ConfSeer: Leveraging Customer Support Knowledge Bases for Automated Misconfiguration Detection

Rahul Potharaju†, Joseph Chan†, Luhui Hu†, Cristina Nita-Rotaru∗,
Mingshi Wang†, Liyuan Zhang†, Navendu Jain‡
†Microsoft ∗Purdue University ‡Microsoft Research

ABSTRACT

We introduce ConfSeer, an automated system that detects potential configuration issues or deviations from identified best practices by leveraging a knowledge base (KB) of technical solutions. The intuition is that these KB articles describe the configuration problems and their fixes so if the system can *accurately* understand them, it can *automatically* pinpoint both the errors and their resolution. Unfortunately, finding an accurate match is difficult because (a) the KB articles are written in natural language text, and (b) configuration files typically contain a large number of parameters with a high value range. Thus, expert-driven manual troubleshooting is not scalable.

While there are several state-of-the-art techniques proposed for individual tasks such as keyword matching, concept determination and entity resolution, none offer a practical *end-to-end* solution to detect problems in machine configurations. In this paper, we describe our experiences building ConfSeer using a novel combinations of ideas from natural language processing, information retrieval and interactive learning. ConfSeer powers the recommendation engine behind Microsoft Operations Management Suite that proposes fixes for software configuration errors. The system has been running in production for about a year to proactively find misconfigurations on tens of thousands of servers. Our evaluation of ConfSeer against an expert-defined rule-based commercial system, an expert survey and web search engines shows that it achieves 80%-97.5% accuracy and incurs low runtime overheads.

1. INTRODUCTION

Configuration errors have a significant impact on system performance and availability. For instance, a misconfiguration in the user-authentication system caused login problems for several Google services including Gmail and Drive [47]. A software misconfiguration in Windows Azure caused a 2.5 hour outage in 2012 [65]. Many configuration errors are due to faulty patches (e.g., changed file paths causing incompatibility with other applications, empty fields in Registry), failed uninstallations (e.g., dangling file references), and manual fixes by users, risking compromise or loss of data e.g., reducing the number of backup copies.

Unfortunately, troubleshooting misconfigurations is time-consuming, hard and expensive. First, today's software configurations are becoming increasingly complex and large comprising hundreds of parameters and their settings [76]. Thus, given a large install base of diverse applications with numerous third party packages, it becomes difficult to specify the 'perfect configuration state' or ask developers to manually specify the correct value of each parameter. Second, many of these errors manifest as silent failures leaving users clueless: they either search online or contact customer service and support (CSS) which can cause loss of productivity, time and effort due to manual triage. Further, the CSS contracts incur a high cost of ownership of up to tens of millions of dollars annually [60]. Prior studies have reported that technical support contributes 17% of the total cost of ownership of desktop PCs [34].

To address these challenges, several research efforts have proposed techniques to identify [31, 67], diagnose [76, 16, 61, 50, 64, 74, 77, 75], and fix configuration problems [13, 62]. Some commercial tools are also available to manage system configurations or to automate certain configuration tasks [21, 28, 22]. However, many of these approaches either assume the presence of a large set of configurations to apply statistical testing (e.g., PeerPressure [61]); periodically checkpoint disk state (e.g., Chronus [64]) risking high overheads; use data flow analysis (e.g., ConfAid [16]) for error-tracing; use a speculative OS kernel to try out fixes (e.g., AutoBash [57]); or inject specific types of configuration errors (e.g., ConfErr [35]) which may not be feasible in production settings, risk incompleteness in covering diverse misconfigurations, or make it the users' responsibility to identify the misconfigured machine.

1.1 Our Contributions

This paper presents the design, implementation and evaluation of ConfSeer, a system that aims to proactively find misconfigurations on user machines using a knowledge base (KB) of technical solutions. Specifically, ConfSeer focuses on addressing parameter-related misconfigurations, as they account for a majority of user configuration errors [73]. Our key idea

Proceedings of the VLDB Endowment, Vol. 8, No. 12

Type of Error (A)	Capability (B)	<parameter name, value> [Software](C)	KB Id (D)	Matched Text from KB articles (in bold) (E)
Value constraint	Event ID-based exact matching	Message 823 [SQL Server 2005]	2015755	How to troubleshoot a **Msg 823** error in SQL Server
Range constraint	Proximity-based matching	NumOfLogicalProcessors 8 [SQL Server 2008]	2154845	... as a general rule, **if the number of logical processors is less than or equal to 8**, use the same number of data files as logical processors.
Range constraint	Constraint evaluation on well-formed text containing numerics	TcpMaxDataRetransmissions 10 [Windows Server 2008]	170359	... segment is retransmitted, up to the **TcpMaxDataRetransmissions** value. The **default value for this parameter is 5**.
Range constraint	Proximity-based matching	SynAttackProtect 0 [Windows Server 2003]	324270	Value name: **SynAttackProtect** ... **0** (default value): No SYN attack protection. **1: Set SynAttackProtect to 1 for better protection against SYN attacks.**
Value constraint	Proximity-based matching	FileGrowth 4 GB [SQL Server 2008]	2633151	The **file growth that is configured for the SQL Server database transaction log file is 4 gigabytes (GB)** or multiples thereof (for example, 8 GB, 12 GB, and so on).
Value constraint	Abbreviation detection with proximity-based approximate matching	ProxyFQDNEnabled true [Lync Server 2013]	2743783	Lync Server hosting provider **ProxyFqdn** value causes push notifications to fail
Value constraint	Abbreviation detection with proximity-based approximate matching	LSACrashOnAuditFailFlag true [IIS 6.0]	832981	2 Start Registry Editor 3 Locate the following key, and then **set the value of this key to 1** HKEY_LOCAL.../Control/**Lsa/CrashOnAuditFail**...
Range constraint	Constraint evaluation containing numerics	NumOfLogicalProcessors 100 [Windows Server 2008]	2510206	Performance issues **when more than 64 logical processors are used** in Windows Server 2008 R2
Value constraint	Constraint evaluation on free-form text	NumberOfDatabaseCopies 2 [Exchange Server 2010]	2617815	The database has not been backed up within the recommended timeframe, and **there are less than three database copies**.
Range constraint	Constraint evaluation with proximity matching	CurrentIORate 1000 MB per second [SQL Server 2012]	920093	... **High I/O rates**, for example, more than 10,000 physical I/O per second or **more than 500 megabytes (MB) per second**
Range constraint	Constraint evaluation containing numerics	NumMailboxThreads 10 [Exchange Server 2010]	2603736	There are several types of events for which the Exchange store tags a mailbox as a potential threat: • If **there are more than five threads in that mailbox** that haven't made progress for a long time

Table 1: Real-world examples of misconfigurations (column C) in commercial software, the matched KB article ID (column D) along-with its matching natural language text (column E) output by ConfSeer.

behind ConfSeer is to enable *configuration-diagnosis-as-a-service* by automatically matching configuration problems to their solutions described in free-form text (e.g., KB articles) and pinpointing them to users for proactive resolution.

The main idea is as follows: First, *ConfSeer* takes the snapshots of configuration files from a user machine as input. These are typically uploaded by agents running on these machines. Second, it extracts the configuration parameter names and value settings from the snapshots and matches them against a large set of KB articles, which are published and actively maintained by many vendors. Third, after a match is found, ConfSeer automatically pinpoints the configuration error with its matching KB article so users can apply the suggested fix.

