

Performance Prediction for Set Similarity Joins

Christiane Faleiro Sidney
Federal University of Lavras, Brazil
Department of Computer Science
christiane.faleiro@lemaf.ufla.br

Leonardo Andrade Ribeiro
Federal University of Goiás, Brazil
Instituto de Informática
laribeiro@inf.ufg.br

Diego Sarmiento Mendes
Federal University of Lavras, Brazil
Department of Computer Science
diego.mendes@posgrad.ufla.br

Theo Härder
University of Kaiserslautern, Germany
Department of Computer Science
haerder@cs.uni-kl.de

ABSTRACT

Query performance prediction is essential for many important tasks in cloud-based database management including resource provisioning, admission control, and pricing. Recently, there has been some work on building prediction models to estimate execution time of traditional SQL queries. While suitable for typical OLTP/OLAP workloads, these existing approaches are insufficient to model performance of complex data processing activities for deep analytics such as cleaning and integration of data. These activities are largely based on similarity operations—radically different from regular relational operators. In this paper, we consider prediction models for set similarity joins. We exploit knowledge of optimization techniques and design details popularly found in set similarity join algorithms to identify relevant features, which are then used to construct prediction models based on statistical machine learning. An extensive experimental evaluation confirms the accuracy of our approach.

Categories and Subject Descriptors

H.2.4 [Database Management]: Systems—*Query processing*; C.4 [Computer Systems Organization]: Performance of Systems—*Modeling techniques*

General Terms

Algorithms, Management, Measurement, Performance

Keywords

Set Similarity Join, Performance Prediction, Cloud Databases

1. INTRODUCTION

Predicting query execution time has been widely recognized as a key-enabling technology behind (multi-tenant) data management services hosted in the cloud [5, 1, 9].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'15 April 13-17, 2015, Salamanca, Spain.

Copyright 2015 ACM 978-1-4503-3196-8/15/04...\$15.00.

<http://dx.doi.org/10.1145/2695664.2695694>

These systems have the challenging task of managing shared resources in a cost-effective way while meeting service-level agreements (SLAs). Accurate estimates of query execution time can support many important system management decisions in such a context, including:

- *Resource Provisioning*: Assessing achievable performance for a given hardware configuration is instrumental to minimize the risk of resource underprovisioning or overprovisioning.
- *Admission Control*: Incoming queries can be blocked, queued, or scheduled for immediate execution depending on their estimated runtime.
- *Pricing*: Instead of virtualized resources such as CPU, I/O, and memory, providers can define SLAs based on the assurance of high-level, measurable performance metrics such as query response time and throughput.

Recently, there has been a flurry of recent work on predicting execution time (and resource usage) of SQL queries [6, 5, 8, 1, 15]. Most proposals use machine learning techniques to uncover statistical relationships between measured query runtimes and features, such as occurrence count of each operator type in the query plan and optimizer estimates of operators' cardinalities and costs. Another approach consists in directly tuning the cost model and calibrating cardinality estimates of the chosen query plan to predict runtimes [15].

The popularization of cloud computing and the ever increasing hype around “Big Data” have been fueling a need for on-demand deep analytics support in cloud-based database services [3]. But current approaches to performance prediction of SQL queries are insufficient to model complex data processing activities involved in deep analytics. Examples of such activities are data integration and cleaning, which are largely based on operations beyond traditional SQL queries.

The *set similarity join* is a core operation for text data cleaning and integration [4, 2, 11, 16, 12], which pairs strings represented as sets, whose similarity is not less than a specified threshold. A set similarity function is employed in the join predicate to mathematically approximate the underlying notion of similarity. The set similarity join is attractive owing to its efficiency in dealing with large datasets and versatility in supporting a variety of similarity functions.

