Keyword++: A Framework to Improve Keyword Search Over Entity Databases

Venkatesh Ganti [*]
Google Inc
vganti@google.com

Yeye He [†]
University of Wisconsin
heyeye@cs.wisc.edu

Dong Xin
Microsoft Research
dongxin@microsoft.com

ABSTRACT

Keyword search over entity databases (*e.g.*, product, movie databases) is an important problem. Current techniques for keyword search on databases may often return *incomplete* and *imprecise* results. On the one hand, they either require that relevant entities contain all (or most) of the query keywords, or that relevant entities and the query keywords occur together in several documents from a known collection. Neither of these requirements may be satisfied for a number of user queries. Hence results for such queries are likely to be incomplete in that highly relevant entities may not be returned. On the other hand, although some returned entities contain all (or most) of the query keywords, the intention of the keywords in the query could be different from that in the entities. Therefore, the results could also be imprecise.

To remedy this problem, in this paper, we propose a general framework that can improve an existing search interface by translating a keyword query to a structured query. Specifically, we leverage the keyword to attribute value associations discovered in the results returned by the original search interface. We show empirically that the translated structured queries alleviate the above problems.

1. INTRODUCTION

Keyword search over databases is an important problem. Several prototype systems like DBXplorer [2], BANKS [4], DISCOVER [12] have pioneered the problem of keyword search over databases. These systems answer keyword queries efficiently and return tuples which contain all or most of the query keywords. Many vertical search engines, such as Amazon.com, Bing Shopping, Google Products, are driven by keyword search over databases. Typically, in those applications, the databases are *entity databases* where each tuple corresponds to a real world entity (*e.g.*, a product). The goal is to find related entities given search keywords.

Existing keyword search techniques over databases primarily return tuples (denoted as *filter set*) whose attribute values contain all (or most) query keywords. This approach might work satisfactorily in some cases, but still suffers from several limitations, especially in the context of keyword search over entity databases. Specifically, this keyword matching approach, (i) may not return all relevant results and (ii) may also return irrelevant results. The primary reason is that keywords in attribute values of many entities (i) may not contain all keywords, with which users query products, and (ii) may also contain matching keywords with irrelevant intention.

Consider a typical keyword search engine which returns products in the results *only* if all query keywords are present in a product tuple. For the query [rugged laptops] issued against the laptop database in Table 1, the filter set may be *incomplete*, as some laptop, which is actually a rugged laptop but does not contain the keyword "rugged", may not be returned. For instance, the laptop product with $ID = 004$ is relevant, since Panasonic ToughBook laptops are designed for rugged reliability, but not returned. As another example, consider the query [small IBM laptop] against the same database. The filter set may be *imprecise*, as some result, while containing all query keywords, may be irrelevant. For instance, the laptop product with $ID = 002$ contains all keywords, and thus is returned. However, the laptop is actually not small; and the keyword "small" in the product description does not match with user's intention.

Recently, several entity search engines, which return entities relevant to user queries even if query keywords do not match entity tuples, have been proposed [1, 3, 5, 8, 15]. These entity search engines rely on the entities being mentioned in the vicinity of query keywords across various documents. Consider the above two queries again. Many of the relevant products in the database may not be mentioned often in documents with the query keywords, and thus are not returned; if at all, a few popular laptops (but not necessarily relevant) might be mentioned across several documents. So, these techniques are likely to suffer from *incompleteness* and *impreciseness* in the query results.

In this paper, we address the above incompleteness and impreciseness issues under the context of *keyword search over entity search*. We *map* query keywords[1] to matching predicates or ordering clauses. The modified queries may be cast as either SQL queries to be processed by a typical database system or as keyword queries with additional

[*]Work done while at Microsoft
[†]Work done while at Microsoft

Proceedings of the VLDB Endowment, Vol. 3, No. 1

[1]We adopt IR terminology and use the term *keyword* to denote single words or phrases.

predicates or ordering clauses to be processed by a typical IR engine. In this paper, we primarily focus on translating keyword queries to SQL queries. But our techniques can be easily adopted by IR systems.

When a query keyword are very highly correlated with a categorical attribute, we map the keyword to a predicate of the form "Attribute value = <value>". When the correlated attribute is numeric, we may map the keyword to an ordering clause "order by <attribute> <ASC|DESC>". Most of the unmapped query keywords are still issued as keyword queries against the textual columns. For example, consider the query [rugged laptops] on laptop database. By mapping the keyword "rugged" to a predicate "ProductLine=ToughBook", we are likely to enable the retrieval of more relevant products. For the query [small IBM laptop], we may map the keyword "small" to an ordering clause that sorts the results by ScreenSize ascendingly.

In order to automatically translate a keyword to a SQL predicate, we propose a framework which leverages an existing keyword search engine (or, *the baseline search interface*). Basically, the baseline search interface takes the search keywords as input, and produces a list of (possibly noisy) entities as output, which is further used to mine the keyword to predicate mapping. In our experiments, we apply our framework on several baseline interface, and show that it consistently improves the precision-recall of the retrieved results. Note that we do not assume that we have access to the internal logic of the baseline interface, which is typically not publicly available, especially for production systems.

In this paper, we consider the entity database as a single relation, or a (materialized) view which involves joins over multiple base relations. Essentially, we assume that each tuple in the relation describes an entity and its attributes. This is often the case in many entity search tasks.

We now summarize our main contributions.
(1) We propose a general framework, which builds upon a given *baseline keyword search interface* over an entity database, to map keywords in a query to predicates or ordering clauses.
(2) We develop techniques that measure the correlation between a keyword and a predicate (or an ordering clause) by analyzing two result sets, from the baseline search engine, of a *differential query pair* with respect to the keyword.
(3) We improve the quality of keyword to predicate mapping by *aggregating* measured correlations over multiple differential query pairs discovered from the query log.
(4) We develop a system that efficiently and accurately translates an input keyword query to a SQL query. We materialize mappings for keywords observed in the query log. For a given keyword query at run time, we derive a query translation based on the materialized mappings.

The remainder of the paper is organized as follows. In Section 2, we formally define the problem. In Section 3, we discuss the algorithm for mapping keywords to predicates. In Section 4, we discuss the algorithm for translating a keyword query to SQL query. In Section 5, we present the experimental study. We conclude in Section 6. Related work and extensions of the proposed method are presented in Section 8.

2. PROBLEM DEFINITION

We use Table 1 as our running example to illustrate the problem and our definitions. In this table, each tuple corresponds to a model of laptop, with various categorical attributes (*e.g.*, "BrandName", "ProductLine"), and numerical attributes (*e.g.*, "ScreenSize", "Weight").

2.1 Preliminaries

2.1.1 Keyword To Predicate Mapping

Our goal is to translate keyword queries to SQL queries. When a user specifies a keyword query, each keyword may imply a predicate that reflects user's intention. Before we discuss the scope of the SQL statement considered in this paper, we first define the scope of the predicates, which includes categorical, textual, and numerical predicates.

Let E be the relation containing information about entities to be searched. Let $E^c \cup E^n \cup E^t$ be E's attributes, where $E^c = \{A_1^c, ...A_N^c\}$ are categorical attributes, $E^n = \{A_1^n, ..., A_N^n\}$ are numerical attributes, and $E^t = \{A_1^t, ..., A_T^t\}$ are textual attribute. The classification of attributes are not exclusive. For instance, a categorical attribute could also be a textual attribute, and a numerical attribute could also be a categorical attribute if we treat numerical values as strings. Denote $D(A) = \{v_1, v_2, \ldots, v_n\}$ the value domain of A. Note although value domain of numerical attributes could be continuous, the values appearing in database are finite and enumerable. Let *tok* be a tokenization function which returns the set $tok(s)$ of tokens in a given string s.

We denote a keyword query by $Q = [t_1, t_2, \ldots, t_q]$, where t_i are single tokens. Let $k = t_i, t_{i+1}, ..., t_j$ be a token, or a multi-token phrase. Following standard conventions in Information Retrieval literature, we use *keyword* to denote both single token and multi-token phrase.

DEFINITION 1. *(***Predicate***) Let $e \in E$ be an entity. Define the categorical predicate $\sigma_{A=v}(e)$, where $A \in E^c$, to be* true *if $e[A] = v$. Define the textual predicate $\sigma_{Contains(A,k)}(e)$, where $A \in E^t$, to be* true *if $tok(k) \subseteq tok(e[A])$. Define the null predicate σ_{TRUE}, which simply returns true.*

We can apply the predicates on a set S_e of entities to return a subset $\sigma(S_e)$ containing only entities for which the predicate is true. Because S_e is clear from the context, we loosely use the notation $\sigma_{A=v}$ and $\sigma_{Contains(A,t)}$ without referring to the entity set S_e these are applied to.

