ColumbuScout: Towards Building Local Search Engines over Large Databases

Cody Hansen Feifei Li
{sanans, lifeifei}@cs.utah.edu
School of Computing, University of Utah

In many database applications, search is still executed via form-based query interfaces, which are then translated into SQL statements to find matching records. Ranking is usually not implemented unless users have explicitly indicated how to rank the matching records, e.g., in the ascending order of year. Often, this approach is neither intuitive nor user-friendly (especially with many search fields in a query form). It also requires application developers to design schema-specific query forms and develop specific programs that understand these forms. In this work, we propose to demonstrate the *ColumbuScout* system that aims at quickly building and deploying a local search engine over one or more large databases. The ColumbuScout system adopts a search-engine-style approach for searches over local databases. It introduces its own indexing structures and storage designs, to improve its overall efficiency and scalability. We will demonstrate that it is simple for application developers to deploy ColumbuScout over any databases, and ColumbuScout is able to support search-engine-like types of search over large databases efficiently and effectively.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management—*Systems. Subject: Query processing*

1. INTRODUCTION

Indexing the web data and enabling the rank-based search over such data has been a fundamental problem over the last decade. A search engine usually provides a query interface with a single text box that enables the search over the underlying data based on one or more keywords. Furthermore, the search results are ranked before returning to the users. In contrast, in numerous database applications, search is still achieved via form-like query interfaces. While being useful in a lot of scenarios, we argue that this approach suffers two drawbacks. Firstly, it requires application developers to design schema-specific query forms and develop specific programs that understand these forms. A new, distinct query form is required for each new search type. Secondly, this approach can be user-unfriendly. Users may have to type values in or select values for a large number of search fields in a query form. More importantly, users are restricted to the types of search that the query forms were designed for; and in order to construct meaningful queries, users are forced to understand the schema of the underlying databases to some degree, either explicitly or implicitly.

In this work, we propose to demonstrate the *ColumbuScout* system, where its objective is to enable any user to build and deploy a search-engine-like system over their large databases quickly. Once deployed, the ColumbuScout instance should be able to execute search-engine-like queries efficiently and effectively. In particular, it should support key features like search-as-you-type, approximate search using multiple keywords, recommendations and rankings based on the query keywords (beyond simple database ranking operators, e.g., rank by year in ascending order). The challenge is to achieve these goals with efficiency and scalability, as demanded by the ever-increasing size of today's large databases. This demo will showcase our efforts in realizing this goal. Examples of running instances of ColumbuScout are available at our project website [4]. The Florida Craigslist instance indexes and searches 1.7 billion records from *Craigslist sites in Florida*; and the LinkedIn instance indexes and searches more than 12 millions records. In each instance, the underlying database was obtained by scraping open/public data from the Craigslist or the LinkedIn website respectively, and casting them into a corresponding relational schema where each record contains multiple fields (of different data types).

2. THE COLUMBUSCOUT SYSTEM

The ColumbuScout system starts with a database D and ends up with a web-based, search-engine-like query interface. The overall system architecture of ColumbuScout is shown in Figure 1. It consists of four main modules: the parser, the merger, the Flamingo builder, and the searcher. We will explain these modules in detail, while going through the construction of the engine, the search, and the updates in the ColumbuScout system. For ease of illustration, we assume that D consists of a single table.

Construction. Building a ColumbuScout engine has three major steps: parsing, merging, and indexing. They are the responsibility of the parser, the merger, and the Flamingo builder respectively.

The parsing step parses D into keyword-based inverted lists, one list per distinct keyword in D. For a distinct keyword w, its list $L(w)$ has the record ids for all records in D that contain w. That said, the *parser* maintains a hashmap H while scanning through records in D. An entry e in H has a (distinct) keyword $w(e)$ as its key, and a vector $\mathbf{v}(e)$ of record ids (rids) as its value, such that every record corresponding to a rid value from $\mathbf{v}(e)$ contains $w(e)$ (in at least one of its attributes). When the parser iterates through a record r in D, the parser views the entire record r as a single string s, regardless of the types of its different attributes. Next, s is tokenized into a set of keywords $\{w_1, \ldots, w_t\}$ (using the white-

SIGMOD'12, May 20–24, 2012, Scottsdale, Arizona, USA.