Predicting the runtime of set similarity joins is a difficult task. Most set similarity join algorithms adopt a processing model based on inverted lists [13, 2, 11, 16, 12], which is radically different from approaches based on relational database technology [4]. Moreover, runtimes of different executions

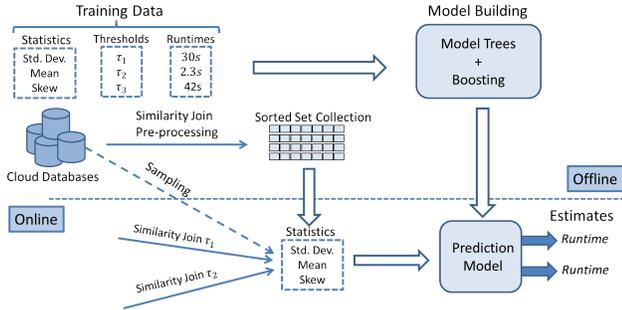


Figure 1: Our framework for performance modeling.

can vary dramatically depending on threshold values and data distribution. Algorithms based on inverted lists deliver considerably better performance than the ones on top of SQL engines (e.g., see experiments in [16]) and, therefore, are likely to be the preferred solution in cloud-based systems with strict scalability requirements.

In this paper, we present an approach to performance prediction of set similarity joins. To the best of our knowledge, this is the first contribution addressing this problem. We analyze the impact of popular optimization techniques and algorithm design details to judiciously select a compact set of data statistics that primarily dictates the corresponding algorithm performance. These statistics are used along with threshold values as input to a machine learning technique. In this context, we adopt an experiment-driven modeling approach to cover a representative space of possible data distributions and threshold values. We present a framework that deploys our model to predict the runtime of similarity queries. All statistical features can be easily collected in a pre-preprocessing phase of set similarity join algorithms. Finally, we conduct an extensive evaluation to validate the accuracy of our approach.

The rest of this paper is organized as follows. Our framework is introduced in Section 2 and background material is provided in Section 3. Our performance modeling approach is presented in Section 4. We describe the experimental setup and discuss our results in Section 5. Relevant related work is discussed in Section 6, before we wrap up with conclusions in Section 7.

2. OUR PREDICTION FRAMEWORK

In this section, we give an overview of our approach. Figure 1 illustrates the main operations encompassing the performance modeling of set similarity joins. We adopt statistical learning techniques to build a performance prediction model. As in previous work [6, 8, 1], the model is built offline using training data and deployed online to obtain runtime estimates. However, there exist unique aspects in our approach owing to key differences between a single similarity join operator and execution plans of regular SQL queries. On the other hand, prediction models built in our approach can be—in principle—composed with other models in an operator-level modeling approach [1] in cases where similarity joins appear as part of a larger query (see Section 6).

In the offline phase, we mainly use statistics collected from semi-synthetic data as training features. In our experiment-driven modeling approach, we generate new data from original real-world data to substantially increase the coverage

of possible data distributions and thresholds (see Section 5). The similarity threshold is the *single feature* extracted from similarity join instances. This aspect sharply contrasts with approaches targeting SQL queries, where a rich feature space can be extracted from query plans. Together with the query execution time as dependent variable, statistics and threshold values are used as input to a statistical learning technique based on *boosted model trees* [14, 10].

In the online phase, the prediction model built offline is deployed to estimate execution time of incoming queries before they start executing. To this end, we need the statistics of the same kind as used for training. We can obtain these statistics using offline sampling—therefore, avoiding the relative high runtime overhead associated with online sampling. An alternative is to collect statistics during the pre-processing phase of the similarity join algorithm, which is usually performed offline anyway.

Cloud-based data integration services are prominent examples of systems that can benefit from our framework to performance prediction of similarity joins. But, an even more compelling example is perhaps cloud-based support of data exploration for deep analytics [3]. In this scenario, instead of being used in “single-shot” data integration activities, similarity join operations could be repeatedly invoked by different users with varying thresholds during the data exploration process.

Finally, we only consider performance prediction of a single, stand-alone similarity join query. Predicting execution time of multiple queries running concurrently is an important and challenging problem left for future work.

3. BACKGROUND

We map strings to *sets of tokens* using the popular concept of *q-grams*, i.e., substrings of size q obtained by “sliding” a window over the characters of a given string. For example, the string “token” can be mapped to the set of 2-gram tokens {**to,ok,ke,en**}. As the result can be a multi-set, we append the symbol of a sequential ordinal number to each occurrence of a token [4] to convert multi-sets into sets.

