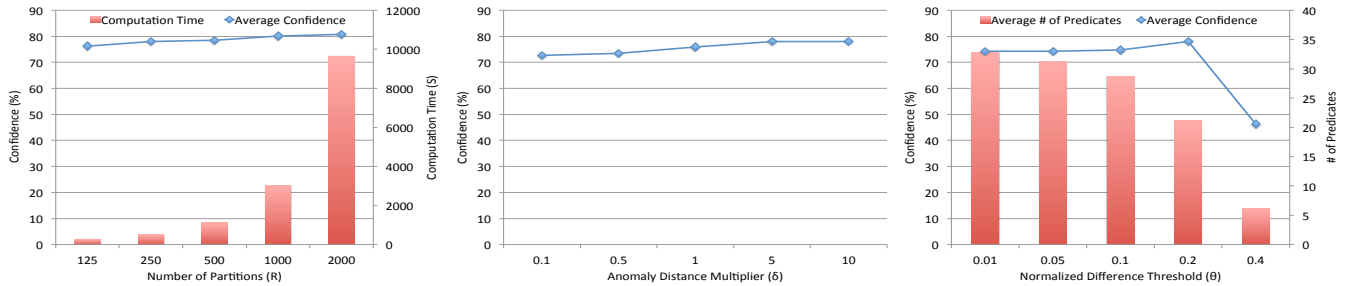

Figure 12: The effect of (a) the number of partitions on our algorithm's average confidence and computation time, (b) the anomaly distance multiplier on its confidence, and (c) the normalized difference threshold on its average confidence and number of generated predicates.

with $\theta = 0.4$. This is because a large value of θ filters out most predicates, leaving only a few predicates that are too specific to their training dataset and do not generalize to others.

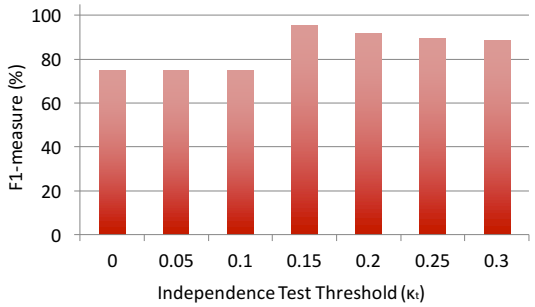

Figure 13: The effect of the parameter κ_t to average F1-measure.

We also performed a sensitivity analysis with the parameter κ_t that we use to filter secondary symptoms with the domain knowledge. We compared the average F1-measures with different values of κ_t using the same experiment setup as described in Appendix F. As demonstrated in Figure 13, the value of 0.15 for κ_t gave the highest average F1-measure.

E. DBSHERLOCK'S ACCURACY WITH AUTOMATIC ANOMALY DETECTION

We conducted an experiment to test DBSherlock's accuracy when automatic anomaly detection is used and also compared against another anomaly detection algorithm, PerfAugur. For PerfAugur, we supplied the overall average latency as its performance indicator variable and used their naïve algorithm with the original scoring function to compute the abnormal region.

For this experiment, we have generated datasets with the longer duration (i.e., 10 minutes) of the normal workload to ensure that the normal region is larger than the abnormal region, necessary for automatic detection algorithm to distinguish between both. Then, causal models were constructed for each individual test case by merging the causal models from 10 datasets. Abnormal regions were manually specified with our 'ground-truth' knowledge of each test case, to simulate a perfect user diagnosing the problem. The anomaly detection algorithm described in Section 7 and PerfAugur then identified abnormal regions for the remaining dataset of each test case. This leave-one-out cross validation set-up is designed to test the effectiveness of DBSherlock using merged causal models, but in the absence of any user input. DBSherlock used the automatically detected region of anomaly as its input and generated explanations for each test case.

Detection Algorithms	**Accuracy if shown top-1 cause**	**Accuracy if shown top-2 causes**
Manual Anomaly Detection	94.6%	99.1%
Automatic Anomaly Detection	90%	95.5%
PerfAugur	77.3%	88.2%

Table 7: Ratio of the correct causes for different strategies.