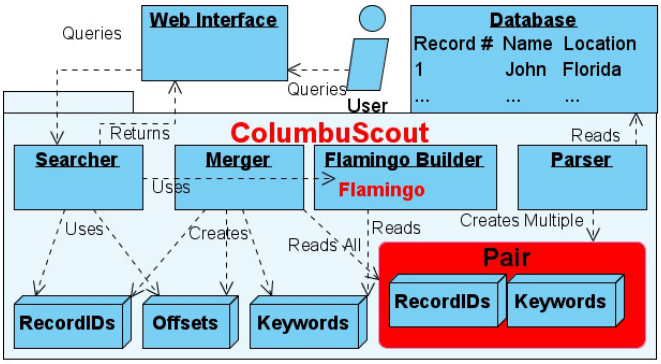

Figure 1: System architecture.

space as the delimiter, except for certain special attributes such as phone numbers), one at a time. Suppose the record id (rid) of r is j, we insert the pairs (w_i, j) for $i = 1, \ldots, t$ into H.

The main challenge is what to do when H becomes too large (to fit in the memory). To handle this situation, we have designed a simple yet effective storage engine that caters for massive data. Figure 2 illustrates our idea. Basically, in a live ColumbuScout instance, there are three files: the *ascii* keyword file, the *binary* rids file, and the *binary* offset file. The keyword file stores all distinct keywords $\{w_1, \ldots, w_m\}$ *in sorted order* from H, line delimited. The rids file stores the lists of record ids for keywords in the keyword file, in the same order as keywords appear in the keyword file. At the head of each list L_i for keyword k_i, it also stores an integer n_i which states how many rids L_i contains. Since each element in any list L_i is a rid which can be represented as an unsigned integer, and any n_i is also an unsigned integer. The entire rids file is just a binary file of unsigned integers. Finally, we also need the *binary* offset file (for facilitating the *searcher*), which stores m offset values. The ith value offset_i stores the offset of the ith list in the rids file, i.e., it is the starting address of n_i in the rids file.

When H is first initialized, we denote this as the *iteration 1*. Whenever the size of H exceeds the available memory after inserting a (keyword, rid) pair, we dump H to disk and empty it. This marks the end of the current iteration. We then resume processing the next (keyword, rid) pair which marks the beginning of a new iteration. That said, at the end of the ith iteration, we dump H into two files F_k^i and F_{rid}^i, where F_k^i is a (sorted) keyword file storing all keywords from H in the ith iteration, and F_{rid}^i is a rids file storing the corresponding rids lists. At the end of the parsing step, we end up with a series of these pairs of files, one pair per iteration.

Suppose the parsing phase produces T iterations. The next phase is the merging phase, where *the merger* merges T pairs of files into a single pair of files, i.e., it creates a single keyword file F_k and rids file F_{rid} from $\{(F_k^1, F_{\text{rid}}^1), \ldots, (F_k^T, F_{\text{rid}}^T)\}$. Since each pair of files is sorted to begin with, this merging step is fairly easy to execute. Consider a special case when $T = 2$, we maintain two cursors I_1 and I_2, one cursor per keyword file (initialized at the first record in each file respectively). We always *output the smaller keyword* to the output keyword file F_k among the two keywords currently pointed by I_1 and I_2. We also maintain two cursors O_1 and O_2, initialized at the first byte of F_{rid}^1 and F_{rid}^2.