We associate a weight with each token to obtain *weighted sets*. A widely adopted weighting scheme is the Inverse Document Frequency (*IDF*), which associates a weight $w(t)$ to a token t as follows: $w(t) = \ln(1 + N/df(t))$, where $df(t)$ is the *document frequency*, i.e., the number of strings a token t appears in a database of N strings. The intuition behind using *IDF* is that rare tokens are more discriminative, usually carry more content information, and, thus, are more important for similarity assessment. We obtain *unweighted sets* by associating the value of 1 to each token. The size of a set r , denoted by $|r|$, is given by the number of tokens in r , whereas the weight of r , denoted by $w(r)$, is given by weight summation of its tokens, i.e., $w(r) = \sum_{t \in r} w(t)$; we have $|r| = w(r)$ for unweighted sets.

Given two sets r and s , a *set similarity function* $sim(r, s)$ returns a value in $[0, 1]$ to represent their similarity; a larger value indicates that r and s have a higher similarity. Definitions of widely used set similarity functions, namely, Jaccard, Dice, and Cosine, are shown in Table 1. Given two set collections \mathcal{C}_R and \mathcal{C}_S , a set similarity function sim , and a threshold τ , the *set similarity join* between \mathcal{C}_R and \mathcal{C}_S returns all scored set pairs $((r, s), \tau')$ s.t. $(r, s) \in \mathcal{C}_R \times \mathcal{C}_S$ and $sim(r, s) = \tau' \geq \tau$. Henceforth, we use the term similarity function (join) to mean set similarity function (join).

Table 1: Set similarity functions.

Function	Definition	$overlap(r, s)$	$[minw(r), maxw(r)]$
Jaccard	$\frac{w(r \cap s)}{w(r \cup s)}$	$\frac{\tau \cdot (w(r) + w(s))}{1 + \tau}$	$\left[\tau \cdot w(r), \frac{w(r)}{\tau} \right]$
Dice	$\frac{2 \cdot w(r \cap s)}{w(r) + w(s)}$	$\frac{\tau \cdot (w(r) + w(s))}{2}$	$\left[\frac{\tau \cdot w(r)}{2 - \tau}, \frac{(2 - \tau) \cdot w(r)}{\tau} \right]$
Cosine	$\frac{w(r \cap s)}{\sqrt{w(r) \cdot w(s)}}$	$\tau \cdot \sqrt{w(r) \cdot w(s)}$	$\left[\tau^2 \cdot w(r), \frac{w(r)}{\tau^2} \right]$

All similarity functions considered measure the overlap between two input sets to derive a similarity value. Thus, predicates involving similarity functions can often be equivalently represented in terms of an *overlap bound* [4]. Formally, the overlap bound between the sets r and s , denoted by $overlap(r, s)$, is a function that maps a threshold τ and the weights of r and s to a real value, s.t. $sim(r, s) \geq \tau$ iff $w(r \cap s) \geq overlap(r, s)$. Now, the similarity join can be reduced to the problem of identifying all set pairs r and s whose overlap is not less than $overlap(r, s)$. Further, similar sets have, in general, roughly similar weights. We can thus derive bounds for immediate pruning of candidate pairs whose weights differ enough. Formally, the weight bounds of r , denoted by $minw(r)$ and $maxw(r)$, are functions that map τ and $w(r)$ to a real value s.t. $\forall s$, if $sim(r, s) \geq \tau$, then $minw(r) \leq w(s) \leq maxw(r)$ (originally defined as size bounds for unweighted sets [13]). Thus, given a set r , we can safely ignore all sets whose weights do not fall within the interval $[minw(r), maxw(r)]$. Table 1 shows the overlap and weight bounds of the previous similarity functions.

We can also prune a large share of the comparison space by exploiting the *prefix filtering principle* [13, 4]. Prefixes allow selecting or discarding candidate pairs by examining only a fraction of the original sets. First, fix a global ordering \mathcal{O} ; tokens of all sets are then sorted according to this ordering. A set $r' \subseteq r$ is a prefix of r if r' contains the first $|r'|$ tokens of r . Further, $pref_\beta(r)$ is the shortest prefix of r , the weights of whose tokens add up to more than β . It can be easily shown: for any two sets r and s , if $w(r \cap s) \geq \alpha$, then $pref_{\beta_r}(r) \cap pref_{\beta_s}(s) \neq \emptyset$, where $\beta_r = w(r) - \alpha$ and $\beta_s = w(s) - \alpha$, respectively [4]. By taking $\alpha = overlap$, we can use prefix filtering with any similarity function. Prefix overlap is a condition necessary, but not sufficient to satisfy the original overlap constraint: an additional verification must be performed on the candidate pairs. Finally, the number of candidates can be reduced by using *document frequency ordering*, \mathcal{O}_{df} , as global token order to obtain sets ordered by increasing token frequency in the collection \mathcal{C} . The idea is to minimize the number of sets agreeing on prefix elements and, in turn, candidate pairs by shifting lower frequency tokens to the prefix positions.