ConfSeer builds on three key ideas: (1) semantic matching, (2) separating the matching mechanism from policy-based ranking, and (3) interactive learning to improve accuracy. First, since KB articles are written in free-form text, it processes them into a canonical representation, builds features for matching, and then constructs an inverted index to perform fast lookups of feature values across KB articles; the index can be updated periodically or on-demand as new articles are added. Then given a list of <parameter name, value> pairs from an input configuration snapshot, ConfSeer performs feature matching in parallel based on (a) exact keywords, (b) proximity, (c) synonym/abbreviation of parameter names, and (d) constraints on parameter settings e.g., data type, format, and value range. We also propose a new technique that does constraint evaluation on text using Abstract-Syntax-Trees that has not been proposed earlier to our knowledge. Second, it aggregates the accuracy scores of KB articles matched based on individual <parameter name, value> pairs and then computes their final ranking for the entire snapshot based on specified policies. This separation allows applications the flexibility to use any desired technique to rank matching articles e.g., using KB article popularity, expert guidelines or customer feedback. To improve result accuracy, ConfSeer relies on incremental machine learning. Table 1 shows real-world examples of the matching KB articles output by ConfSeer for different types of misconfigurations in a commercial software system.

We evaluate ConfSeer against the previous release of Microsoft Operations Management Suite (OMS) [7] which used about 200 manual rules to match misconfigurations to KB articles. ConfSeer is able to automatically infer 97.5% of the configuration errors covered by these manual rules. Further, its learning component exhibits a high F-score of 0.84. An expert evaluation and a user survey on diagnosing misconfigurations against an expanded KB article set show that ConfSeer achieves 85%-97.5% accuracy in baseline comparison to popular search engines (69.5% for Bing and 65.5% for Google) that are based on state-of-the-art IR, NLP and ML techniques, and that it incurs low runtime overheads. The OMS system has been running ConfSeer for over a year to automatically

analyze misconfigurations on tens of thousands of machines.

ConfSeer is the first approach that combines traditional IR and NLP techniques (e.g., indexing, synonyms) with new domain specific techniques (e.g., constraint evaluation, synonym expansion with named-entity resolution) to build an end-to-end practical system to detect misconfigurations. It is part of a larger system-building effort, SysSieve [11], to automatically detect software errors and misconfigurations by leveraging a broad range of data sources such as knowledge bases, technical help articles [44], and question and answer forums [10], which contain valuable yet unstructured information to perform diagnosis.

2. CUSTOMER SERVICE AND SUPPORT

Given a configuration problem, users typically first query a search engine (e.g., using the error message as keywords) for resolution. These engines aim to search the relevant documents in their index built from crawled web documents, discussion forums/e-mail threads, software manuals, or other public documentation of the software. If that fails, they contact a CSS center (e.g., via email, IM, phone call) where a support engineer tries to search the reported error or problem description in their KB database or browse through a classification scheme e.g., based on software or error types. These KB databases provide access to expert-written facts and rules to solve problems which can be directly provided to the customer. Further, since these articles are written by experts, any errors are likely to be corrected quickly. Many software vendors maintain KBs for customer support e.g., desk.com [2], VMWare [12], Oracle [9], IBM [6], EMC [3], Google [4], Apple [1], and Microsoft [8] having more than 250k+ articles [5].

If the support engineers observe a recurring problem across multiple complaints or error reports (e.g., [45]), they would document a solution in form of a KB article so that it can be directed as a response to customers. These technical solutions also get notified to subject matter experts (SMEs) on different software applications. These SMEs in turn may author (1) code logic (e.g., XML files, shell scripts) to evaluate configuration settings on a user machine, and (2) a rule to link the outcome of the script with the matching KB article.

Drawbacks. While the human-guided triage in CSS provides personal attention to customers, it has several drawbacks. First, it incurs a significant cost which increases linearly with the number of customer calls and call duration. Second, it risks long wait times particularly during periods of high call volume e.g., new releases or problems due to faulty bug patches. Third, it is reactive in nature – the fix is applied *after* the problem has occurred e.g., network disconnection. Finally, this troubleshooting process is dependent on the SME-defined rules which risk incompleteness for large KB databases or risk becoming outdated as the software evolves.

3. ConfSeer OVERVIEW

Figure 1 shows an overview of ConfSeer. ConfSeer operates in three phases described below (see Figure 2).

Offline Index Building Phase: The goal of this phase is to parse (§4.1) each KB document to build an intermediate canonical representation that is analyzed through a pipeline of filters

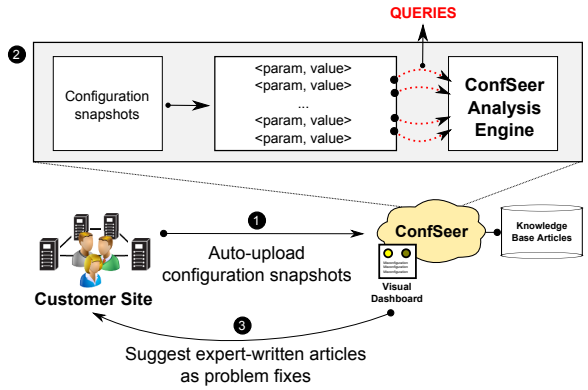

Figure 1: Overview of ConfSeer: (1) The configuration snapshots from customer machines are given as input. (2) A semantic matching of the configuration parameter names and value settings from the snapshots against the inverted index of KB articles is performed. (3) Matched articles along with the corresponding machine information are displayed to the user.

(§4.2) to extract features for indexing and querying. Finally, an inverted index is built on top to enable fast matching.

Online Query Processing Phase: The goal of this phase is to leverage the inverted index previously built to automatically retrieve KBs describing solutions to configuration errors in a given configuration snapshot. This is achieved in three steps. First, the configuration snapshot is parsed into a set of multiple independent key-value type queries. Second, each query is converted into an abstract form through a Query Processor (§5.1) and distributed to multiple matchers (§5.2). Finally, the results from each matcher are sent to a ranking engine (§5.3) which ranks the KB articles in order of relevance.

Interactive Learning Phase: This phase leverages human guidance to improve accuracy and ranking of results. In ConfSeer, learning happens in two stages. First, a classifier is built based on a training set of expert-labeled KB articles and then it is used to filter the relevant results. Second, an interactive interface is provided to get user feedback for incremental learning.

4. OFFLINE INDEX BUILDING PHASE

Building an index that serves a multitude of queries (e.g., free-form, fuzzy matching) requires addressing three key questions. First, what type of features to extract from the raw KBs to enable search? Second, how do we extract this information in a scalable manner from a large input corpus? Third, how to perform constraint evaluation on free-form text?

4.1 Document Parsing

The first step is to pre-process the KB documents into a form suitable for indexing. ConfSeer implements a variety of parsers for several popular document formats like XML, HTML, CSV, and plain text. Each input document is parsed into an intermediate representation which the indexer can process further or store directly. However, a key challenge is that simply stripping the meta-data tags from a document will likely lose the semantic context of the neighboring text e.g., flattening an HTML table will not associate the header column information with each of the row values. To address this challenge, ConfSeer *flattens* an input file as follows. Regular text

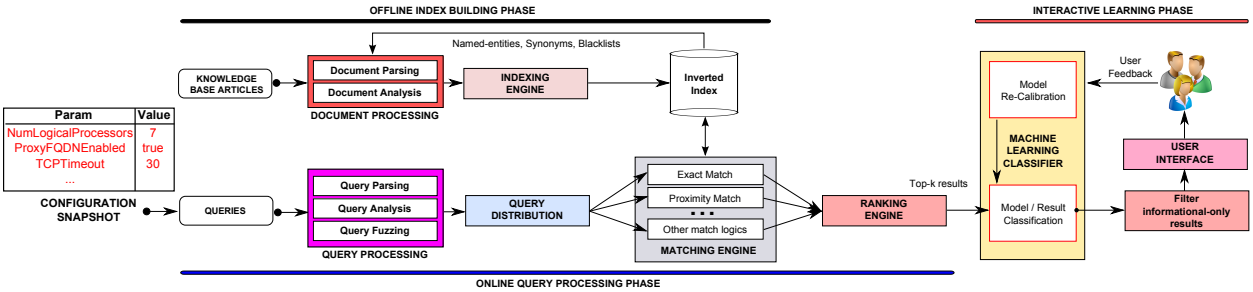

Figure 2: ConfSeer operates in three phases: (1) Offline Index Building, (2) Online Query Processing, and (3) Interactive Learning.

i.e., text inside paragraph tags (e.g., <p>,) is converted as is. To preserve the semantic information in tables, ConfSeer replicates the information contained in the header for each row by transforming the hierarchical document structure into its corresponding JSON format [24].