Let $\sigma_{(A,SO)}(S_e)$, where $A \in E^n$ and $SO \in \{ASC, DESC\}$, denote the list of entities in S_e sorted in the ascending order if $SO = ASC$ or in the descending order if $SO = DESC$ of the attribute values in A. For clarity and uniformity in exposition, we abuse the terminology numerical predicate and notation $\sigma_{(A,SO)}$.[2]

We map a selected set of keywords to categorical, textual, numerical or null predicates. Let $\{\sigma_{A=v}\} \cup \{\sigma_{Contains(A,t)}\} \cup \{\sigma_{(A,SO)}\} \cup \{\sigma_{TRUE}\}$ denote the set of all such predicates over the entity relation E.

DEFINITION 2. *(***Mapping***) Let k be a keyword in the keyword query Q over table E, define the keyword-predicate mapping function $Map(k, \sigma) : [0, +\infty]$, where $\sigma \in \{\sigma_{A=v}\} \cup \{\sigma_{Contains(A,k)}\} \cup \{\sigma_{(A,SO)}\} \cup \{\sigma_{TRUE}\}$, and $Map(k, \sigma)$ returns a confidence score for mapping k to σ. Define the best predicate $M_\sigma(k) = argmax_\sigma Map(k, \sigma)$, and the corresponding confidence score $M_s(k) = max_\sigma Map(k, \sigma)$.*

[2]In Section 8.4, we describe a mechanism to optionally map keywords to range predicates on numeric attributes instead of ordering clauses.

Table 1: The laptop product table

ID	ProductName	BrandName	ProductLine	ScreenSize	Weight	ProductDescription	...
001	Panasonic Toughbook W8 - Core 2 Duo SU9300 1.2 GHz - 12.1" TFT	Panasonic	Toughbook	12.1	3.1	If you want a fully rugged, lightweight ...	...
002	Lenovo ThinkPad T60 2008 - Core Duo T2500 2 GHz - 14.1" TFT, 80GB HD	Lenovo	ThinkPad	14.1	5.5	The IBM laptop...small business support	...
003	ThinkPad X40 2372 - Pentium M 1.2 GHz - 12.1" TFT, 40GB HD, 512MB RAM	Lenovo	ThinkPad	12.1	3.1	The thin, light, ThinkPad X40 notebook ...	...
004	Panasonic Toughbook 30 - Core 2 Duo SL9300 1.6 GHz - 13.3" TFT	Panasonic	Toughbook	13.3	8.4	the durable Panasonic Toughbook 30 ...	...

In this paper, we map a keyword to the predicate with the maximal confidence score (as defined by $M_\sigma(k)$ and $M_s(k)$). Our method can be extended to map a keyword to multiple predicates, and we will discuss it in Section 8.4.1. We now illustrate the four types of mappings using examples.
(1) Mapping from a keyword to a categorical predicate: For instance, the keyword "IBM" in query [small IBM refurbished laptop] semantically refers to the attribute value "Lenovo" in column *BrandName* (Table 1), due to the fact that IBM laptop production was acquired by Lenovo. Therefore, we have $M_\sigma(\text{"}IBM\text{"}) = \sigma(BrandName = \text{"}Lenovo\text{"})$.
(2) Mapping from a keyword to a numerical predicate: Using the query [small IBM refurbished laptop] again, the keyword "small" could be associated with the predicate $M_\sigma(\text{"}small\text{"}) = \sigma(ScreenSize, ASC)$.
(3) Mapping from a keyword to a textual predicate: Some keywords may only appear in textual attributes such as *ProductName* or *Description*. Although there is no corresponding categorical attributes, they are also valuable to filter the results. For instance, we can associate "refurbished" with a textual predicate $\sigma(Contains(ProductName, \text{"}refurbished\text{"}))$.
(4) In some cases there is no meaning of the keyword. Many stop words seen in queries ("the", "and", "a", *etc*), all fall in this category. We assign $M_\sigma(\text{"the"}) = \sigma_{TRUE}$.

2.1.2 Query Translation

Once the keyword to predicate mappings are established, we propose to translate a keyword query to a SQL query. The scope of SQL statements in this paper is confined to the *CNF SQL* queries.

Definition 3. *A CNF SQL has the format:*
*SELECT * FROM Table*
WHERE $cnf(\sigma_{A=v})$ *AND* $cnf(\sigma_{Contains(A,t)})$
ORDER BY $\{\sigma_{(A,SO)}\}$
where $cnf(\sigma_{A=v})$ *and* $cnf(\sigma_{Contains(A,t)})$ *are conjunctive forms of multiple categorical and textual predicates, and* $\{\sigma_{(A,SO)}\}$ *is an ordered list of numerical predicates.*

Using CNF, we can uniquely rewrite a set of predicates (null predicates are ignored) to a SQL query. Therefore,we use a set of predicates to represent a CNF SQL thereafter.

Given a keyword query, there are multiple ways to translate it to a CNF query. We now define the "best" query translation based on the notion of *keyword segmentation* and *translation scoring*. Informally, we sequentially segment Q into multiple keywords, where each keyword contains at most n tokens (typically, $n = 2$ or 3). We map each segmentation to a set of predicates, based on the keyword to predicate mapping. Therefore, each segmentation maps to a CNF query. We define the confidence score for the query translation to be an aggregate of keyword to predicate mapping scores. Among all possible segmentations, the one with the highest confidence score defines the best query translation.

Definition 4. (**Query Translation**) *Given a keyword query* $Q = [t_1, \ldots, t_q]$, *we define* $G = \{k_1, k_2, \ldots, k_g\}$ *as a* n*-segmentation of query* Q, *where* $k_i = t_{d_i+1}, \ldots, t_{d_{i+1}}$, $0 = d_1 \le d_2 \le \ldots \le d_{g+1} = q$, *and* $\max(d_{i+1} - d_i) = n$. *For each segmentation* G, *denote* $f_\sigma(G) = \{M_\sigma(k_1), \ldots, M_\sigma(k_g)\}$ *the set of mapped predicates from each keyword. Denote* $f_s(G) = \sum_{i=1}^{g} M_s(k_i)$ *the translation score of the segmentation* G. *Let* $\mathcal{G}_n(Q)$ *be the set of all possible* n*-segmentations. The best SQL translation of* Q *is*

$$T_\sigma(Q) = f_\sigma(argmax_{G \in \mathcal{G}_n(Q)} f_s(G))$$

2.2 System Architecture

The architecture of our proposed system (Figure 1) has an *offline* (the upper half) and an *online* (the lower half) component.
Offline Component: We exploit a set of *historical queries*[3] (without click information) in the domain of entity search. Given an entity category, the vocabulary in user queries is often limited. Therefore, it is sufficient to materialize the mappings for all keywords that appear in the historical queries. Note that even for a new query which does not appear in the historical set, the chance that all its keyword mappings have been pre-computed is still quite high. The mappings for all keywords are materialized in the *mapping database*.
Online Component: Given a keyword query Q, we search the mapping database for the mappings of keywords in Q. We use these mapping predicates to rewrite Q into a CNF SQL query. In the case that Q contains a keyword whose mapping is not pre-computed, we can optionally compute the mapping online (represented by the dashed line in the figure, see Section 8.5.2 for the related discussion). In reality, we observe that if a keyword does not appear in query log, it is less likely that the keyword has semantic meaning (*e.g.*, mapping to a categorical or a numerical predicate). Therefore, a practical and efficient approach is to simply map it to a textual or null predicate, without going to the mapping analysis component at online query processing. We discuss the details in the following sections.

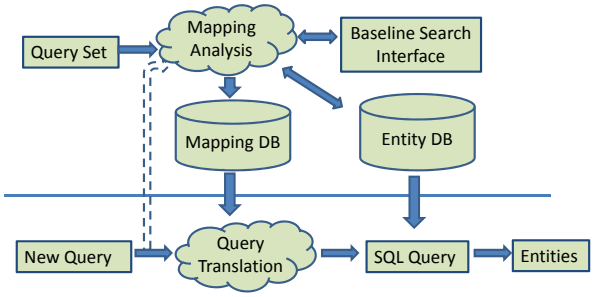

Figure 1: System architecture

[3]In Section 3.2 and Section 8.4.2, we discuss how to handle the case where a set of historical queries is not available.