EXAMPLE 1. Consider the weighted sets $r = \{\langle b, 9 \rangle, \langle c, 7 \rangle, \langle d, 7 \rangle, \langle e, 7 \rangle, \langle h, 5 \rangle, \langle i, 5 \rangle\}$ and $s = \{\langle a, 10 \rangle, \langle b, 9 \rangle, \langle c, 7 \rangle, \langle d, 7 \rangle, \langle e, 7 \rangle, \langle f, 5 \rangle, \langle h, 5 \rangle\}$ — note the token-weight association $\langle t, w(t) \rangle$ and assume that both sets are already sorted. We have $w(r) = 40$, $w(s) = 50$, and $w(r \cap s) = 35$. Considering Sim as the Jaccard similarity, we have $Sim(r, s) = \frac{w(r \cap s)}{w(r \cup s)} = \frac{35}{40 + 50 - 35} \approx 0.64$. For $\tau = 0.6$, we have $overlap(r, s) = 33.75$, $[minw(r), maxw(r)] = [24, 66.7]$, $pref_{\beta_r}(r) = \{\langle b, 9 \rangle\}$ and $pref_{\beta_s}(s) = \{\langle a, 10 \rangle, \langle b, 9 \rangle\}$.

Algorithm 1: General (self-) similarity join algorithm.

Input: A sorted set collection \mathcal{C} , a threshold τ
Output: A set S containing all pairs (r, s) s.t. $Sim(r, s) \geq \tau$

```

1  $I_1, I_2, \dots, I_{|\mathcal{U}|} \leftarrow \emptyset, S \leftarrow \emptyset$ 
2 foreach  $r \in \mathcal{C}$  do
3   foreach  $t \in pref_\beta(r)$  do
4     foreach  $s \in I_t$  do
5       if not  $filter(r, s)$ 
6          $S \leftarrow S \cup refine(r, s)$ 
7        $I_t \leftarrow I_t \cup \{r\}$ 
8 return  $S$ 

```

Similarity join algorithms based on inverted lists are effective in exploiting the above optimizations [13, 2, 16, 12]. Most of such algorithms have a common high-level structure following a filter-and-refine framework. Algorithm 1 formalizes the steps of this similarity join framework. An inverted list I_t stores all sets containing a token t in their prefix. The input collection \mathcal{C} is scanned and, for each set r , its prefix tokens are used to find candidate sets in the corresponding inverted lists (lines 2–4). Each candidate s is checked using filters based on overlap and weight bounds (line 5); if the candidate passes through, the actual similarity computation is performed in the refine step and r and s are added to the result if they are similar (line 6). Finally, r is appended to the inverted list (line 7). Algorithm 1 is actually a self-join. Its extension to a binary join is trivial: we first index the smaller collection and then go through the larger collection to identify matching pairs. Also, several optimization details such as positional filtering [16] and list reduction techniques [2, 12] were left unspecified. Nevertheless, these optimizations are mostly based on bounds and prefixes and, therefore, our discussion here remains valid. Finally, there is a pre-processing phase, where each set is sorted according to \mathcal{O}_{df} and \mathcal{C} is sorted in increasing order of the set weight. In this phase, we can collect statistics used to build our prediction model, which we describe in the following.

4. MODELING SIMILARITY JOINS

We now present our approach to model similarity joins. Before we outline the features used as input, we provide the rationale behind their choice. Our design decisions in this context were mostly guided by knowledge of the optimization techniques and algorithm details described in Section 3. In this context, we distinguish several performance drivers with major impact on algorithm runtime.

First, there is a clear correlation between threshold values and similarity join performance. Invariably, runtime decreases or increases following higher or lower threshold values, respectively. The reason is that higher threshold values imply larger overlap bounds, shorter prefixes, and tighter weight and size bounds. As a result, a larger number of candidate pairs are promptly discarded at earlier stages of processing or even not considered.