4.2 Document Analysis

After parsing the KB articles, our next goal is to extract their content for indexing. One pre-requisite for building an index is that the free-form text inside KBs be converted into a fine-grained representation called terms which can be indexed. For instance, the sentence *"Check if server has malfunctioning network card"* can be indexed as a sequence of terms: 'Check', 'server', and so on. However, this poses the fundamental question of selecting what terms to index, which we describe next in form of four specific challenges:

C1. Contextual Information: Intuitively, words that would be most useful to build an index should capture information related to the parameter names and values in configuration snapshots i.e., domain-specific information and any contextual information to facilitate accurate matching. For instance, in the sentence "*set the TcpTimedWaitDelay setting to 30*", it is important to capture "*TcpTimedWaitDelay*" along with its corresponding parameter value "*30*".

C2. Technical Words: In KB articles, it is common to see camel case words e.g., GetMethod, IsSecurityLogOverwrite. Thus, a query to find "security log" using these unmodified keywords may not yield the expected result.

C3. Constraint Evaluation: Given a query such as "*NumOfLogicalProcessors 6*", it is non-trivial to retrieve matching text such as "*the number of logical processors should be less than or equal to 8*" which logically applies to the query. One strawman is to enumerate numerical values e.g., replace "*less than or equal to 8*" with the set $[1, 8]$ so that a search for "6" will retrieve this text. However, this simple heuristic will be expensive for large value ranges or be inaccurate to handle different value types e.g., fractions, enumerations.

C4. Synonym Search: Given a query such as "sql" or "sql server", it may be necessary to fetch all technical articles containing the phrase "sql server database" or "sql engine". There are two techniques to achieve this: (1) expand the query to include the *synonyms* called Query-time Synonym Expansion, or (2) expand the text during indexing to include all the synonyms of a token(s) called Index-time Synonym Expansion. However, two challenges arise in using them for synonym search:

• **Query-time vs. index-time expansion:** Deciding between the two techniques requires managing a typical space-time trade-off. In particular, the former consumes less space for the index but it will increase query latency as multiple queries (one query per synonym) need to be executed, while the latter decreases query latency at the cost of higher space overheads of indexing as the tokens and their synonyms need to be stored.

• **When to perform synonym expansion?** For instance, performing a naïve synonym expansion of the token "SQL" in the sentence "*Install SQL Server*" will yield "*Install [SQL,SQL Server] Server*" which is incorrect. Our key idea to solve this problem is to first detect that "SQL Server" is a named-entity and ensure that the synonym expansion happens on the right candidate i.e., on "SQL Server" instead of "SQL".

To address **C1**, ConfSeer adopts the vector-space model to represent documents as bag-of-words and stores them in an inverted index [54] comprising the individual terms and the documents they appear in as well as the position offset to enable proximity-search queries (e.g., term *x near* term *y*). We next describe how ConfSeer addresses the **C[2-4]** challenges (see Figure 3 for an overview).

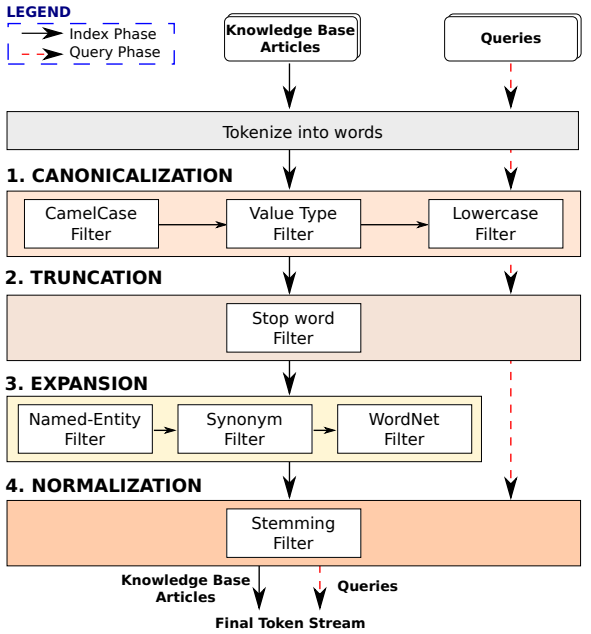

Figure 3: A pipeline of processing filters to extract tokens from KB articles for building an inverted index.

4.2.1 Constraint-based Search

ConfSeer supports queries on technical words (**C2**) and constraint evaluation (e.g., NumProcessors ≤ 8) on free-form text (**C3**), through *Canonicalization* via three sequential filters:

F1. CamelCase Filter: To enable free-form search of camelcase words, ConfSeer performs camelcase expansion i.e., expanding them into their constituent words. We consider the following five common cases obtained through an empirical analysis of the KB articles: (1) the standard case 'CamelCase' is often used for type names and references in source code or configuration entities e.g., GetMethodName; (2) the interior case 'camelCase' is often used for identifier names e.g., methodName; (3) capital letters as suffix of the word 'CamelCASE' e.g., GetBIT, (4) all capital letters 'CAMELCASE' are often used in abbreviations or boolean bits e.g., ENABLED; and (5) a mixture of camel case and capital letters are often used to include abbreviations inside the word e.g., WeakRSAKeys. Given an input camel case word, this filter outputs the original token along with its constituent words. For instance, 'IsSecurityLogOverwrite' will be converted into [IsSecurityLogOverwrite, Is, Security, Log, Overwrite].

F2. Value Type Filter: To achieve constraint evaluation on free-form text, we propose a simple yet novel index-time optimization: add hints regarding the value type to the index (e.g., NUMERIC for 8, BOOLEAN for true) with the same token position as the value. Therefore, during tokenization, this filter will detect the type of the token (e.g., numerics such as 1,2,3, booleans such as true/false, version numbers such as 1.0.0.1) and output a replacement token with the same position as the input token (see Box 1). Note that the position for both 8/NUMERIC and true/BOOLEAN remains same in the index i.e., 15 and 21, respectively. §5.2 describes how we leverage these index-time value hints during query processing.

> **Original**: In SQL, if the number of logical processors is less than or equal to 8, set the InMemory bit to true $\cdots$.
> **Index-time hints**: In SQL, if the number of logical processors is less than or equal to 8 ***NUMERIC***, set the InMemory bit to true ***BOOLEAN*** $\cdots$.

Box 1: Example showing modification by the Value Type Filter.

F3. Lowercase Filter: This filter outputs a lowercased version of the input token. Note that the Lowercase filter should only be used after applying the CamelCase filter otherwise the property of camelcased words would not be preserved and hence expansion becomes infeasible.

4.2.2 Synonym Search

We perform synonym search (**C4**) in three stages. First, commonly used words or phrases are filtered out through a process called *Truncation* to reduce the number of candidates for synonym expansion. Second, we perform *Expansion* in which we first recognize named-entities and subsequently perform synonym expansion through two dictionaries — one that we built from the KB articles in our dataset and another using WordNet [46]. Finally, we perform *Normalization* to reduce words into their root forms — this ensures that all word forms are equivalent (e.g., upgraded, upgrading, upgrade are all reduced to their root form 'upgrad').

F4. Stop word Filter: To filter noise words, we utilize a stop list of the most frequently occurring words in English, or a negative dictionary. This dictionary includes words such as 'a', 'the', 'and', 'my', and 'I'.

F5. Named-Entity (NE) Filter: The aim of this filter is to identify NEs as a precursor for the next step of expanding synonyms. For instance, in the sentence "*Install SQL Server in your machine*", it is necessary to recognize that "SQL Server" is an NE and thus a candidate for synonym expansion. Once the filter recognizes the set of tokens to be grouped as an NE, it adjusts the positional information of all tokens in that group to the position of the first token in the NE. This adjustment will allow us to identify NEs in subsequent stages.