Without loss of generality, suppose the first keyword w_1 in F_k^1 pointed by I_1 is the first one being pushed to F_k. Clearly, the starting address of n_1 and L_1 (the list of rids for records that contain w_1) in F_{rid}^1 is given by O_1. We can read $(n_1 + 1) \cdot b$ bytes sequentially from F_{rid}^1 to retrieve the binary content of n_1 and L_1, where b is the size of an unsigned integer. These $(n_1 + 1) \cdot b$ bytes will be pushed to the output file F_{rid}. After that, we move forward I_1 to the second keyword in F_k^1, and O_1 by $(n_1 + 1) \cdot b$ bytes. We

keywords	rids file	offsets
w_1	n_1 $\text{rid}_{1,1}, \ldots, \text{rid}_{1,n_1}$	offset_1
w_2	n_2 $\text{rid}_{2,1}, \ldots, \text{rid}_{2,n_2}$	offset_2
$\cdots$	$\cdots$	$\cdots$
w_m	n_m $\text{rid}_{m,1}, \ldots, \text{rid}_{m,n_m}$	offset_m

Figure 2: The storage engine.

also write the starting address of n_1 and L_1 in F_{rid} as an unsigned integer to a file F_{offset}. A special case is when keywords w_i and w_j pointed by I_1 and I_2 are the same word w. When this happens, we simply merge the corresponding lists L_i and L_j (pointed by O_1 and O_2 in F_{rid}^1 and F_{rid}^2 respectively) into one list $L(w)$, write w to F_w and $(|L(w)|, L(w))$ to F_{rid}, and move I_1, I_2, O_1, and O_2 forward accordingly. This process is then recursed, till both I_1 and I_2 point to the end of F_k^1 and F_k^2 respectively.

One can generalize the above procedure to merging T pairs of keywords and rids files at the same time, by maintaining T cursors instead. This produces the keywords, the rids, and the offsets files F_k, F_{rid} and F_{offset}, just as described in Figure 2. They correspond to the dumped content of the hashmap H as if it was built over the entire D and never exceeded the available memory. Obviously, we can also do this merging in parallel (two pairs in a thread) following a divide-and-conquer scheme. More interestingly, our parsing and merging phases are ideal for parallelization over massive data in MapReduce, which is our ongoing effort. We omit the details.

The third and final phase is to index the keywords from F_k to support the approximate string search and the search-as-you-type feature. We assign *unique ids* to keywords in F_k based on the ordering in F_k, i.e., the ith keyword in F_k is assigned the id i. Flamingo does an excellent job in indexing these unique ids so that given a query keyword q, it can quickly retrieve *all keyword ids* that correspond to keywords that are similar to q [1]. It also supports a variety of approximate string match metrics, such as the edit distance, the jaccard similarity, the cosine similarity and the dice similarity. We incorporated the Flamingo library in the ColumbuScout system (the Flamingo Builder in Figure 1), which builds the index over F_k and we denote this index as the "Flamingo" index.

Search. An overview of the search process in ColumbuScout is given in Figure 3. It starts with parsing a user query into a set of query keywords $\{q_1, \ldots, q_u\}$. Next, for each q_i we use the Flamingo index to get a vector $\mathbf{w}_i$ of *keyword ids* that correspond to keywords in F_k that are similar to q_i (based on any of the string similarity metrics, by default, we used the edit distance). Next, we convert $\mathbf{w}_i$ to a vector $\mathbf{v}_i$ of rids, which correspond to those records that contain at least one keyword identified by keyword ids in $\mathbf{w}_i$. To do so efficiently, for every keyword id $j \in \mathbf{w}_i$, we first find its offset value offset_j in F_{offset}. Note that this can be done in constant time and IO, by using the seek functionality available in a binary file. Specifically, the starting address of offset_j in F_{offset} must be $(j-1)b$, where b is the size of an unsigned integer. Given offset_j, we use the seek functionality again but on the binary file F_{rid}, to retrieve the value n_j in constant time and IO. After that, we simply load L_j sequentially by reading $n_j b$ bytes from F_{rid} coming after n_j. We insert every such L_j into $\mathbf{v}_i$ for any $j \in \mathbf{w}_i$.