The size of the underlying dataset is an obvious factor affecting performance of similarity joins. In particular, it has been observed on several datasets that the candidate set and, in turn, the runtime grow quadratically with the dataset size [16]. Note, despite the optimizations discussed earlier, the output size of any join algorithm is still $\mathcal{O}(n^2)$ in the worst-case scenario.

Table 2: Features for the similarity join models.

Name	Description
<i>threshold</i>	Similarity join threshold value
<i>noSets</i>	# of sets
<i>ssCenter</i>	Mean set size
<i>ssGeoCenter</i>	Geometric mean of set size
<i>ssSpread</i>	Set size standard deviation
<i>ssSkew</i>	Set size skewness
<i>weiCenter</i>	Mean set weight
<i>weiSpread</i>	Set weight standard deviation
<i>weiSkew</i>	Set weights skewness
<i>tokfCenter</i>	Mean token frequency
<i>tokSpread</i>	Token frequency standard deviation
<i>tokSkew</i>	Token frequencies skewness
<i>tokF₀</i>	F_0 measure of the token frequencies
<i>tokF₂</i>	F_2 measure of the token frequencies

The frequency of tokens in the prefix largely dictates the number of candidates for a given set. The main motivation for adopting a global token order based on document frequency is to place rare tokens in the prefixes; as a result, inverted lists are shorter and there is much less prefix overlap between dissimilar sets, thereby decreasing the number of generated candidates. Of course, the effectiveness of the document frequency ordering totally depends on the underlying token frequency distribution. For a uniform distribution, it behaves not better than lexicographical order. Conversely, highly skewed distributions normally exhibit an increased number of low-frequency tokens that are shifted to prefix positions due to the document frequency ordering.

For unweighted sets, larger sets translate into larger prefixes and, therefore, more tokens are used to probe the index in the candidate generation phase. Moreover, larger subsets have to be processed in the verification phase. For weighted sets, prefix sizes are dictated by the set weights (along with the threshold value); the workload in the verification phase also depends on set weight, although set size is clearly a factor, too. Another important data characteristic is the degree of dispersion of set weights (sizes) as filtering based on weight bounds becomes substantially more effective as it is exploited by optimizations based on weight (size) bounds and the min-prefix principle.

Based on the above considerations, we designed features aiming at the characterization of threshold values, distributions of token frequency, as well as set size and weight. All features considered are shown in Table 2. Threshold and dataset size values directly represent features. For data distributions, we used general measures based on the distribution *moments*, namely, mean, standard deviation, and skewness; note, for set size, we use the geometric mean as alternative to the arithmetic mean. For token frequency distribution, we also calculated the F_2 and F_0 measures, which are given by the *frequency of moments* definition $F_k = \sum_{t \in \mathcal{U}} df(t)^k$, for $k = 0, 2$. F_2 represents an alternative measure for skewness, whereas F_0 gives the number of distinct tokens. Both unweighted and weighted datasets are represented by features for token frequency and set size distribution; weighted sets also have features for weight distribution. In this way, we ended up with 10 and 13 features for unweighted and weighted sets, respectively. An important observation is that all features are inexpensive to calculate and do not require storing all sample values.

We employ a feature selection strategy to identify the best-performing subset of features for each model. A few rules are applied to reduce the search space: the features *threshold* and *noSets* are used in all models; F_k features are always considered together and never in conjunction with token frequency skewness; and arithmetic and geometric mean are never used together. The resulting search space is thus quite moderate: 144 and 864 feature subsets for unweighted and weighted sets, respectively.

In this paper, we used *boosted M5 model trees* to build the predictive models proposed as depicted in Figure 1. Boosting methods combine multiple models by iteratively building a sequence of models that complement one another. At each stage, estimation errors for the training data are calculated and a new model is learnt to correct these errors without altering the previous models. M5 is a model tree learner introduced by [10]. Model trees store a linear regression model at each leaf that predicts the class value of instances reaching the leaf [14]—in contrast to regular regression trees, which typically store the average value of instances reaching the leaf. M5 stores linear models for each internal node, not only the leaves, for use in a smoothing process to reduce discontinuities between adjacent linear models at the leaves. We have also tested several techniques such as multi-layer perceptron and support vector machines [14], but both were outperformed by boosted model trees in our study.