ConfSeer takes a three step approach to recognize NEs:

• **Obtaining the list of NEs**: To automatically obtain the list of NEs from the input corpus, we find the most frequently occurring phrases and then use RIDF [23] to identify the important ones — phrases with high RIDF are domain-specific and have distributions that cannot be attributed to a chance occurrence.

• **Data structures for NE Lookup**: Given the list of named-entities, the NE filter builds a graph as a pre-processing step to enable fast lookups. In this graph, nodes denote words and edges connect words if they occur together in a phrase in the NE list. For instance, given the words [sql, sql server, sql server database, sql server 2005], the final graph will contain four nodes (sql, server, database, 2005), an edge from sql $\rightarrow$ server, server $\rightarrow$ database, and server $\rightarrow$ 2005. Next, we leverage this graph to perform index-time NE recognition.

• **Index-time NE Recognition**: We perform NE recognition during indexing. For each input token we check if the graph consists of a node having the token string as its value. If it does, a check is made to see if the next token exists as a neighbor of the token string in the graph and concatenated with the previous token and pushed onto a stack. This process is continued as long as the next token exists as a neighbor of the previous token. Once the condition is violated, the stack is popped and the concatenated string will be given the same position as the first entry in the stack.

F6 and F7. Synonym/WordNet Filter: ConfSeer enables synonym retrieval by performing synonym expansion during index-time to reduce query latency. Further, since the synonyms for technical words change rarely (e.g., till a new product is released), storing them in an index is reasonable. The Synonym Filter works similarly to the NE Filter with one exception – upon finding a token with synonyms, it injects all synonyms as new tokens into the stream and sets their positional value to be the same as that of the original token.

F8. Stemming Filter: In many cases, the semantic of the word matters more than the word itself. For instance, a technical article describing "upgrading your software" and "upgradation of a software" most likely are about the same concept *upgrade*. Therefore, it is useful to convert words into their root form. For this, we leverage the standard Porter stemming algorithm [48] which is a process for removing the commoner morphological and inflexional endings from words (e.g., connection, connected, connecting $\rightarrow$ connect) in English.

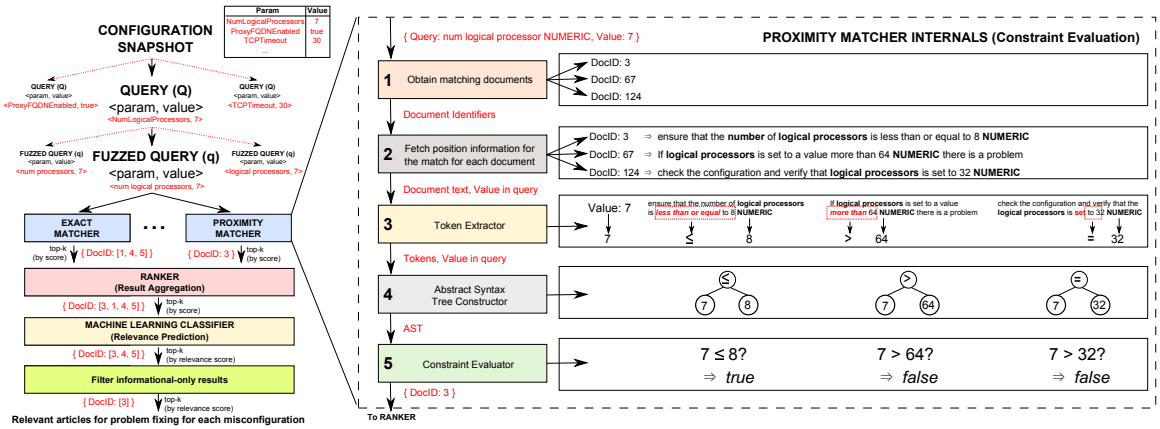

Figure 4: End-to-end query processing pipeline in ConfSeer: Given a configuration snapshot, each (param, value) pair is transformed to a set of queries which are distributed to a variety of feature matchers running in parallel. The scores of the matched documents are aggregated, ranked and sent to classifier to filter the relevant documents out of which the top-k results are output. In this example, *NumLogicalProcessors 7* (see KB article 2154845) is a detected misconfiguration.

5. ONLINE QUERY PROCESSING PHASE

The goal of this phase is to leverage the inverted index to enable automated retrieval of relevant KB articles for a given snapshot. In ConfSeer, retrieval happens in three steps. First, the query is converted into an abstract form in the *Query Processing* stage and distributed to multiple matchers implemented in the *Matching Engine* by the *Query Distribution* module. Second, the Matching Engine executes multiple search logics and returns a list of articles with their scores. Finally, the *Ranking Engine* aggregates the results.

5.1 Query Processing

Three key requirements drive matching an input query of a configuration snapshot against the inverted index:

[R1] *Processing Homogeneity*: We need to ensure that the retrieval is performed *after* the query is processed in a manner similar to the offline analysis (§4.2) of KB articles. For instance, recall that using F3, we converted the input to lower case to enable case-insensitive matching. Therefore, we pass each input query through a *Query Analysis* module. The Query Analysis module performs two tasks: (a) prepare an abstract representation of the query by rewriting any value-types it finds (e.g., numbers like 7 or seven, boolean words like true or false) with generics (e.g., NUMERIC, BOOLEAN) and storing the value separately for constraint evaluation by the Matching Engine (§5.2), and (b) mimic the processing performed on the input corpus during the index building phase. Figure 3 shows the various filters that process the input query. Notice that the input query does not pass through the Expansion phase as we already injected synonyms during the index building stage.

[R2] *Fuzzing*: For many queries, using exact keywords may not return a result. For instance, a naive search for an input query "LSA Crash On Audit Fail Flag" would return only documents containing all the terms from the input query but it will not return documents where, for instance, the word "audit flag" is missing. Hence, it is important to construct approximate queries from the original one. We implement *Query Fuzzing* to address this issue. Specifically, given an input query, we construct a powerset of the terms in the input query to facilitate approximate searches that match documents containing a subset of terms from the input query.

[R3] *Matching Logic*: For an input query such as "ErrorID 823", an exact search or even an approximate search may not return documents containing the text "the errorId was logged to be 823" due to the presence of additional words between the terms from the input query. In such scenarios, it may be necessary to do relaxed queries that allow for other terms to appear in between the input query terms. We solve this problem by modularizing the matching logic – a *Query Distributor* does a parallel invocation of different matchers (e.g., exact matching, proximity matching) and finally aggregates the results (§5.3).

5.2 Matching Engine

To perform automated retrieval, a key question is how to establish a relationship between individual configuration parameters inside a snapshot and the KB articles. In particular, given a key-value pair with a constraint, we want to retrieve all KB articles that satisfy its truth condition. For instance, given "NumOfLogicalProcessors=7", we want to retrieve all KB articles where the user sets the number of logical processors to 7. However, there are numerous challenges associated with such a retrieval. First, KB articles are written in free-form text so an accurate mapping should be made from NumOfLogicalProcessors to candidate free-form text such as "ensure that the logical processors are set to seven". Second, the value being searched for may not even be present in the KB article e.g., "the number of logical processors is less than 8" has no 7 but still satisfies the constraint.

In ConfSeer, we implement two types of matchers: (1) *Exact Match* is a simple search mechanism that looks for documents that contain the key-value pairs in their exact form, and (2) *Proximity Match* incorporates proximity based approximate matching along with a constraint evaluation module as a mechanism to find two tokens within a given proximity of each other that satisfy a constraint (if any). Recall from §4.2 that tokens emitted during the analysis include a position from the previous token which allows us to support proximity matches using the inverted index.

Proximity matching happens in the following two stages:

Retrieval Phase: For an input query, the Proximity Matcher

retrieves the relevant documents as well as tracks positions of every term occurrence that matches in the inverted index. An additional pruning stage removes all matches which are beyond a specified distance. For instance, given a constraint such as "logical, processors, Distance=1", only documents containing "logical" and "processors" next to each other are retrieved. Subsequently, all words between the matched positional offsets are obtained from the inverted index. Note that setting the value of *Distance* too high may yield a false positive (e.g., 'logical' and 'processor' can appear in different sentences), while setting it too low may result in false negatives (e.g., using $Distance = 1$ for the sentence "the processor is a logical one" will not output "logical processor"). Using collocation analysis [56], we found that setting $Distance = 15$ in practice offers a good trade-off.