2.3 Problem Statement

We formalize the problem below.

DEFINITION 5. *(**Problem Definition**) Given a search interface $\mathcal{S}$ over an entity relation E, a set of historical queries $\mathcal{Q}$, the query translation problem has two sub-problems: (1) For each keyword k in $\mathcal{Q}$, find its mapping $M_\sigma(k)$ and the confidence score $M_s(k)$ for the mapping (Definition 2); (2) Using the mapping M, compute the best CNF SQL query $T_\sigma(Q)$ for a keyword query Q (Definition 4).*

3. MAPPING KEYWORDS TO PREDICATES

In this section, we discuss our techniques to find the mapping between keywords and predicates. In Section 8.4, we discuss more related issues and extensions.

Ideally, in the set of entities returned by the baseline interface, the mappings determined by the keyword query should dominate the results and be obvious to tell. For instance, in a perfect setting most of the entities returned for query [small IBM laptop] should be indeed of *BrandName* "Lenovo" and with *ScreenSize* as "small" as possible. This however, is very hard to guarantee in practice (even for production search engines) precisely due to the imperfect nature of the search engine as discussed earlier.

Therefore, instead of hoping for a good baseline keyword search engine, we propose to use a *differential query pair* (or, DQP). Intuitively, the DQP approach uses statistical difference aggregated over several selectively chosen query pairs to find the mappings from keywords to predicates. In the remainder of this section, we first define differential query pairs, and then show how aggregation over multiple differential query pairs allows us to derive accurate mappings.

3.1 Differential Query Pair

DEFINITION 6. *(**Differential Query Pair**) A pair of keyword queries $Q_f = [t_1^f, ..., t_m^f]$ and $Q_b = [t_1^b, ..., t_n^b]$, where t_i^f, t_j^b are tokens, is said to be a* differential query pair (Q_f, Q_b) *with respect to a keyword k, if $Q_f = Q_b \cup \{k\}$* [4].

We name Q_f the foreground query, Q_b the background query, and k the differential keyword.

In other words, a differential query pair is a pair of queries that differs only by the keyword under consideration. Some example are shown in Table 2.

Intuitively, the results returned by a differential query pair should differ most on the attribute mapping to the differential keyword. Let us illustrate this using the example below.

EXAMPLE 1. *Consider a keyword "IBM", and Q_f=[small IBM laptop] and Q_b=[small laptop]. For Q_b, suppose a search interface returns 20 laptop models, out of which, 3 are of brand "Lenovo", 7 "Dell", 6 "HP", 2 "Sony" and 2 "Asus". For Q_f, suppose 10 laptops are returned, of which 5 are "Lenovo", 2 "Dell", 1 "HP", 1 "Sony" and 1 "Asus". We then compare, attribute by attribute, the results of Q_b and Q_f. In this example, on this attribute* BrandName, *we can clear see that value "Lenovo" in the Q_f (5 out of 10) has the biggest relative increase from the Q_b (3 out of 20). Therefore, we conclude that keyword "IBM" might have some association with* BrandName *"Lenovo". The exact definition of the relative increase will be clear in Definition 7 and Definition 8.*

The insight here is that even though the desired results, in this case laptops with *BrandName* "Lenovo" may not dominate the result set for the foreground query [small IBM laptop], if one compares the results returned by the foreground and background queries, there should be a noticeable statistical difference for the attribute (value) that is truly mapped by the differential keyword.

Our algorithm of course does not know that "IBM" corresponds to the attribute *BrandName*; so all other attributes in the table are also analyzed in the same manner. We discuss the algorithm later in this section.

3.2 Generating DQPs for Keywords

We now discuss the generation of DQPs for keywords. Given a query Q and a keyword $k \in Q$, we can derive differential query pairs from Q by assigning $Q_f = Q$ and $Q_b = Q - \{k\}$. For instance, if Q = [*small IBM laptop*], and k ="small", we will generate Q_f = [*small IBM laptop*] and Q_b = [*IBM laptop*].

When a set of historical queries $\mathcal{Q}$ is available, we find more differential query pairs for a keyword k as follows. Given k, we generate the differential query pairs by retrieving all query pairs Q_f and Q_b, such that $Q_f \in \mathcal{Q}$, $Q_b \in \mathcal{Q}$ and $Q_f = Q_b \cup \{k\}$. We aggregate the scores of keyword to predicate mappings across multiple differential query pairs. We will show in Section 3.3.2 that aggregation significantly improves the accuracy of keyword to predicate mappings.

When the query log is not present or is not rich enough, one may exploit other sources. First, one can consider all entity names in the database as queries. Secondly, one may extract wikipedia page titles and category names in the related domain, as queries. Finally, one may identify a set of related web documents (*e.g.*, by sending entity names to web search engine, and retrieving top ranked documents), and extract titles and anchor text of those documents, as queries.

3.3 Scoring Predicates Using DQPs

We now discuss the confidence score computation for a predicate given a keyword, based on DQPs. We first discuss how to score categorical and numerical predicates by measuring the statistical difference w.r.t. some attribute/value between the results from foreground and background queries. Any statistical difference measure can be used, but we find in practice that *KL-divergence* [13] and *Earth Mover's Distance* [14, 18] works best for categorical and numerical attributes, respectively. Both of these two are very common metrics. We then discuss how to extend the scoring framework to textual and null predicates.

3.3.1 Correlation Metrics

Let $D(A)$ be the value domain of A. For any $v \in D(A)$, let $p(v, A, S_e) = \frac{|\{e[A]=v, e \in S_e\}|}{|S_e|}$ be the probability of attribute value v appearing in objects in S_e on attribute A. Let $P(A, S_e)$ be the distribution of $p(v, A, S_e)$ for all $v \in D(A)$. Given a differential query pair (Q_f, Q_b), let S_f and S_b be the set of results returned by the search interface.

We first discuss the KL-divergence for categorical attributes. The KL-divergence in terms of the foreground and background queries is defined as follows.

[4]Note Q_f, Q_b and k are all sequences. $Q_f = Q_b \cup \{k\}$ means that Q_f can be obtained by inserting k at any place of Q_b. For simplicity, we use the set operator $\cup$.

Table 2: Example differential query pairs

Differential keyword	Foreground query	Background query
IBM	[small refurbished IBM laptop]	[small refurbished laptop]
IBM	[lightweight IBM laptop]	[lightweight laptop]
small	[small refurbished IBM laptop]	[refurbished IBM laptop]
small	[small HP laptop]	[HP laptop]

DEFINITION 7. *(**KL-divergence**[13]) Given S_f and S_b, the KL divergence between $p(v, A, S_f)$ and $p(v, A, S_b)$, with respect to an attribute A, is defined to be*

$$KL(p(v, A, S_f)||p(v, A, S_b)) = p(v, A, S_f) log \frac{p(v, A, S_f)}{p(v, A, S_b)}$$

We apply some standard smoothing techniques if $p(v, A, S_b)$ is zero for some v. Given (Q_f, Q_b), we define the score for categorical predicates as follows:

$$Score_{kl}(\sigma(A = v)|Q_f, Q_b) = KL(p(v, A, S_f)||p(v, A, S_b))$$

EXAMPLE 2. *In Example 1, considering the results of Q_f, we see that a tuple has 0.5 probability of bearing brand "Lenovo" (5 out of 10), while the same probability is 0.3 for Q_b (6 out of 20). By applying Definition 7, we have the statistical difference of value "Lenovo" to be $0.5 \times \log_2 \frac{0.5}{0.3} = 0.368$. Similarly the same measure can be computed for all other values: -0.161 for "Dell", -0.158 for "HP", -0.1 for "Sony", and -0.1 for "Asus". Only looking at this query pair and attribute* BrandName*, "IBM" is more likely to map to "Lenovo" than any other brands.*

We now discuss the earth mover's distance for scoring numerical predicates.