5. EXPERIMENTS

5.1 Experimental Setup

We started with two well-known datasets: *DBLP* is a computer science bibliography (dblp.uni-trier.de) and *IMDB* contains information about movies (imdb.com). We randomly selected 50k–100k publications (movies) from DBLP (IMDB) in each data generation run and formed strings by concatenating titles with up to 10 authors (actors).

To increase the coverage of our training data, we generated new data from the original real-world datasets. In this context, datasets containing fuzzy duplicates were generated by creating exact copies of the original string and then performing transformations such as character insertion, deletion, or substitution. We generated datasets with the following percentage of duplicates: 0%, 50%, 25%, and 90%. Besides the number of duplicates, we also derived datasets with different token sizes by varying the value of q from 2 to 5. The skew of the resulting frequency distribution tends to increase with the token size. For example, the distribution for 5-grams is likely to exhibit higher skew than the one for 2-grams. We then performed 25 runs for each combination of number of duplicates and token size, thereby generating a collection of 400 semi-synthetic unweighted datasets from each source dataset; the same number of weighted datasets were derived from this collection. Note that the generation approach involves several random processes, namely, string transformations, sampling of source datasets, and selection of sample size. Thus, there are already significant differences even among datasets generated from the same sources.

We used an implementation of the *mpjoin* algorithm [12], a set similarity join algorithm following the general framework discussed in Section 3. We executed *mpjoin* with 5 different threshold values from 0.5 to 0.9 and collected the average runtime over repeated executions for each value. We end up with training (and test) data containing 2k instances

for each unweighted and weighted dataset collection. We used the implementation of the M5 model tree provided by the WEKA software package [7]. The fixed number of boosting iterations was 10 with a shrinkage rate of 1 and the minimum number of instances allowed at a leaf node was 4.

Prediction accuracy was measured using the *mean relative error* (MRE), which was the metric used in several previous studies [5, 1, 15]. It is defined as $MRE = \frac{1}{T} \times \sum_{Q \in TestSet} \frac{|pred(Q) - actual(Q)|}{actual(Q)}$, where T is the number of instances, and $pred(Q)$ and $actual(Q)$ are the predicted value and the actual value for the test instance Q , respectively.

We ran our experiments on two different hardware platforms. One is an Intel Xeon E3-1270 3.4 GHz, 8MB CPU cache, and about 8 GB of main memory; we call this platform *server*. The other is an Intel Q8400 2.66 GHz, 4MB CPU cache, and about 4 GB of main memory; we call this platform *client*. We used Java JDK 7 (Oracle) 1.7.0. We considered two types of experiments to evaluate our methods: cross-validation (10-fold cross-validation) based on a single training dataset and generalization based on the two datasets. This latter experiment type is more challenging and aims at evaluating the ability of our models in generalizing beyond the training data.

5.2 Prediction Results

After having described our experimental setup, we now present and analyze our prediction results. In our charts, we divide the results into those originated from short- (0-5 seconds), mid- (5-30 seconds), and long- (> 30 seconds) running queries. We report the MRE metric together with standard deviation as error bars.

Our first experiment evaluates the accuracy of our approach using cross-validation (CV) on the server platform. Figures 2(a) and 2(b) show prediction results on the DBLP and IMDB datasets, for unweighted and weighted sets, respectively. Our first observation is that prediction errors in all settings are at most 16%. This result is comparable to results reported in prior work for SQL queries (e.g., [1]). Indeed, practically all estimated runtimes are close to a perfect prediction. Accuracy significantly increases from short- to long-running queries. While prediction errors for short queries are always higher than 10%, accuracy for long queries stayed below 4%. This result is particularly important as long queries are more susceptible to cause user dissatisfaction as well as certain SLA violations, in particular, when a long query is predicted as a short one. Further, accuracy is better for weighted sets as compared to unweighted sets on both datasets. We achieved better accuracy on IMDB for unweighed sets; this trend is less noticeable for weighted sets (in fact, our models performed slightly poorer on IMDB for long queries).