Constraint Evaluation Phase: Once the relevant documents are retrieved along with the set of words between the matched positions of the input query terms, ConfSeer performs an on-the-fly truth condition evaluation as follows: First, each match is passed through a Token Extractor module (see Figure 4). Based on linguistic analysis, we incorporate two pieces of information into the Token Extractor module: (a) frequently-used comparative expressions in free-form text (e.g,. 'less than or equal to', 'set to', 'greater than') and (b) a simple type system to infer values (e.g., numerical, boolean, floating point, and version numbers). Using this information, comparative expressions along with any surrounding values are extracted. Second, an Abstract Syntax Tree (AST) is constructed for individual expressions to be evaluated. Recall from the Query Analysis phase that any values found in the input query are abstracted by the Value-Type Filter (filter F2 from §4.2) and substituted with a generic type. Therefore, the AST will be constructed with the operator as the root node and the operands (value extracted from the query and the retrieved document) as the leaf nodes. Third, the ASTs are compiled in-memory and evaluated to obtain the truth condition. Only matches satisfying the truth condition are returned for subsequent ranking.

Example. Figure 4 illustrates these steps for the example query "NumOfLogicalProcessors 8". In Step 1, the Query Processor performs query analysis using all filters except Expansion and Normalization (see Figure 3) to obtain "num logical processor 7"; we omit processing by other filters for simplicity. Next, the value-type in this query is extracted into a separate field and substituted with a generic token (NUMERIC in this case). In Step 2, a simple proximity search is made using the generic query "num logical processor NUMERIC" and the relevant document identifiers [3,67,124] are retrieved. In Step 3, the positional information along with the text between these positional offsets is obtained for each match from the inverted index. In Step 4, the Token Extractor module parses the free-form text and constructs the appropriate token representations (e.g., "less than or equal to" $\rightarrow \leq$, "eight" $\rightarrow 8$, "greater than" $\rightarrow >$). In Step 5, an AST is constructed with the leaves representing the values (i.e., [7,8], [7,64], [7,32]) to be compared and the root node representing the type of operator (i.e., $\leq$, $>$, $>$) to be executed on the leaves. Finally, in Step 6, the Truth Condition Evaluator compiles the AST using a compiler and checks if it satisfies the condition and the matching documents (DocID: 3) are returned for subsequent ranking.

5.3 Ranking

Our goal in this phase is to score and rank candidate KB articles. Because there can be multiple matchers (see Figure 4), ranking happens at multiple stages:

Scoring results from an exact match: We use a slightly modified variant of the standard *tf-idf* [54] (term frequency-inverse document frequency) metric to score the retrieved results from both exact and fuzzy matching. In *tf-idf*, each of the terms in a document is associated with the number of times it occurs in that document. Terms are then weighted according to how common they are across the corpus, the intuition being that rare terms are more central to the meaning of a document than the terms that occur frequently. The score $S(q,d)$ of a document d for a query q is calculated as:

$$S_{exact}(q,d) = c(q,d) \cdot F(q) \cdot \sum_{t \in q} \left(tf(d) \cdot idf(t)^2 \right) \quad (1)$$

where $c(q,d)$ is the normalized ratio of the number of query terms from q found in the document d and the total number of terms in the query q. Thus, presence of all query terms in a given document will increase this ratio and vice versa. Recall the Query Fuzzing module from §5.2 that fuzzes a given query. Intuitively, hits from "fuzzed" queries should receive a lower score compared to those from the original query. Therefore, we define a *fuzz factor* $F(q)$ to take this into consideration. $F(q)$ is defined as the inverse of the distance between the fuzzed query q and the original query (say, Q) where distance is defined as the number of terms that need to be added to q to get Q. Notice that for $q = Q$, $F(q) = \infty$ giving us an invalid score. To handle this case i.e., prevent zeros in the denominator, we apply Laplace correction by adding 1 to the denominator which yields $F(q) = 1$ when $q = Q$. $tf(d)$ is the frequency of the term t in document d. Therefore, documents having more occurrences of a given term receive a higher score. Finally, $idf(t)$ or inverse document frequency measures how common or rare a term is across documents and is computed by taking the ratio of the total number of documents D and the number of documents containing the term t:

$$idf(t) = 1 + log\left(\frac{|D|}{|d \in D : t \in d|}\right) \quad (2)$$

Scoring results from a proximity match: To score a result from a proximity match, we have to take into account the maximum allowable positional distance between terms. Intuitively, the score should be inversely proportional to the positional distance between terms i.e., larger the distance between the matched terms, smaller the assigned score. We compute this score based on tf-idf [54]:

$$S_{prox}(q,d) = c(q,d) \cdot F(q) \cdot \frac{1}{\sqrt{\sum_{t \in q} idf(t)^2}} \sum_{t \in q} idf(t) \frac{L_{avg}(d)}{L(d)} \quad (3)$$

where $c(q,d)$ is same as before, L_{avg} is the average length of a document and $L(d)$ is the length of document d. $F(q)$ is the *fuzz factor* as before computed as $\frac{1}{D_1+D_2}$ where (1) D_1 is the distance of the query q from the original query (say, Q), and (2) D_2 is the number of positional moves of the terms in the matched document required to approximate Q. For example, consider the case when $q = Q =$"Crash On Audit". As $q = Q$, there is no fuzzing and hence $D_1 = 0$. But if we consider the

matched text "Audit flag had the crash bit" then $D_2 = 5$ because it takes 4 positional moves to move "crash" to the location of "Audit" and then 1 positional move to move "Audit" to the next location giving us the final fuzz factor of $\frac{1}{5}$.

Aggregation of results from multiple matchers. Recall that ConfSeer incorporates multiple matchers and does not make any assumptions on the aggregation function, but here we give one for illustration. We build the final result set by aggregating the results from individual matchers. We found in practice that a linear aggregation with weighted ranking works well:

$$S(q,d) = \sum_{x \in n} w_x \cdot M_x \quad (4)$$

where w_x is the weight assigned to the matcher M_x and n is the total number of matchers implemented ($n = 2$ for ConfSeer). Through sensitivity analysis, we set $w_{exact} = 1.0$ and $w_{proximity} = 0.25$.

ConfSeer outputs the top-k articles sorted by score for all detected misconfigurations (see Figure 4) which are then filtered based on relevance (§6). In practice, we found that the number of actionable errors per configuration snapshot per machine was small (2-4 on average).

6. INTERACTIVE LEARNING PHASE

ConfSeer performs interactive learning to incrementally improve the accuracy and ranking of results. The key challenge is to differentiate matched articles solving a problem from the ones that may only be informational or recommendations e.g., tutorials, How-To articles.

To address this challenge, ConfSeer performs interactive machine learning. The goal of the learning component is to leverage the human feedback to accurately identify the relevant KB articles that pinpoint configuration problems. ConfSeer performs learning in two stages. First, to bootstrap, an expert is asked to tag a small set of KB articles with one of two labels: 'Problem-solving (P)' or 'Informational (I)'. Since we want to separate articles that *do not* fix errors, it suffices to do labeling at the document level as opposed to the query level i.e., whether an article is useful for a *specific* query; the latter approach also risks the state space explosion problem. Then, a classification model is trained based on this labeled data. Second, the set of ranked documents output by the *Online Query Processing Phase* are input to the classifier (see Figure 4) to filter informational-only KBs and show the relevant ones.