DEFINITION 8. *(**Earth Mover's Distance**[14, 18]) Let S_f and S_b be two sets of objects, and let $D(A) = \{v_1, v_2, \ldots, v_n\}$ be the sorted values on a numerical attribute A. The Earth Mover's Distance (EMD) is defined w.r.t. an optimal flow $F = (f_{ij})$, which optimizes*
$W(P(A, S_f), P(A, S_b), F) = \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{ij}$,
where $d_{ij} = d(v_i, v_j)$ is some measure of dissimilarity between v_i and v_j, e.g., the Euclidean distance. The flow (f_{ij}) must satisfy the following constraints:
$f_{ij} \geq 0, 1 \leq i \leq n, 1 \leq j \leq n$
$\sum_{j=1}^{n} f_{ij} \leq p(v_i, A, S_f), 1 \leq i \leq n$
$\sum_{i=1}^{n} f_{ij} \leq p(v_j, A, S_b), 1 \leq j \leq n$
Once the optimal flow f_{ij}^* is found, the Earth Mover's Distance between P and Q is defined as

$$EMD(P(A, S_f), P(A, S_b)) = \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij}^* d_{ij}$$

Intuitively, given two distributions over a numerical domain, one distribution can be seen as a mass of earth properly spread in space according to its probability density function, while the other as a collection of holes in that same space. Then, the EMD measures the least amount of work needed to fill the holes with earth (where a unit of work corresponds to transporting a unit of earth by a unit of ground distance) [14]. So the smaller the EMD, the closer two distributions are. The definition of EMD naturally captures the locality of data points in the numerical domain, is thus ideal to measure difference of numerical distributions. Furthermore, positive EMD difference from $P(A, S_f)$ to $P(A, S_b)$ indicates the correlation with smaller values (or ascending sorting preference). Therefore, we define the score for numerical predicates as follows:

$$Score_{emd}(\sigma_{(A,SO)}|Q_f, Q_b) = \begin{cases} EMD(P(A, S_f), P(A, S_b)) & if \quad SO = ASC \\ -EMD(P(A, S_f), P(A, S_b)) & if \quad SO = DESC \end{cases}$$

The EMD can be computed based on the solution of the transportation problem [11], and in our one-dimensional case, it can be efficiently computed by scanning sorted data points and keeping track of how much earth should be moved between neighboring data points. We will skip the details here.

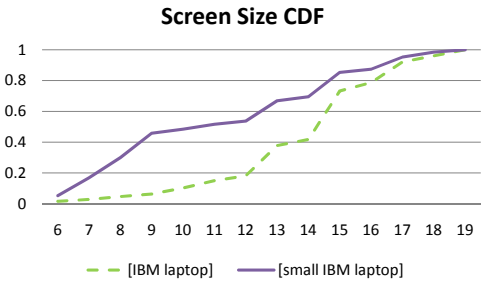

Figure 2: Cumulative Distributions

EXAMPLE 3. *Consider the keyword "small", and Q_f=[small IBM laptop] and Q_b=[IBM laptop]. We plot the cumulative distribution for results of both Q_f and Q_b in Figure 2, where x-axis is the ScreenSize, and y-axis is the cumulative probability. We see that the CDF moves considerably upward from query [IBM laptop] to [small IBM laptop] (positive EMD difference from Q_f to Q_b), indicating the differential keyword "small" has correlation with smaller ScreenSize (or ascending sorting preference).*

3.3.2 Score Aggregation

While the differential query pair approach alleviates the problem of noisy output of the baseline interface, sometimes there are random fluctuations in distributions w.r.t. one differential query pair. We first illustrate the random fluctuation in an example, and then discuss our solution.

EXAMPLE 4. *Using the same settings as Example 1, we may notice another significant statistical difference on the attribute* ProcessorManufacturer*. For instance, out of the 20 laptops returned for the background query [small laptop], 3 have the value "AMD" for this attribute, and remaining 17 have "Intel"; while in the 10 laptops returned for the foreground query [small IBM laptop], 6 have "AMD" and 4 have "Intel". Now depending on how we measure statistical difference, we may find that the difference observed on attribute* ProcessorManufacturer *is larger than that on the attribute* BrandName*. Therefore, if we only look at this query, one may reach the wrong conclusion that keyword "IBM" maps to attribute* ProcessorManufacturer *with value "AMD".*

In order to overcome the random fluctuations in distributions with respect to one differential query pair, we "aggregate" the differences among distributions across multiple

differential query pairs. The idea here is that for a particular differential query pair with respect to "IBM", there may be a significant statistical difference on attribute *ProcessorManufacturer* (many more "AMD" in the foreground query). However, we may observe that for many other differential query pairs with respect to "IBM", there is not much increase of "AMD" from foreground queries to background queries, or we may even see less "AMD" from foreground queries. Now if we aggregate the statistical difference over many differential query pairs, we obtain a more robust measure of keyword to predicate mapping.

DEFINITION 9. *(***Score Aggregation***) Given a keyword k, and a set of differential query pairs $\{(Q_f^1, Q_b^1), \ldots, (Q_f^n, Q_b^n)\}$, each of which with respect to k. Define the aggregate score for keyword k with respect to a predicate σ as:*

$$AggScore(\sigma|k) = \frac{1}{n}\sum_{i=1}^{n} Score(\sigma|(Q_f^i, Q_b^i))$$

where Score can be either $Score_{kl}$ or $Score_{emd}$.

3.3.3 Score Thresholding

Categorical and Numerical Predicates: The aggregated correlation scores are compared with a threshold. A mapping is kept if the aggregate score is above the threshold. As we discussed above, the keyword to predicate mapping is more robust with more differential query pairs. Intuitively, for those keywords with a small number of differential query pairs, we need to set a high threshold to ensure a high statistical significance as they tend to have high variations in the aggregated scores. The threshold value can be lower when we have more the number of differential query pairs.

Setting up thresholds with multiple criteria has been studied in the literature. In this paper, we adopt a solution which is similar to that in [6]. The main idea is to consider each keyword to predicate mapping as a point in a two-dimensional space, where the x-axis is the aggregate score and y-axis is the number of differential query pairs. A set of positive and negative samples are provided. The method searches for an optimal set of thresholding points (*e.g.*, 5 points) in the space, such that among the points which are at the top-right of any of these thresholding points, as many (or less) as positive (or negative) samples are included. With that approach, we generate a set of thresholds, each of which corresponds to a range with respect to the number of differential query pairs.

Note that we have two different metrics for categorical and numerical predicates. Hence, the scores are not comparable to each other. Therefore, it is necessary to set up a threshold for each of them separately. Furthermore, after score thresholding, we normalize scores as follows. For each score s which is above the threshold θ (with respect to the corresponding number of differential query pairs), we update a relative score by $s = \frac{s}{\theta}$.

Textual and Null Predicates: For each keyword k, we will compute the score for all possible categorical and numerical predicates. If none of these predicate whose score passes the threshold, then the keyword is not strongly correlated with any categorical or numeric predicates. Therefore, we assign textual or null predicate to k. Specifically, for any keyword k that does not appear in the textual attributes in the relation, or belongs to a stop word list, we assign a null predicate σ_{TRUE} with score 0 to k. Otherwise, we assign a textual predicate $\sigma_{Contains(A,k)}$ with score 0 if k appears in a textual attribute A.

3.4 The Algorithm

We now describe our algorithm for generating mappings from keywords to predicates. For all (or, a subset of frequent) keywords that appear in query log, we generate the mapping predicates using the baseline search interface. We first generate the differential query pairs (by leveraging the query log) for each keyword as discussed in Section 3.2. We then generate the relevant mapping predicate for the keyword, following the procedures in Section 3.3. Specifically, for each candidate predicate, we compute correlation score from each DQP, aggregate scores over all DQPs, and output a predicate by score thresholding. The detailed algorithm and its complexity analysis can be found in 8.2. The output of the algorithm is the predicate mapping $M_\sigma(k)$ and corresponding confidence score $M_s(k)$, for keyword k.

Note that we may invoke the baseline search interface for many (differential query pair) queries with respect to a keyword. Oftentimes, same query may be issued again for different keywords. We cache these results to avoid invoking the search interface for the same query repeatedly.

4. QUERY TRANSLATION

As we discussed earlier in Section 2.1.2, the query translation consists of two steps: *keyword segmentation* and *translation scoring*. Conceptually, the query segmentation step sequentially splits the query tokens into keywords, where each keyword is up to n tokens. For each such segmentation, the translation scoring step finds the best predicate mapping for each keyword, and computes the overall score for a target SQL rewriting.

A naive implementation of the above two steps would explicitly enumerate all possible keyword segmentations, and then score the SQL rewriting for each segmentation. Since the segmentation is conducted sequentially, a dynamic programming strategy can be applied here to reduce the complexity. Specifically, given a query $Q = [t_1, t_2, \ldots, t_q]$, let $Q_i = [t_1, \ldots, t_i]$ be the prefix of Q with i tokens. Recall that $T_\sigma(Q_i)$ is the best SQL rewriting of Q_i (see Definition 4). Let $T_s(Q_i)$ be the score of the best SQL rewriting of Q_i. $M_s(k)$ (Definition 2) is materialized offline and stored in a database. For new keywords whose predicate mapping were not found in the database, we consider them as textual predicates.