Besides adopting feature selection to build our models, we conducted a sensitive analysis using several algorithms provided by WEKA to identify a dominant subset of features. While we found most results quite inconsistent along different datasets and algorithms, token frequency skewness appeared more frequently as the feature most correlated with the measured runtimes.

We next validate our prediction techniques on a different hardware configuration. Figures 2(c) and 2(d) present the results on the *client* platform for unweighed and weighted datasets, respectively. In general, the results are quite similar to those for the server platform: prediction errors are

at most 18% for short queries on unweighted sets and at most 7% for weighted sets. Again, most estimated runtimes are close to the perfect prediction and our prediction models performed considerably better for weighted sets. Our models achieved better accuracy for long-running queries as compared to short ones on unweighted sets, whereas there is little difference on weighted sets. Our results indicate that we can effectively predict similarity join runtimes regardless of the system environment.

We now present the results from generalization experiments where the prediction model was learned using a dataset and tested on other datasets with different characteristics. Dealing with differences between training and test data is a notoriously difficult problem independent of the underlying machine learning technique. Hence, we beforehand did not expect to obtain the same results as delivered by the previous experiments. Figures 2(e) and 2(f) show the results on the server platform for unweighed and weighted datasets, respectively. Although accuracy errors increased, such a deterioration is not dramatic, with relative errors always below 40% in all settings. Our models were more successful at generalizing from IMDB to DBLP. Also, the models tend to overestimation when trained with DBLP and underestimation when trained with IMDB, because the underlying sets of DBLP are larger as compared to IMDB.

6. RELATED WORK

Predicting the runtime of SQL queries and resource usage has been addressed both for a single query in isolation [6, 1, 15] as well as in the context of multiple concurrently running queries [5]. Statistical techniques that have been used in these works include *multivariate linear regression* [1], *Kernel Canonical Correlation Analysis* [6], *Support Vector Machines* [1]. In particular, a learning method based on *boosted regression trees* was successfully applied to resource usage prediction in [8]. Here, we used model trees, which can deal with non-linear dependencies between feature values and measured runtime values as regression trees, while delivering more compact models.

SQL queries can be modeled at the query template level or the operator level [1]. In the latter approach, two separate prediction models are built for each operator: a start-time model, used to estimate the time spent by an operator to produce its first tuple, and a runtime model, used to estimate total runtime of the operator. To integrate our model into an operator-level modeling approach, we would need to define a start-time and runtime model. For the former model, we only need to sum the start-time estimate of the similarity join’s input operator with the estimate of its own start-time estimate. For most set similarity join algorithms, start operations only involve the initialization of a few data structures such as the index and the score map. For the runtime model, we only need to add the input estimate with the runtime estimate of our model.

An alternative approach is to use the optimizer cost model directly to predict runtimes [15]. To this end, the work in [15] adjusts cost units offline using a set of carefully chosen calibrating queries and corrects the cardinality estimates of the query plan chosen by the optimizer through a sampling estimator. Calibration queries are designed to isolate a cost unit from the others. Cost units in our context could be the threshold value and distribution information. Instead of calibration queries, we could use dataset instances. Never-