Building a classification model. Our initial training dataset consisted of 650 KB articles labeled by experts. Each document was represented by a set of 5,634 features corresponding to the text in the titles of the KB articles; adding more features like the text from the "Problem" section in the KBs reduced the accuracy due to increased noise. Capitalization was ignored and stop words (e.g., 'how', 'I', 'why') were removed. We use the stochastic gradient descent algorithm from Vowpal Wabbit (VW) [39] for interactive learning as it gave the highest accuracy. Interactive learning has the advantage that in a dynamic setting, every addition of a new data point only recalibrates the model as opposed to re-building it from scratch. This combined with the hashing trick [63] makes our system scalable. Finally, we use a Golden Section Search [36] to obtain the optimal model parameters. In our case, we found using elastic net regularization i.e., enabling both L1 (=5e-08) and L2 (=0.00001) regularizations yielded the best performance.

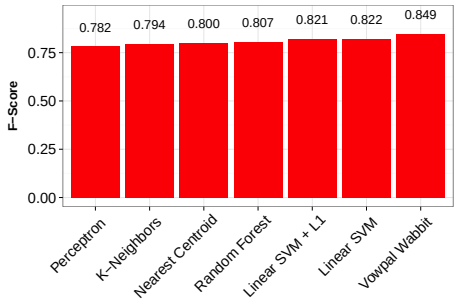

Figure 5: Accuracy comparison of different learning algorithms based on training data of 650 KB articles.

Evaluation. We use the standard F-score [71] computed as $\frac{2TP}{2TP+FP+FN}$, where TP, TN, FP, FN are true positives, true negatives, false positives and false negatives, respectively. F-score considers both precision ($\frac{TP}{TP+FP}$) and recall ($\frac{TP}{TP+FN}$), and its value of 1 indicates a perfect classifier. Finally, to minimize the bias associated with the random sampling of the training and validation data, we use stratified k-fold cross-validation to compute the predictive capability of our classifier. In k-fold cross-validation, the complete dataset D is randomly split into k mutually exclusive subsets (the folds: D_1, D_2, ..., D_k) of approximately equal size. The classification model is then trained and tested k times. Each iteration, it is trained on all $(k-1)$ but one fold. The cross-validation estimate of the overall F-score is calculated as the average of the k individual F-scores. In stratified k-fold, the only difference is that the folds are made by preserving the percentage of samples for each class. For our experiments, we set $k = 10$ as used in practice [37].

Classification Results. Our classification model was initially trained on 50 KB articles which gave us a poor F-score of 0.582. However, when the training set size was increased to 320 KB articles, the F-score significantly increased to 0.814. Adding more KB articles yielded diminishing returns. In the end, our classification model achieved an F-score of 0.849 when trained on a set of 650 KB articles. Figure 5 shows the accuracy comparison of the classifier implemented using VW with other standard learning algorithms.

7. SYSTEM EVALUATION

To evaluate ConfSeer, we want to answer the following questions: (1) How does ConfSeer compare against the perfect accuracy of human-written rules? (2) Does ConfSeer return semantically meaningful results and how does it compare against web search engines? (3) What is the contribution of each individual technique to overall accuracy? and (4) What is the end-to-end latency in querying ConfSeer? For all experiments, we use a Xeon 2.67 GHz eight-core node with 48 GB RAM.

7.1 Evaluating Accuracy

Comparison with Ground Truth Rules: Recall from §2 the notion of expert-written rules to map configuration problems to KB articles. These rules are typically a combination of XML and shell script files, and are executed by a rule engine

(a) (b) (c)

Figure 6: (a) Latencies incurred at different stages in the query pipeline, (b) ConfSeer accuracy on real configuration errors from four large enterprise clusters containing thousands of machines, and (c) Total number of recommendations output by ConfSeer deployed in production.

on configuration snapshots from customer servers. To evaluate the accuracy of ConfSeer in comparison to human-written rules, we randomly selected 120 rules from the rule repository of the previous release of the OMS system. Our goal is to verify if ConfSeer is able to output the same KB article as the match suggested by the expert-defined rule.

We evaluate accuracy of ConfSeer as follows. First, for each rule, we get an associated configuration snapshot on which an alert was trigged (i.e., the rule engine detected one or more configuration errors). Second, we treat each line in the configuration snapshot as a single query to ConfSeer. We return the top-3, top-2 and top-1 scoring results for each query. Third, we check if ConfSeer returned the ground truth KB article associated with the rule in its result set. Note that this step checks for both the "presence" of the ground truth as well as takes ranking into consideration. This is because a given configuration snapshot could have multiple misconfigurations in place not covered by a single rule.

Table 2 row 1 shows the results. For the top-3 result set, we observed the accuracy of ConfSeer to be 97.5% i.e., its suggestions matched 117/120 human-written rules which currently cover tens of thousands of configuration snapshots from user servers. The remaining three cases were attributed to ambiguity in semantically matching the free-form text in the corresponding KB articles. For the top-2 and top-1 results, ConfSeer still achieves a reasonable accuracy of about 95% and 91%, respectively.

Survey from Domain-Experts: We conducted a user study involving four domain-experts to evaluate the accuracy of ConfSeer for knowledge-based matching. Each domain-expert was shown 30 randomly selected snapshots for a total of 120 along with the KB identifiers of the top-3 results (by score) that ConfSeer returned corresponding to the errors detected from those configuration snapshots. In practice, we found that the number of actionable errors per snapshot per machine was small. Subsequently, the domain-expert was asked to validate if the KB articles were "semantically" relevant to the snapshot. In each case, we attributed success *only* when ConfSeer suggested the relevant article in the first position. Note that this evaluation is more strict compared to the previous comparison. Further, it allows us to evaluate both the matching accuracy and ranking of results output by ConfSeer.

Table 2 rows 2-5 show the results. As expected, ConfSeer achieves 100% accuracy for exact queries. Proximity queries, on the other hand, are among the most complicated to handle — results retrieved contain the words from the query but they may not be semantically relevant according to the domain-expert. As ConfSeer leverages several natural language processing techniques, it is also subject to their well-known limitations caused by anaphoras e.g., the text "run the CsKerberosAssignment command, enable its topology bit" will be fetched for the query "IsKerberosEnabled" as it is non-trivial to associate "enable" only with "topology" but not with "kerberos"; we plan to address this problem in future work. Overall, ConfSeer achieves an accuracy of 85%-97.5%.

Comparison with Search Engines: For completeness, we compare ConfSeer with popular search engines. Because a direct comparison is not possible (e.g., search engines will fail to understand an entire configuration snapshot expressed as a query), we leverage the help of a domain-expert to achieve this task. The domain-expert randomly selected another set of 200 queries from the existing human-written rules such that in each case, it was possible to specify only one relevant knowledge base article without ambiguity. Next, the domain expert queried Google, Bing, and ConfSeer using the <param, value> entries (e.g., <NumLogicalProcessors, 8>) mimicking how real users copy-paste error messages or parameter names from logs as keyword queries for search engines, and recorded the position at which the KB article appeared. All results beyond the first page on the search engines were ignored. We added the search parameter "site:support.microsoft.com" to ensure fairness and provide sufficient hints to the search engine to cover only the articles from that site. We observed the following accuracy results: 69.5% (Bing), 65.5% (Google), and 95.5% (ConfSeer). However, this only takes into account whether the ground truth appeared anywhere in the first result page. If we consider the more strict criteria that the ground truth article should appear in the first place, then web search engines exhibit a much lower accuracy: 60.5% for Bing and 55.5% for Google whereas ConfSeer still has a reasonable accuracy of 91%. ConfSeer's main advantage over web search engines is its capability to perform constraint evaluation on configuration parameters and free-form text, its ability to do fuzzy matching with synonyms, and its functionality of adding semantic context to documents.

Figure 7 shows the contribution of each individual technique in ConfSeer to the overall accuracy by omitting each filter (in Figures 3) and fuzzed queries, exact and proximity matcher (in §4) one at a time and measuring the end-to-end accuracy. Overall, the canonicalization filter dominates, but

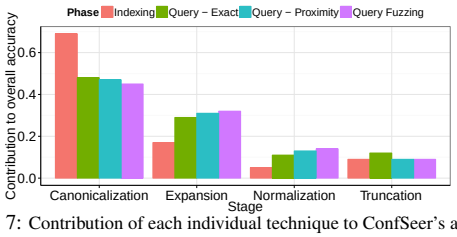

Figure 7: Contribution of each individual technique to ConfSeer's accuracy.