Suppose we consider up to n grams in segmentation (*e.g.*, each keyword has at most n tokens), we have

$$T_s(Q_i) = \max_{j=1}^{n}(T_s(Q_{i-j}) + M_s(\{t_{i+1-j}, \ldots, t_i\})$$

We demonstrate the recursive function as follows.

EXAMPLE 5. *Suppose $Q = [small\ IBM\ laptop]$ and $n = 2$ (i.e., up to 2-grams in segmentation). We start with $Q_1 = [small]$, and figure out $T_s(Q_1) = M_s(\text{"small"})$. We then move to $Q_2 = [small\ IBM]$. We have two options to rewrite Q_2 to a SQL: first, considering "IBM" as a new segment, we can rewrite Q_2 as $T_s(Q_2) = T_s(Q_1) + M_s(\text{"IBM"})$; secondly, considering "small IBM" as a segment, we can rewrite Q_2 as $T_s(Q_2) = M_s(\text{"small IBM"})$. We compare the two options, and pick the one with higher score for Q_2 rewriting. We then move to Q_3, and so on so forth.*

The pseudocode of the translation algorithm is outlined in Section 8.3. Note that the recursive function considers all possible keyword segmentations. Many such segments are actually not semantically meaningful. We also discuss how to incorporate semantical segmentation in Section 8.3.

5. PERFORMANCE STUDY

We now evaluate the techniques described in this paper. The experiments are conducted with respect to three different aspects: coverage on queries, computational overhead and mapping quality. To begin with, Table 3 shows some example predicate mappings discovered by our method.

Table 3: Example Keyword to Predicate Mapping

Keyword	Predicate
Rugged	ProductLine=ToughBook
lightweight	Weight ASC
mini	ScreenSize ASC

5.1 Experimental Setting

Data: We conduct experiments with real data sets. The entity table is a collection of $8k$ laptops. The table has 28 categorical attributes and 7 numerical attributes. We use *ProductName* and *ProductDescription* as textual attributes. We use $100k$ web search queries which were classified as laptop queries[5], from which, 500 queries are sampled and separated as a test set.

Comparison Methods: We experiment two baseline search interface. The first search interface is the commonly used *keyword-and* approach, which returns entities containing all query tokens. The second search interface is the *query-portal* approach [1], where queries are submitted to a production web search engine, and entities appearing in relevant documents are extracted and returned. These two baseline interface are two representative approaches of entity search over database. We denote our approach as *keyword++*.

In addition to the baseline search interface, we compare our method with two other approaches that could be possibly built using the baseline interface. The first, *bounding-box* approach, finds the minimum hyper-rectangular box that bounds all entities returned by the baseline search interface, and augments the results with other entities within the box. The second, *decision-tree* approach, constructs a decision tree by considering entities returned by the baseline interface as positive samples, and those not as negative samples. Each node in the decision tree corresponds to a predicate, and one can translate a decision tree to a SQL statement. Note that decision-tree based approach is a variation of the method discussed in [21]. The detailed description of all methods can be found in Appendix (Section 8.5).

Evaluation Metric: We evaluate the mapping quality at both the query level and keyword level. At the query level, we examine how close the set of results produced using different techniques is, when comparing with the ground truth, which are manually labeled for all queries in the test set. Given a result set R and a true set T, the evaluation metric includes $Precision = \frac{R \cap T}{R}$, $Recall = \frac{R \cap T}{T}$, and $Jaccard = \frac{R \cap T}{R \cup T}$. At the keyword level, we examine for a set of popular keywords, how many mappings are correct.

[5]The query classification is beyond the scope of this paper. Therefore, we omit the detailed description here

5.2 Query Coverage

We extract 2000 most frequent keywords (with one or two tokens), and manually label the corresponding predicates. Out of the top 2000 keywords, 218 categorical mappings and 20 numerical mappings are identified by domain experts. We refer the subset of keywords that map to categorical or numerical predicates as *semantical keywords*. We examine the coverage of those semantical keywords in the test queries.

We found 76.7% of test queries having at least one semantical keyword. Note that although a semantical keyword maps to a predicate, replacing the keyword by a predicate may not necessarily change the search results. For instance, we may identify "Dell" is a BrandName. If "Dell" appears (and only appears) in the ProductName of all Dell laptops, then the search results of a query [Dell laptop] by *keyword++* will be same as those of *keyword-and*. We are really interested in the number of queries whose results are improved by identifying the semantical keywords. We found 39.4% of the test queries whose results are improved. We examine the quality of the results in Section 5.4.

5.3 Computational Performance

We conduct experiments on a PC with 2.4 GHZ intel Core 2 Duo and 4GB memory. The entity table is stored in SQL server. For each of the extracted 2000 keywords, we identify DQPs from the query log. The maximum number of DQP for a keyword is 2535, and the minimum number of DQP for a keyword is 9. On average, each keyword has 41 DQPs, and it takes $1.61s$ to compute the mapping. The online query translation and query execution is very efficient. For each test query, it takes $6.6ms$ on average.

5.4 Retrieval Quality

Here we report the experimental results on mapping qualities, in terms of both query and keyword level evaluation. More detailed experiments that examine the effects of mapping predicates, multi-DQP aggregation, and multiple interface combination, are reported in Appendix (Section 8.5).

Figure 3 and Figure 4 shows the Jaccard, precision and recall scores, using *query-portal* and *keyword-and* as baseline interface, respectively. We have the following observations. First, *keyword++* consistently improves both baseline interface, despite the fact that *query-portal* and *keyword-and* use different mechanisms to answer user queries.

Secondly, the precision-recall of the baseline *query-portal* is worse than that of the baseline *keyword-and*, because the results generated by *query-portal* are indirectly drawn from related web documents. However, after applying *keyword++*, *query-portal* outperforms *keyword-and*. This is because many test queries contain keywords that do not appear in the database. As a result, *keyword-and* is not able to interpret those keywords. On the other hand, *query-portal* leverages the knowledge from the web search engine. Therefore, it captures the semantic meaning of keywords, which in turn is effectively identified and aggregated by *keyword++*.

Finally, comparing with *keyword++*, both *bounding-box* and *decision-tree* use the results by the submitted keyword query itself (See Section 8.5.1). When the baseline search interface returns noisy results, both *bounding-box* and *decision-tree* may not be able to correctly interpret the meaning of keywords. For instance, *query-portal*'s results contain more noise. As a result, the precision of *bounding-box* and *decision-tree* is even worse than that of the baseline *query-*

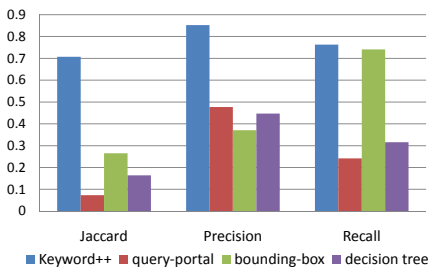

Figure 3: Jaccard, Precision and Recall on Query-Portal

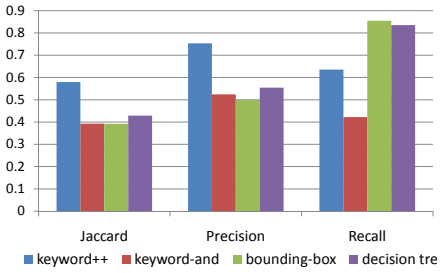

Figure 4: Jaccard, Precision and Recall on Keyword-And

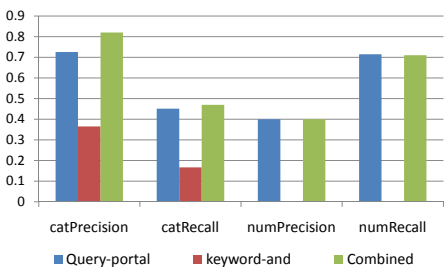

Figure 5: Precision and Recall on Keyword to Predicate Mapping

portal. On the other hand, *keyword++* achieves consistent improvement over both baseline search interface.

For keyword level evaluation (Figure 5), we only report precision-recall on semantical keywords. This is because the non-semantical keywords, which map to textual or null predicates, are easy to predict. Including the non-semantical keywords may easily push precision-recall to $> 95\%$. We examine categorical and numerical predicates w.r.t. *query-portal*, *keyword-and*, and a *combined* schema that merges predicates from the two baseline interface (See detailed description in Section 8.5.2).