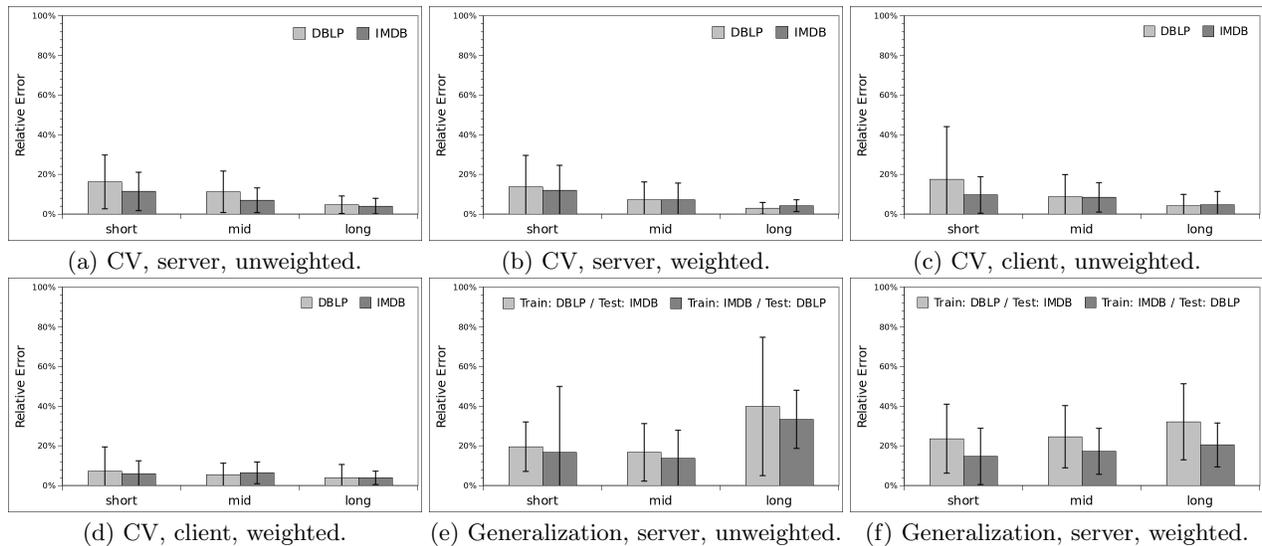


Figure 2: Prediction results for cross-validation and generalization experiments on different hardware platforms.

theless, it is unclear how to isolate the effects of threshold and token frequency distribution.

There is already a large body of work on set similarity joins [13, 4, 2, 11, 12, 16]. Aspects most relevant to our work have already been discussed in Section 3. Here, we only mention that any similarity join algorithm exploiting weight (and size) bounds and prefix filtering can be used in our performance prediction framework.

7. CONCLUSION

In this paper, we considered the challenging problem of modeling performance of set similarity joins. To the best of our knowledge, this problem has not been previously explored in the literature. We adopted an experiment-driven approach to build a performance model for a state-of-the-art similarity join algorithm. To this end, we identified simple statistics dictating effectiveness of well-known optimization techniques and used these statistics as input features to a machine learning technique. We also presented a framework that deploys our model to predict execution time of incoming similarity queries. Our experimental evaluation not only demonstrated the feasibility of predicting performance of complex similarity algorithms, but also often showed results comparable with previous work on traditional SQL queries.

8. ACKNOWLEDGMENTS

This research was partially supported by the Brazilian agencies CNPq (grant 487087/2012-7) and FAPEMIG (grant CEX APQ-00553-12).

9. REFERENCES

- [1] M. Akdere, U. Çetintemel, M. Riondato, and et al. Learning-based query performance modeling and prediction. In *ICDE*, pages 390–401, 2012.
- [2] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.
- [3] S. Chaudhuri. What next?: a half-dozen data management research goals for big data and the cloud. In *PODS*, pages 1–4, 2012.
- [4] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.
- [5] J. Duggan, U. Çetintemel, O. Papaemmanouil, and E. Upfal. Performance prediction for concurrent database workloads. In *SIGMOD*, 337–348, 2011.
- [6] A. Ganapathi, H. A. Kuno, and et al. et al. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *ICDE*, 592–603, 2009.
- [7] M. Hall, E. Frank, G. Holmes, B. Pfahringer, and et al. The WEKA data mining software: An update. *SIGKDD Explorations*, 11(1):10–18, 2009.
- [8] J. Li, A. C. König, V. R. Narasayya, and S. Chaudhuri. Robust estimation of resource consumption for SQL queries using statistical techniques. *PVLDB*, 5(11):1555–1566, 2012.
- [9] B. Mozafari, C. Curino, and S. Madden. Dbseer: Resource and performance prediction for building a next generation database cloud. In *CIDR*, 2013.
- [10] J. R. Quinlan. Learning with continuous classes. In *5th Australian Joint Conference on Artificial Intelligence*, pages 343–348, 1992.
- [11] L. Ribeiro and T. Härder. Efficient set similarity joins using min-prefixes. In *ADBIS*, pages 88–102, 2009.
- [12] L. A. Ribeiro and T. Härder. Generalizing prefix filtering to improve set similarity joins. *Information Systems*, 36(1):62–78, 2011.
- [13] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, 743–754, 2004.
- [14] I. H. Witten, E. Frank, and M. A. Hall. *Data mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 3rd edition, 2011.
- [15] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *ICDE*, pages 1081–1092, 2013.
- [16] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *ACM TODS*, 36(3):15, 2011.