Type	Mean	Median	Q95	StdDev	COV
Individual Latencies Inside a Query (ms)					
Rewrite	0.018	0.018	0.044	0.013	0.751
Fuzz	0.001	0.001	0.002	3.8e-04	0.338
Retrieval	10.912	1.336	51.438	19.422	1.781
End-to-End Latencies (ms)					
Exact	4.320	1.175	26.363	7.677	1.778
Proximity	12.395	7.947	40.314	12.606	1.017
Constraint	19.550	20.271	50.363	15.560	0.795

Table 3: Latencies (in ms) at different stages in the query pipeline and total end-to-end latency for different query types; COV denotes coefficient of variation.

the additive effect of the other filters also contributes significantly towards the end-to-end accuracy.

Accuracy in real deployments: ConfSeer started being deployed within Microsoft cautiously with a private preview for a few selected customers in January 2014. As its deployments matured, the install base grew rapidly and currently its public release analyzes configurations on tens of thousands of machines spanning thousands of customers; absolute counts omitted due to confidentiality reasons. Figure 6(b) shows that ConfSeer achieves 80% - 100% accuracy across four large real customer deployments. Figure 6(c) shows the total number of recommendations across all deployments of ConfSeer which is 50x larger than the previous rule based system (till December 2013) due to ConfSeer's automated indexing/matching of about 100k+ KB articles.

7.2 Evaluating Performance

To evaluate the distribution of the time taken during each stage of the query pipeline, we issue 100K random queries taken from configuration snapshots against ConfSeer. Figure 6(a) shows the results. Observe that the median time to fuzz an input query is 1 μs. The reason for this is because 95% of the queries contain no more than six individual terms (e.g., "NumOfLogicalProcessors 8" contains four individual terms after camel case expansion). Further, we observe that the median retrieval latency is 1.3 ms. The reason for a higher 95th percentile and standard deviation is due to the presence of different types of complex queries as we show next.

Table 3 shows the latency split by query type. As expected, constraint evaluation takes the longest (median of 20 ms) because expressions are dynamically constructed and compiled. The COV values indicate that the constraint evaluation queries take the longest in general whereas exact and proximity queries exhibit high variability e.g., some exact queries take $>$ 26 ms to execute. Upon inspecting the queries that took the longest from the query log, we observed this was due to cases where a subset of the query terms had a high frequency e.g., sql. Although we pick the top-3 documents for each query, determining the global top-3 requires evaluating the score for each individual document and it becomes expensive when the retrieved result set is huge. Overall, ConfSeer experiences a reasonable 50 ms worst case latency per query *without* optimizations such as query parallelization; we plan to explore them in future work.

8. RELATED WORK

The work related to ConfSeer falls into three categories. First, there are research proposals for detecting [31, 67, 75] and troubleshooting [13, 15, 16] misconfigurations. In addition, the concept of using statistical techniques [61, 62, 51] for problem identification has emerged as a powerful tool aiding black-box detection. Strider [62] filters suspicious entries by using manually labeled set of configuration settings. Recently, there have been proposals [70, 42, 69, 76] on detecting anomalous patterns based on console logs or ground truth configuration snapshots. ConfSeer, on the other hand, builds its ground-truth from already available expert-written knowledge base articles. In contrast to these systems, ConfSeer also pinpoints the error and its solution.

Zhang et al. [76] detect misconfigurations by learning the interaction between the configuration settings and the executing environment, and correlations between configuration entries. Our system is complementary to theirs – instead of learning from a set of sample configurations, ConfSeer uses existing KB articles as ground truth.

Tools for analyzing and testing configurations [68, 35, 19] can be used to generate test cases to detect misconfigurations. Our work is complementary to ConfErr [35] and fuzz testing [18, 25] (for generating random data as configuration settings) in that ConfSeer can be leveraged by these tools to generate high-quality test cases. For instance, the values and value ranges extracted by ConfSeer from the KB articles can be used to narrow down the number of test cases to be generated.

Second, there is a large body of work on keyword association mining [72], automated concept determination [26], relationship extraction [33] and entity resolution [14] and supporting efficient keyword searches [27, 52]. None of these systems provide an *end-to-end* solution for our problem domain. In addition, these systems cannot be leveraged for supporting value-type queries or fuzzy matches as ConfSeer, which is fundamental for constraint evaluation. We also plan to leverage these complementary efforts in ConfSeer in future work.

Third, there are some recent efforts [38, 41, 53, 58] on automatic construction of knowledge bases that can help in solving this problem — through extraction of structured information from natural language text. These techniques can be based on rules (e.g., dictionaries [49], ontologies [66]), patterns [30], or statistics (e.g., CRFs [43], SVMs [32]). Simoes et al. [55] propose an optimization for the information extraction process. NetSieve [49] focused on inferring root causes of network failures from trouble tickets. The idea is to build a domain-specific dictionary of frequent, technical keywords used in tickets and then use this dictionary to output a *summary* of problems in a given ticket. ConfSeer's techniques are complementary to these efforts and in some cases we benefit from them. For instance, we leverage the WLZW and RIDF techniques from NetSieve to build the list of Named-Entities (§4.2.2).

Row	Evaluation Type	Accuracy % Exact	Accuracy % Proximity	Accuracy % Constraint Evaluation	Accuracy % Overall
1	**Human-written Ground Truth (Top-3)**	100%	93.8%	100%	97.5%
	Human-written Ground Truth (Top-2)	100%	89%	98%	95%
	Human-written Ground Truth (Top-1)	100%	81.6%	97%	91.3%
2	**Domain-expert 1**	100%	93.3%	100%	98.7%
3	**Domain-expert 2**	100%	85.7%	100%	97%
4	**Domain-expert 3**	100%	90%	100%	98%
5	**Domain-expert 4**	100%	87.5%	100%	97.5%
6	**Search Engine: Bing**	100%	46.9%	19.3%	69.5%
7	**Search Engine: Google**	83%	51%	22.5%	65.5%

Table 2: Evaluating ConfSeer accuracy against human-written rules, domain-experts and search engines.

There are also state-of-the-art methods for harnessing knowledge bases for data and text analytics. Comprehensive machine-readable KBs have been pursued since the seminal projects Cyc [40] and WordNet [46]. Other prominent endeavors include DBpedia [17], KnowItAll [29], NELL [20] and YAGO [59], as well as industrial ones such as Freebase. These projects provide automatically constructed KBs of facts about named entities, their semantic classes, and their mutual relationships. While we leverage NLP fundamentals from these works, we leave it to future work to explore how these systems can help improve ConfSeer's accuracy.

9. CONCLUSION

Software misconfigurations continue to be prevalent and are a major cause of system failures. We presented ConfSeer that automatically detects configuration errors by matching them against a knowledge base of technical articles. Our evaluation of ConfSeer shows that it achieves 80%-97.5% accuracy while incurring low overheads. ConfSeer has been running in Microsoft Operations Management Suite for over a year to automatically detect software configuration errors. In future work, we plan to extend our techniques to (a) handle KB articles with multiple conditional expressions and dependence between configuration parameters, and (b) analyze technical blogs and question and answer forums (e.g., stackoverflow) written in free-form text similar to KB articles.

10. ACKNOWLEDGEMENTS

We thank Arnd Christian König, Vivek Narasayya, Sriram Rao, Raghu Ramakrishnan and the OMS team for insightful discussions, and the anonymous reviewers for their feedback.