Once we have these vectors of rids, $\mathbf{v}_i$ for q_i, we need to get the rids that appear at least τ (a system threshold) times. To find those rids, ColumbuScout designs an algorithm that shares similar principles to the MergeSkip algorithm in [9]. In particular, this algorithm (*rids Merger* in Figure 3) uses a heap, binary search and additional popping and pruning rules to achieve efficiency and scalability. During this process ColumbuScout also makes sure that the count on the number of matches do come from different keywords, e.g., a search for "blue cat" does not return a record containing both "fat" and "cat", but no words that match "blue" (if $\tau = 2$).

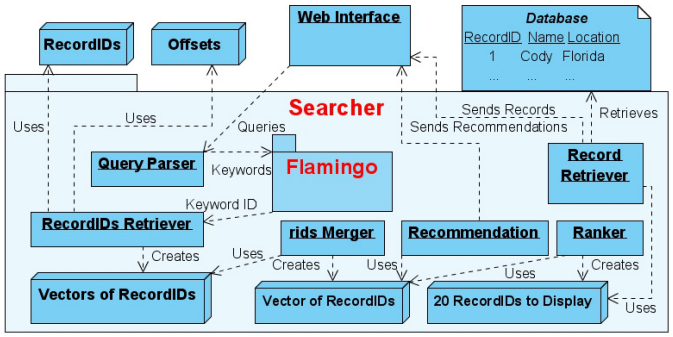

Figure 3: Overview of search in ColumbuScout.

Figure 4: Recommendation and ranking.

For brevity, we omit the details of this algorithm. For the current version we have the threshold τ equal to the number of keywords in user queries, but this is easily adjusted, even dynamically.

The rids merger stores the desired rids in a vector $\mathbf{v}$. Every rid in $\mathbf{v}$ corresponds to a record that *matches* the query (with at least τ similar keywords between the two). Instead of simply displaying these records back to the user, ColumbuScout performs recommendations and rankings based the results of the merger. Suppose for a rid j in $\mathbf{v}$, its record r_j matches $\alpha \geq \tau$ query keywords as determined by the merger. ColumbuScout can rank the results by a variety of methods that we have explored. Currently, by default it ranks records that match every query keyword exactly above all others. The rest are ranked by the combination of their α values and the *rarity of the keyword combination* they matched by.

Consider the example in Figure 4, if the system is queried by "cody orlando", records that contain both "cody" and "orlando" will be ranked first. Then let there be one record that contains "cozy" and "orlando", and three records that contain "body" and "orlando". The record containing "cozy" and "orlando" would be ranked above the other three records because the keyword combination "cozy" and "orlando" is more rare than the combination of "body" and "orlando". This ranking shows the user the rare matches above others and will therefore effectively prune out obvious and common keywords that are not effective to search by. When there is a tie in the rarity of two records (the rarity for the combination of matched keywords from them), we use their α values to break the tie. To estimate the rarity of a record r, we adopted two strategy. In the first strategy, we estimate the rarity of the matched keywords of r based on only records from $\mathbf{v}$. This can be done at the query time by dynamically building a matched-keywords histogram while we process records in $\mathbf{v}$. In the second strategy, we estimate the rarity of the matched keywords of r based on all records in D, which builds (and maintains) a matched-keywords histogram incrementally over time. This can be done via the help of the inverted lists of distinct keywords and the search history. Due to the space constraint, we omit the details.

There are other types of ranking strategies that we have tested, such as ranked by the rarest keyword from each matched record (instead of using the combination of matched keywords). The ranking is easily adjustable and could be also offered as a user choice in our system. By default, we have used the above ranking method with strategy one to decide the rarity of the combination of matched keywords. Based on this ranking framework, we also designed a flexible and effective *recommendation module*. When a search is made (while users are typing it) the potentially matched keyword combinations are showed, along with how many records matched that keyword combination. They are adjusted in real time while user is tying each single character in the search box. And just like most popular search engines, users can select a search to see those results. This allows users to make a search, see the results and see what other keywords are in the data, and easily find the records they wanted to see (i.e., refine their search). ColumbuScout also has a prefix search that is done on the final word of the query in addition to the fuzzy search. These features together guide the users to the results they want to see quickly and efficiently, even when they know nothing about the schema of the underlying data.