We observe that for categorical predicates, when we combine results from the two baseline interfaces, the precisions reaches more than 80%. For numerical predicates, interestingly, with *keyword-and*, we are not able to discover numerical predicates. This is because most keywords that map to numerical predicate does not appear in our database. The precision of the numerical predicates, with *query-portal*, is also low. The main reason is that many numerical attributes in the database are correlated. For instance, smaller screen size often leads to smaller weight. As the result, a keyword which correlates to smaller screen size may also correlate to smaller weight. In our evaluation, we only compare the mapping predicates with the manually labeled ones, without considering the correlation between predicates.

6. CONCLUSIONS

In this paper we studied the problem of leveraging existing keyword search interface to derive keyword to predicate mappings, which can then be used to construct SQL for robust keyword query answering. We validated our approach using experiments conducted on multiple search interface over the real data sets, and concluded that Keyword++ is a viable alternative to answering keyword queries.

7. REFERENCES

[1] S. Agrawal, K. Chakrabarti, S. Chaudhuri, V. Ganti, A. C. Konig, and D. Xin. Exploiting web search engines to search structured databases. In *WWW*, 2009.

[2] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.

[3] M. Bautin and S. Skiena. Concordance-based entity-oriented search. In *Web Intelligence*, pages 586–592, 2007.

[4] G. Bhalotia, A. Hulgeri, C. Naukhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, 2002.

[5] S. Chakrabarti, K. Puniyani, and S. Das. Optimizing scoring functions and indexes for proximity search in type-annotated corpora. In *WWW*, 2006.

[6] S. Chaudhuri, B.-C. Chen, V. Ganti, and R. Kaushik. Example-driven design of efficient record matching queries. In *VLDB*, 2007.

[7] S. Chaudhuri, V. Ganti, and D. Xin. Exploiting web search to generate synonyms for entities. In *WWW*, 2009.

[8] T. Cheng and K. C.-C. Chang. Entity search engine: Towards agile best-effort information integration over the web. In *CIDR*, 2007.

[9] T. Cheng, H. W. Lauw, and S. Paparizos. Fuzzy matching of web queries to structured data. In *ICDE*, 2010.

[10] E. Chu, A. Baid, X. Chai, A. Doan, and J. Naughton. Combining keyword search and forms for ad hoc querying of databases. In *SIGMOD*, 2009.

[11] G. B. Dantzig. Application of the simplexmethod to a transportation problem. In *Activity Analysis of Production and Allocation*, 1951.

[12] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, 2002.

[13] S. Kullback. The kullback-leibler distance. *The American Statistician*, 41, 1987.

[14] E. Levina and P. Bickel. The earthmovers distance is the mallows distance: Some insights from statistics. In *ICCV*, 2001.

[15] Z. Nie, J.-R. Wen, and W.-Y. Ma. Object-level vertical search. In *CIDR*, 2007.

[16] S. Paprizos, A. Ntoulas, J. Shafer, and R. Agrawal. Answering web queries using structured data sources. In *SIGMOD*, 2009.

[17] J. Pound, I. F. Ilyas, and G. E. Weddell. Expressive and flexible access to web-extracted data: a keyword-based structured query language. In *SIGMOD Conference*, 2010.

[18] Y. Rubner, C. Tomasi, and L. J. Guibas. A metric for distributions with applications to image databases. In *ICCV*, 1998.

[19] N. Sarkas, S. Paparizos, and P. Tsaparas. Structured annotations of web queries. In *SIGMOD Conference*, 2010.

[20] S. Tata and G. M. Lohman. Sqak: Doing more with keywords. In *SIGMOD*, 2008.

[21] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *SIGMOD*, 2009.

8. APPENDIX

We first review the related work, and then discuss the details of the algorithms, the possible extensions, and the supplemental experimental results.

8.1 Related Work

There are a number of recent work [10, 20] that also propose to answer keyword query by some form of SQL query translation. Specifically, authors in [20] develope the SQAK system to allow users to pose keyword queries to ask aggregate questions without constructing complex SQL. However, their techniques focus on aggregate SQL queries, and hence do not apply to our entity search scenarios. Authors in [10] explore the problem of directing a keyword query to appropriate form-based interfaces. The users can then fill out the form presented to them, which will in turn be translated to a structured query over the database.

Our work also share some flavor with the "query by output" paradigm recently proposed in [21], which produces queries given the output of an unknown query. A major difference is that in our problem the "output" is generated by a keyword query, which is inherently much more noisy than the "output" generated by a SQL query as considered in [21]. While methods proposed in [21] can also be applied to our problem, we show in Section 5 they are not most appropriate for our query translation purposes, and the statistical analysis based approaches we use outperform those proposed in [21].

The Helix system [16] proposes a rule based approach to translate keyword queries to SQL queries over a database. They automatically mine and manually enhance a set of patterns from a query log, and associate each pattern with a template SQL query. At query time, once a pattern is identified from a keyword query, the keyword query can be translated to a SQL using the associated query template. Recently, [17, 19] considered the problem of mapping a query to a table of structured data and attributes of the table, given a collection of structured tables. These works are largely complementary to the techniques we propose, in the sense that the keyword to predicate mappings discovered by our technique can be used to enhance the pattern recognition and translation components in [16], and enhance keyword to attribute matching in [17, 19].

Our technique differs from those for synonym discovery [9, 7] in that besides synonyms, it can also discover other types of semantic associations. For instance, "small" leads to an ascending sort on ScreenSize, "rugged" maps to ProductLine "Panasonic toughbook" *etc.* These types of mappings are generally not produced by synonym discovery.

Many commercial keyword search engines (*e.g.*, amazon) leverages query alteration mechanism, which takes into consideration of synonyms in processing user queries. Typically, the synonyms or the alternated queries are mined from user query sessions or user query click-url data. Unfortunately, such data is often not publicly available. Our approach, however, makes no use of such information and can be applied to any black-box search interface.

8.2 The Mapping Algorithm

8.2.1 The Algorithm

The algorithm takes six parameters: the keyword k, the set of all differential query pairs DQP_k, the baseline search interface SI, the entity relation E, thresholds for statistical significance of categorical predicates θ_{kl}, thresholds for statistical significance of numerical predicates θ_{emd}. The output of the algorithm is the predicate mapping $M_\sigma(k)$ and corresponding confidence score $M_s(k)$.

We first try to map the keyword k to a categorical or numeric predicate. Line 1-5 is a loop, where for each differential query pair in DQP_k, and for each attribute (or value), we aggregate the KL score and EMD score. Observe that for categorical predicates, we need to examine the attribute values. However, we only need to consider attribute values that appear in the foreground entities. Attribute values which do not appear in foreground entities will have 0 KL score. Therefore, the number of attribute values we need to examine is not too large.

The algorithm requires that the normalized scores are computed. We pick the categorical predicate and the numerical predicate with the highest score (line 6-9), and then compare it with the threshold. Recall that we have a set of thresholds, each of which corresponds to a range of the number of differential query pairs. If the score is higher than the corresponding threshold (determined by $|DQP_k|$), we compute a normalized score (line 10-11). If at least one of the numerical predicate and categorical predicate whose score is higher than the threshold, the algorithm outputs a numerical predicate (line 12-14), or a categorical predicate (line 15-16). Otherwise, if k is not a stop word, and appears in some textual attribute in the database, the algorithm maps k to a textual predicate. If none of the above is true, k is assigned to the null predicate.