As shown in Table 7, DBSherlock identified about 95% of the correct causes on average with our algorithm, when top-2 possi-

		Actual	
		Positive	**Negative**
Domain Knowledge Test	**Pruned**	91.6%	0.9%
	Not Pruned	8.4%	99.1%

Table 8: Confusion matrix for finding secondary symptoms using incorporated domain knowledge.

ble causes were shown to a user. Our anomaly detection algorithm also performed substantially better than PerfAugur's detection algorithm. This promising result demonstrates that DBSherlock can be used in an automated setting once it has enough user feedback for well-constructed causal models. An interesting future work is to integrate a domain-specific and a more sophisticated anomaly detection algorithm in DBSherlock.

F. TESTING DOMAIN KNOWLEDGE WITH SYNTHETIC DATASET

To test DBSherlock's ability to remove secondary symptoms, we conducted a separate experiment using a synthetic dataset. Using a real dataset for this experiment is non-trivial since the 'ground-truth' casual model is unknown to us. With a synthetic dataset, however, we can define the 'ground-truth' causality between variables and hence, correctly evaluate our framework's ability to prune secondary symptoms. We created the synthetic dataset using a randomly generated *linear causal graph*. A linear causal graph is a directed acyclic graph, where the causal relationship of variables in the graph is defined by a linear structural equation model (SEM) [10, 24]. Using this graph, we randomly generated the domain knowledge to classify which predicates to prune (called **'Actual Positive'**) or keep (called **'Actual Negative'**) before running the experiment.

Now, we give a detailed description of our experimental setup. Our experiment consisted of multiple runs, where we randomly generated synthetic datasets. This bootstrap testing model enables us to empirically evaluate DBSherlock's ability to incorporate domain knowledge in various scenarios. For each run, we generate a linear causal graph $G = \{V_1, V_2, \cdots, V_k\}$, where k is the number of variables in the system. We used $k = 7$. Each node V_i in G represents a variable and corresponds to an attribute $\texttt{Attr}_i$ in our input data. Note that we treat the node V_k specially as an *effect variable*, which is the node with no outgoing edges and at least one incoming edge. From the effect variable V_k, we also define *root cause variables*, denoted as C. Root cause variables are the root nodes of the effect variable V_k (i.e., the ancestors of V_k with no incoming edges). They are the variables causing anomalies. For these variables, we randomly draw their values from $\mathcal{N}(10, 10)$ except 10% of their values are drawn from $\mathcal{N}(100, 10)$, which are contiguous and aligned among the root cause variables, representing an abnormal region of the data.

A variable V_i that is not a root cause variable is defined by a linear structural equation of the form:

$$V_i = \sum_j c_{ji} V_j + \epsilon_i \tag{5}$$

where c_{ji} is a *cause coefficient*, which represents the effect of V_j on V_i and ϵ_i is an error term for the variable V_i. c_{ji} takes a non-zero integer value randomly drawn in [-10, 10]. ϵ_i is drawn from the standard normal distribution (i.e., $\mathcal{N}(0, 1)$). Starting from the root cause variables and applying linear structural equations in order, we generate a synthetic dataset for DBSherlock in the format described in Section 2.1 (i.e., $(\texttt{Timestamp}, \texttt{Attr}_1, \ldots, \texttt{Attr}_k)$). The synthetic dataset we used for the experiment had 600 tuples, representing data collected for 10 minutes with 1-second intervals. 60 consecutive tuples are selected as an abnormal region (i.e., their root cause variables have the distribution of $\mathcal{N}(100, 10)$), representing the anomaly that lasts for a minute.

Now given the root cause variables C, we construct the domain knowledge D by randomly generating rules for the attributes corresponding to the root cause variables, each attribute becoming the cause variable of a rule. Random attributes then become effect attributes to create multiple rules obeying the two conditions of the rule in Section 5. We consult C, D and G to classify which predicates should be pruned (**'Actual Positive'**) or not (**'Actual Negative'**) before running the experiment. A predicate should be pruned only if its attribute is an effect attribute in D and there exists a path to it from its cause variable in G. They are secondary symptoms that we should remove using our domain knowledge. Conversely, a predicate should not be pruned only if its attribute is an effect attribute in D and there exists no path to it from its cause variable in G.

We studied the impact of incorporating domain knowledge on 10,000 randomly generated linear causal graphs (i.e., 10,000 runs). We evaluated the performance of our technique from Section 5 using the confusion matrix shown in Table 8. Our technique for pruning secondary symptoms achieves a 91.6% precision and a 99.1% recall. This experiment demonstrates the effectiveness of our technique in correctly pruning secondary symptoms, as well as its robustness against incorrect domain knowledge (thanks to checking the independence between attributes in the data itself).