11. REFERENCES

[1] Apple Knowledge Base. http://kbase.info.apple.com/.
[2] Desk.com. http://desk.com.
[3] EMC Powerlink. http://powerlink.emc.com.
[4] Google Knowledge Base. http://goo.gl/6wN6oB.
[5] How to query the Microsoft Knowledge Base. http://support.microsoft.com/kb/242450.
[6] IBM Software Knowledge Base. http://goo.gl/fY0cDQ.
[7] Microsoft Operations Management Suite. http://www.microsoft.com/en-us/server-cloud/operations-management-suite/.
[8] Microsoft Support. http://support.microsoft.com/.
[9] Oracle Support. http://support.oracle.com.
[10] StackOverflow. http://stackoverflow.com.
[11] SysSieve. http://research.microsoft.com/en-us/um/people/navendu/syssieve/.
[12] VMWare KB - Knowledge Base Articles for all VMWare Products. http://kb.vmware.com.
[13] B. Agarwal, R. Bhagwan, T. Das, S. Eswaran, V. N. Padmanabhan, and G. M. Voelker. Netprints: Diagnosing home network misconfigurations using shared knowledge. In *NSDI*, 2009.
[14] H. Altwaijry, D. V. Kalashnikov, and S. Mehrotra. Query-driven approach to entity resolution. *Proceedings of the VLDB Endowment*, 2013.
[15] M. Attariyan, M. Chow, and J. Flinn. X-ray: automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.
[16] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, 2010.
[17] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. *Dbpedia: A nucleus for a web of open data*. Springer, 2007.
[18] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *IEEE ICSE*, 2013.
[19] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.
[20] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr, and T. M. Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, 2010.
[21] K. Chen, C. Guo, H. Wu, J. Yuan, Z. Feng, Y. Chen, S. Lu, and W. Wu. Generic and automatic address conf for data center networks. *SIGCOMM CCR*, 2010.
[22] X. Chen, Y. Mao, Z. M. Mao, and J. Van der Merwe. Declarative configuration management for complex and dynamic networks. In *CoNEXT*, 2010.
[23] K. Church and W. Gale. Inverse document frequency (idf): A measure of deviations from poisson. In *NLPVLC*. 1999.
[24] D. Crockford. The application/json media type for javascript object notation (json). https://goo.gl/SM1kDa, 2006.
[25] H. Dai, C. Murphy, and G. Kaiser. Configuration fuzzing for software vulnerability detection. In *IEEE ARES*, 2010.
[26] D. Deng, Y. Jiang, G. Li, J. Li, and C. Yu. Scalable column concept determination for web tables using large knowledge bases. *VLDB*, 2013.
[27] H. Duan, C. Zhai, J. Cheng, and A. Gattani. Supporting keyword search in product database: a probabilistic approach. *Proceedings of the VLDB Endowment*, 2013.
[28] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with ituned. *VLDB Endowment*, 2009.
[29] O. Etzioni, M. Cafarella, D. Downey, S. Kok, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, and A. Yates. Web-scale information extraction in knowitall. In *ACM WWW*, 2004.

[30] Y. Fang and K. C.-C. Chang. Searching patterns for relation extraction over the web: rediscovering the pattern-relation duality. In *WDSM*. ACM, 2011.

[31] N. Feamster and H. Balakrishnan. Detecting bgp configuration faults with static analysis. In *NSDI*, 2005.

[32] C. Giuliano, A. Lavelli, and L. Romano. Exploiting shallow linguistic information for relation extraction from biomedical literature. In *EACL*. Citeseer, 2006.

[33] M. U. Haq, H. Ahmed, and A. M. Qamar. Dynamic entity and relationship extraction from news articles. In *ICET*. IEEE, 2012.

[34] A. Kapoor. Web-to-host: Reducing total cost of ownership. Technical report, 200503, The Tolly Group, 2000.

[35] L. Keller, P. Upadhyaya, and G. Candea. Conferr: A tool for assessing resilience to human configuration errors. In *DSN*, 2008.

[36] J. Kiefer. Sequential minimax search for a maximum. *AMS*, 1953.

[37] R. Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI*, volume 14, pages 1137–1145, 1995.

[38] H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. *VLDB*, 2010.

[39] J. Langford, L. Li, and T. Zhang. Sparse online learning via truncated gradient. *JMLR*, 2009.

[40] D. B. Lenat and R. V. Guha. *Building large knowledge-based systems; representation and inference in the Cyc project*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[41] G. Limaye, S. Sarawagi, and S. Chakrabarti. Annotating and searching web tables using entities, types and relationships. *Proceedings of the VLDB Endowment*, 2010.

[42] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li. Mining invariants from console logs for system problem detection. *ATC*, 2010.

[43] A. McCallum and W. Li. Early results for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons. In *NLL*. ACL, 2003.

[44] Microsoft. Microsoft Developer Network. http://msdn.microsoft.com/.

[45] Microsoft. Windows Error Reporting. http://goo.gl/Tma5G3.

[46] G. A. Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 1995.

[47] J. C. Perez. Google outages blamed on sign-in system. http://goo.gl/PScp6m.

[48] M. F. Porter. Snowball: A language for stemming algorithms, 2001.

[49] R. Potharaju, N. Jain, and C. Nita-Rotaru. Juggling the jigsaw: Towards automated problem inference from network trouble tickets. In *NSDI*, 2013.

[50] A. Rabkin and R. Katz. Precomputing possible configuration error diagnoses. In *IEEE ASE*, 2011.

[51] V. Ramachandran, M. Gupta, M. Sethi, and S. R. Chowdhury. Determining configuration parameter dependencies via analysis of configuration data from multi-tiered enterprise applications. In *ACM ICAC*, 2009.

[52] W. Rao, L. Chen, P. Hui, and S. Tarkoma. Bitlist: New full-text index for low space cost and efficient keyword search. *VLDB*, 2013.

[53] V. Rastogi, N. Dalvi, and M. Garofalakis. Large-scale collective entity matching. *Proceedings of the VLDB Endowment*, 2011.

[54] G. Salton and M. J. McGill. Introduction to modern information retrieval, McGraw-Hill, Inc. 1986.

[55] G. Simoes, H. Galhardas, and L. Gravano. When speed has a price: Fast information extraction using approximate algorithms. *VLDB*, 2013.

[56] F. Smadja. Retrieving collocations from text: Xtract. *CL*, 1993.

[57] Y.-Y. Su, M. Attariyan, and J. Flinn. Autobash: improving configuration management with operating system causality analysis. *SIGOPS OSR*, 2007.

[58] F. M. Suchanek, S. Abiteboul, and P. Senellart. Paris: Probabilistic alignment of relations, instances, and schema. *VLDB*, 2011.

[59] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*. ACM, 2007.

[60] The Guardian. Choose customer service without the call center. http://goo.gl/8sUj7D.

[61] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *OSDI*, 2004.

[62] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. Strider: A black-box, state-based approach to change and configuration management and support. *SCP*, 2004.

[63] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. Feature hashing for large scale multitask learning. In *ICML*, 2009.

[64] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In *OSDI*, 2004.

[65] A. Wilhelm. Microsoft: Azure went down in Western Europe due to misconfigured network device. http://goo.gl/USVRlC.

[66] W. Wu, H. Li, H. Wang, and K. Q. Zhu. Probase: A probabilistic taxonomy for text understanding. In *SIGMOD*. ACM, 2012.

[67] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki. Generating range fixes for software configuration. In *ICSE*, 2012.

[68] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy. Do not blame users for misconfigurations. In *In ACM SOSP*, 2013.

[69] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP*. ACM, 2009.

[70] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan. Mining Console Logs for Large-Scale System Problem Detection. *SysML*, 2008.

[71] M. Yamamoto and K. Church. Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus. *CL*, 2001.

[72] Z. Yan, N. Zheng, Z. G. Ives, P. P. Talukdar, and C. Yu. Active learning in keyword search-based data integration. *The VLDB Journal*, 2015.

[73] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *SOSP*. ACM, 2011.

[74] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated known problem diagnosis with event traces. In *SIGOPS OSR*, 2006.

[75] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar. Context-based online configuration-error detection. In *ATC*, 2011.

[76] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou. Encore: exploiting system environment and correlation information for misconfiguration detection. In *ASPLOS*, 2014.

[77] S. Zhang and M. D. Ernst. Automated diagnosis of software configuration errors. In *ICSE*, 2013.