Algorithm 1 Keyword to Predicate Mapping

Input: A keyword k,
A set of differential query pairs: DQP_k,
Baseline search interface: SI,
Entity relation: $E = E^c \cup E^n \cup E^t$
Threshold for categorical predicates: θ_{kl}
Threshold for numerical predicates: θ_{emd}

1: **for** (each $A_i^n \in E^n$)
2: Compute $AggScore_{emd}(\sigma(A_i^n, SO)|k)$;
3: **for** (each $A_i^c \in E^c$)
4: **for** (each $v_j \in D(A_i^c)$)
5: Computer $AggScore_{kl}(\sigma(A_i, v_j)|k)$;
6: Let $A_{emd} = argmax_{A_i^n}(|AggScore_{emd}(\sigma(A_i^n, SO)|k)|)$;
7: Let $S_{emd} = AggScore_{emd}(\sigma(A_{emd}, SO)|k)/|DQP_k|$;
8: Let $(A_{kl}, v_{kl}) = argmax_{(A_i^c, v_j)}(AggScore_{kl}(\sigma(A_i, v_j)|k))$;
9: Let $S_{kl} = AggScore_{kl}(\sigma(A_i, v_j)|k)/|DQP_k|$;
10: $S_{emd} = (|S_{emd}| > \theta_{emd})?\frac{S_{emd}}{\theta_{emd}} : 0$;
11: $S_{kl} = (S_{kl} > \theta_{kl})?\frac{S_{kl}}{\theta_{kl}} : 0$;
12: **if** $(|S_{emd}| > \max(0, S_{kl}))$
13: $SO = (S_{emd} > 0)?ASC : DESC$;
14: $M_\sigma(k) = \sigma_{(A_{emd}, SO)}$, $M_s(k) = |S_{emd}|$;
15: **else if** $(S_{kl} > \max(0, |S_{emd}|))$
16: $M_\sigma(k) = \sigma_{A_{kl}=v_{kl}}$, $M_s(k) = S_{kl}$;
17: **else if** $(\exists v \in A^t \in E^t,\ s.t.\ Contains(v, k) = true)$
AND (k *is not a stop word*)
18: $M_\sigma(k) = \sigma_{Contains(A^t, k)}$, $M_s(k) = 0$;
19: **else**
20: $M_\sigma(k) = \sigma_{TRUE}$, $M_s(k) = 0$;
21: **return**

8.2.2 Complexity Analysis

We give the complexity analysis of Algorithm 1. For each differential query pair $(Q_f, Q_b) \in DQP_k$, the baseline search interface returns S_f, S_b, respectively. We refer S_f as foreground data, and S_b as background data. To compute KL scores for all categorical predicates, we need to scan the values of entities in S_f and S_b, on each categorical attribute in E^c. We count the foreground and background occurrences of each value, and maintain it in a hash-table. The foreground and background occurrences are sufficient to compute $Score_{kl}$. Therefore, the complexity to compute scores for all categorical predicates is $Cost_{kl} = O(|S_f| + |S_b|)|E^c|)$. To compute the EMD score for each numerical predicate, we first sort the foreground (background) data, and then scan both sorted arrays once. Therefore, the complexity is $Cost_{emd} = O((|S_f|log(|S_f|) + |S_b|log(|S_b|))|E^n|)$. Putting these two together, the complexity of Algorithm 1 is the accumulated cost of $Cost_{kl} + Cost_{emd}$ over all differential query pairs in DQP_k.

We now discuss how effective the proposed method is, in terms of finding the right predicate for a keyword k. Assume the search interface is completely random with no bias for any value v that is not related to k, then v should share the same distribution in the foreground results and background results. Suppose the search interface process the foreground query and background query independently. The expectation of the statistical difference between foreground and background results on v is 0. On the other hand, we know that as long as the search interface is reasonably precise, then for value v which is related to keyword k, its distribution over the foreground results should be different from that over the background results. Therefore, the expectation of statistical difference on v should be nonzero. Although this only states the property of the expectation of the random variable, according to the law of large numbers, the more differential query pairs we get, the better chance the correct predicate will be recognized for a keyword k.

8.3 The Translation Algorithm

8.3.1 The Algorithm

The pseudo code of the translation algorithm is shown in Algorithm 2. The algorithm basically follows the recursive function described in Section 4. The key procedure is how we compute M_σ, the keyword to predicate mapping, in line 2 and line 5-7. As we discussed in Section 3.4, these mapping are pre-computed by Algorithm 1, and materialized in the mapping database. In this algorithm, we just lookup the mapping database.

8.3.2 Segmentation Schema

Note that the recursive function considers all possible keyword segmentation. Many segments are actually not semantically meaningful. One may rely on natural language processing techniques to parse the query, and mark the meaningful segments. In our implementation, we leverage the patterns in the query log. Intuitively, if a segment appears frequently in the query log, it suggests that tokens in the segment are correlated. However, frequent segments do not necessarily lead to "semantically meaningful" segments. On the other hand, if there are two queries which exactly differs on the segment, then the segment is often a semantic unit. Putting these two together, we measure the frequency of a segment by the number of differential query pairs in the query log, and keep a set of frequent segments as *valid segments*.

Algorithm 2 Query Translation

Input: A keyword query: $Q = [t_1, t_2, \ldots, t_q]$,
Mapping Database: M_σ and M_s,
Max number of tokens in keyword: n

1: Let $T_\sigma Q_0 = \phi$, $T_s(Q_0) = 0$;
2: Let $T_\sigma Q_1 = \{M_\sigma(t_1)\}$, $T_s(Q_1) = M_s(t_1)$;
3: **for** ($i = 2$ to q)
4: Let $n' = \min(n, i)$; //ensure $i \geq j$
5: $T_s(Q_i) = \max_{j=1}^{n'}(T_s(Q_{i-j}) + M_s(\{t_{i+1-j}, \ldots, t_i\}))$;
6: Let $j^* = argmax_{j=1}^{n'}(T_s(Q_{i-j}) + M_s(\{t_{i+1-j}, \ldots, t_i\}))$;
7: $T_\sigma(Q_i) = T_\sigma(Q_{i-j^*}) \cup \{M_\sigma(\{t_{i+1-j^*}, \ldots, t_i\})\}$;
8: **return** $T_\sigma(Q_q)$.

In query translation, a valid segment will not be split in keyword segmentation. As an example, in query [laptops for small business], "small business" is a valid segment. Therefore, we will always put "small" and "business" in the same segment. Thus, we will not consider the predicate mapped by "small" only. Since valid segments often carry the context information, enforcing valid segments in keyword segmentation helps to solve the keyword ambiguity issue. To integrate the valid segment constraint into Algorithm 2, one just need to check whether or not $\{t_{i+1-j}, \ldots, t_i\}$ (in line 5) splits a valid segment. If it does, we will skip this segment.

8.4 Extensions of Keyword++

Here we discuss some possible extensions of our proposed approach.

8.4.1 Extensions of Predicates

We write the numerical predicates as an "order by" clause in the SQL statement. This is a relatively "soft" predicate in that it only changes the order in which the results are presented. One may also transform it to a "hard" range predicate by only returning top (or bottom) $p\%$ results according to the sorting criteria.

In this paper, we assume each keyword maps to one predicate. This is true in most cases. In some scenario where a keyword could map to multiple predicates, we can relax our problem definition, and keep all (or some top) predicates whose confidence scores are above the threshold.

8.4.2 Dictionary Integration

Many keyword query systems use dictionaries as additional information. For instance, in the laptop domain, all brand names (*e.g.*, dell, hp, lenovo) may form a brand dictionary. The techniques developed in this paper is complimentary to the dictionary-based approach. On one hand, we can easily integrate the dictionary into our keyword to predicate mapping phase. If a keyword belongs to some reliable dictionary, one can derive a predicate directly. As an example, a keyword "dell" may be directly mapped to the predicate ($BrandName =$ "*dell*") since "dell" is in the brand name dictionary. On the other hand, our proposed method is able to correlate keywords to an attribute. Thus, it may be used to create and expand dictionaries.

The existence of dictionaries may also help to generate

more differential query pairs without looking at the set of historical queries. For instance, given a query Q=[small HP laptop] and the keyword k ="small", we may already derive Q_f=[small HP laptop] and Q_b=[HP laptop]. Observing that "HP" is a member of the brand name dictionary, we can replace "HP" by other members in the dictionary, and generate more differential query pairs for "small". As the result, we will have Q_f=[small Dell laptop] and Q_b=[Dell laptop], Q_f=[small Lenovo laptop] and Q_b=[Lenovo laptop], *etc.*

8.4.3 Integration to Production Search Engine

We discuss the integration in two scenarios. First, how to use a production search engine as a baseline interface in our framework. Secondly, how to integrate our techniques into a production search engine.

Consider the first integration scenario. Although the search interface of those production search engines are publicly available, the back-end databases of those search engines are not visible outside. They are often different from the entity database we have. Fortunately, our framework does not require the baseline interface returning the complete set of results. Therefore, we can match entities returned by the production search engine to those in our entity database. Our framework can take the partial results as input, and analyze the keyword and predicate mappings.

For the second integration scenario, many production search engines often consist of two components: entity retrieval and entity ranking. In the entity retrieval component, the search engine extracts meta data from the keyword query, and then uses both meta data and keyword to generate a filter set. In the entity ranking component, the search engine generates a ranked list of entities in the filter set. Our proposed method can be integrated into the production search engine in two ways. First, the categorical predicates identified from keywords can enrich the meta data. Therefore, it will improve the accuracy of the filter set. Secondly, the numerical predicates, as well as correlation scores computed between categorical values and the query keywords, can be considered as additional features for ranking entities in the filter set.

8.4.4 Multiple Search Interface

Since our method is a framework, which can be applied to multiple baseline search interface, the natural extension is to combine the high confident mappings derived from different search interfaces. Therefore, the overall accuracy of the translated query can be further improved. Intuitively, each search interface has different characteristics such that some mappings can be easily identified from certain search interface while others easily identifiable from a different search interface. For instance, a pure keyword matching search interface may identify "15" is related to ScreenSize. While a search interface which is built upon web search engine may recognize other correlations which are not based on keyword match (*e.g.*, "rugged" to ToughBook). Our approach of answering keyword query by first extracting keyword to predicate mappings allows easy composition of logics embedded in multiple search interface.

8.5 Supplemental Experimental Results

We first describe all comparison algorithms, and then present some additional experimental results.

8.5.1 Description of Comparison Algorithms

keyword-and: The first baseline search interface is the commonly used *keyword-and* approach. Given a keyword query, we first remove tokens which either do not appear in the database or are stop words. We can submit the remaining tokens in the query to a full text search engine, using the "AND" logic. Only entities containing all tokens are returned. We use standard stemming techniques in matching keywords. Despite its simple nature, the *keyword-and* based approach is still one of the main components in many production entity search systems.

query-portal: The second baseline search interface is the *query-portal* approach [1]. Different from the *keyword-and* approach, *query-portal* does not query the entity database directly. It establishes the relationships between web documents and the entities by identifying mentions of entities in the documents. Furthermore, it leverages the intelligence of existing production web search engine (*e.g.*, Bing). Specifically, for each keyword query, it submits the query to a web search engine and retrieves a set of top ranked documents. It then extract the entities in those documents using the materialized document-entity relationships. Intuitively, entities mentioned frequently among those top ranked documents are relevant to the keyword query. The *query-portal* approach exploits the knowledge of production search engines. Therefore, it is able to interpret a larger collection of keywords (*e.g.*, keywords that do not appear in database, but appear in web search queries).

keyword++: The *keyword++* framework is applied to both baseline search interface mentioned above. For each of the baseline search interface, our system is implemented based on the architecture shown in Figure 1. We scan the set of historical queries, and pre-compute keyword to predicate mapping for all single tokens and two-token phrases. We then keep all these mapping information in a mapping database. At online phase, we use Algorithm 2 to translate a keyword query to a SQL statement, which is further processed by a standard relational database engine, over the laptop database.

The baseline search interface is only used in the offline mapping phase. As we discussed in Section 3.4, it is possible that a query is submitted to a baseline search interface multiple times since it appears in multiple differential query pairs (for different keywords). In our implementation, we cache the results retrieved from the baseline search interface, and avoid probing same queries multiple times.

We compare *keyword++* with the baseline search interface. In addition to that, we also compare our method with two other approaches that could be possibly built on top of the baseline search interface.

bounding-box: The *bounding-box* based approach views the results returned by the existing search interface as points in the multi-dimensional data space, and then finds the minimum hyper-rectangular box that bounds all the sampled data points in it. Entities that lie in the minimum hyper-rectangular box are returned. The *bounding-box* approach considers the returning results as positive samples, and tries to find more entities that are close enough to the returning ones. Therefore, it only augments the results. As we have seen in the experimental results, it often improves the recall. But at the same time, it degrades the precision.

decision-tree: The *decision-tree* based approach constructs a decision tree based on the results returned by the baseline search interface. Specifically, one could label those returned

data as positive samples, and label all not-returned data as negative samples. Each node in the decision tree corresponds to a predicate, and one can translate a decision tree to a SQL statement. Since we do not know how deep the tree needs to be constructed, we build decision tree up to level q, where q is the number of tokens in the keyword query. We then compute q different SQL queries, where the i^{th} $(i = 1, \ldots, q)$ query corresponds to decision nodes from root to level i. We report the best results generated by these queries. This is the optimal results that may be achieved by the *decision-tree* based approach. Note that decision-tree approach is a variation of the method discussed in [21].

8.5.2 Additional Experimental Results

Here we present more experimental results with respect to the retrieval quality.

Effect of Mapped Predicates: In Section 5.4, we have shown that *keyword++* significantly improves the precision-recall scores of the results over that obtained by the baseline search interface. Here we drill down to different category of predicates, and examine their impact to the performance. As we discussed in Section 2.1.2, the translated SQL statement contains categorical, numerical and textual predicates. Since we do not consider numerical predicates at query level evaluation, we mainly compare the categorical and textual predicates, by varying the schema for textual predicates.

In Algorithm 1, if a keyword is not mapped to a categorical or numerical predicate (because its confidence score is below the threshold), it may be mapped to textual predicates (if it appears in some textual attribute). For all keywords appearing in textual predicates, we can apply the "and" logic such that all keywords have to be matched. This is the default configuration of *keyword++*. We can also apply the "or" logic such that it only requires one of the keywords to be matched (denoted as *textual-or*). Finally, we can simply drop all textual predicates (denoted by *textual-null*). We compare the Jaccard similarity for all three options in Figure 6, on both baseline search interface.

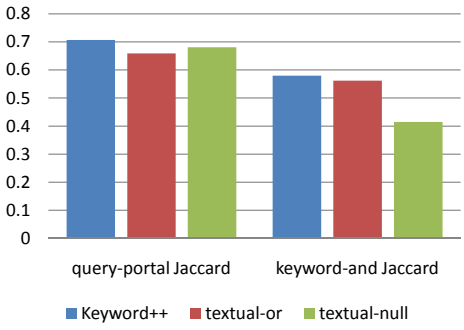

Figure 6: Jaccard w.r.t. Different Textual Predicate Schema

The results show that the textual predicates do not have dominating impact of the results, especially for the *query-portal* search interface, where all three options performs similarly. The performance boost achieved by *keyword++* is mainly contributed by the categorical predicates that were discovered by Algorithm 1. This shows the effectiveness of our proposed approach because it is the identified categorical predicates that distinguishe *keyword++* from common keyword match based systems.

Effect of Multiple Differential Query Pairs: We now examine the effect of score aggregation over multiple differential query pairs. In Section 3.2, we discussed two methods to generate differential query pairs, given a query Q and a keyword $k \in Q$. The first approach computes one differential query pair solely based on Q, and the second approach, which is used in our experiments, generates more differential query pairs from query log. We denote the first approach by *single-dqp*. Again, we conduct experiments on both *keyword-and* and *query-portal* search interface, and the results are shown in Figure 7. As expected, with only one differential query pair, the performance degrades.

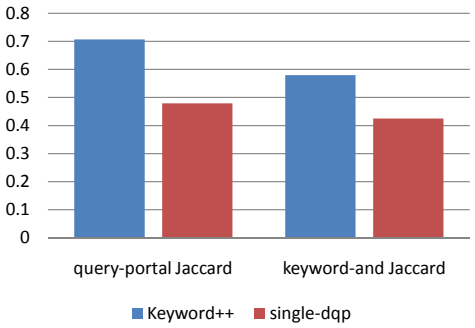

Figure 7: Effect on Number of Differential Query Pairs

Note that the *single-dqp* approach maybe be useful when a keyword did not appear in query log, and thus there is no differential query pair for the keyword from query log. In this case, *single-dqp* may be used if one wants to perform the keyword mapping analysis at online query translation.

Effect of Multiple Search Interface Combination: The last experiments in the query level evaluation is to examine the improvement by combining *query-portal* and *keyword-and* interface. As we discussed in Section 8.4.4, our framework enables us to integrate keyword to predicate mapping from multiple search interface.

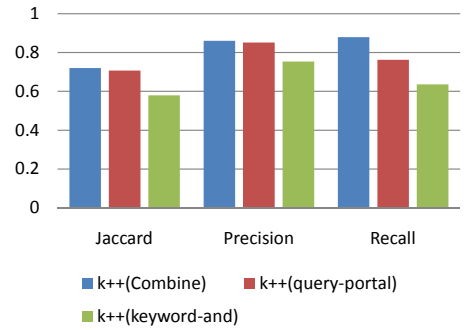

Figure 8: Combining Two Search Interface

We have already shown in Figure 5 that combining multiple baseline interface can boost the accuracy of predicate mapping. Here we complete the experiment by showing the results at query level evaluation. We put predicates from both search interface together, and run query translation algorithm. The results, which are shown in Figure 8, confirm that combining multiple search interface does improve jaccard similarity, precision and recall scores consistently.