Updates. The design of the ColumbuScout system permits efficient updates, especially for batched insertions and deletions. The basic idea is to create a new pair of keywords and rids files for affected records, and use the *merger* in Figure 1 to merge them with the existing keywords and rids files. Supporting batched deletions is trickier, but can also be done efficiently after resolving some technicalities. We omit the details due to the space constraint.

Performance. We deployed two ColumbuScout instances over the Florida Craigslist data and the LinkedIn data. The Craigslist data has 1.7 billion records and each record has 8 keywords on average, leading to a total of 300GB of data; the LinkedIn data has 12 million records and each record has 6 to 20 keywords, leading to a total of more than 10GB of data. There are 367,777 unique words in our Craigslist data, and 1,930,818 unique words in our LinkedIn data. We have tested the performance of the ColumbuScout system on a Linux machine running ubuntu12.9 and mysql Server (5.1.41-3). This machine has 12GB memory, 36GB swap memory, a 2TB hard-drive, and a single Intel(R) Xeon(R) CPU X3470 @ 2.93GHz.

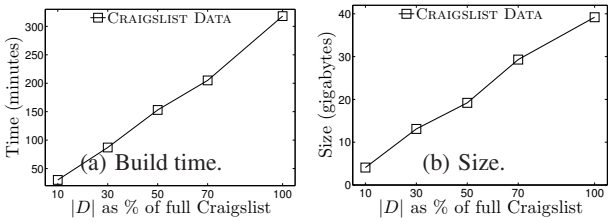

Figure 5: Build time and size of ColumbuScout on Craigslist.

First, we examined the scalability of the ColumbuScout system, where we built ColumbuScout instances over databases of different size. In particular, we have varied the size (number of records) of the Craigslist data from 10% to 100%, and Figure 5 shows the results on the build time and the size of the resulting ColumbuScout instance. Clearly, both the construction cost and the size of the ColumbuScout instances are linear to the database D (and much smaller than $|D|$, for example, it is only 40GB when D is at 300GB). The ColumbuScout system can be quickly deployed in practice. For example, even when D has 1.7 billion records, it only takes about 5 hours to build and deploy ColumbuScout over D; when D has 170 million records, it only takes 30 minutes. Also, as we have discussed, ColumbuScout can be easily built in parallel using MapReduce, which further reduces its construction cost.

Next, we examined the query efficiency. Figure 6 summarizes the results using the *full* Craigslist and LinkedIn data, where u is the number of query keywords in one search, and k is for recommendation and ranking display (top-k records). In summary, it can

search two datasets in only tens to hundreds of milliseconds, one being 1.7 billion records and the other being 12 million (larger) records, while making recommendations, rankings, and supporting search-as-you-type with approximate matching over multiple query keywords. In essence, ColumbuScout gives the power of a large corporation's search engine to anyone who deploys it over their local data.

	Full Craigslist		Full LinkedIn	
u	1	3	1	3
$k = 200$	0.0286	0.0669	0.0157	0.0408
$k = \|D\|$	0.0353	0.0889	0.0506	0.1359

Figure 6: Query efficiency in ColumbuScout: seconds.

Ongoing work. There are other interesting issues we are actively working on to improve the efficiency and the effectiveness of the ColumbuScout system. In terms of efficiency, we have recently implemented a parallel version of the construction module using MapReduce. It achieves almost a linear speedup to the number of machines in a MapReduce cluster. We are also investigating how to use a cluster of commodity machines in the query process. In terms of effectiveness, a critical problem we are actively working on is to enhance the ranking and recommendation modules. In particular, we want to leverage on associations and linkage/lineage between keywords inside the database to make ranking and recommendation decisions. We are also investigating how to use certain ontology information (either built from the underlying database or made available through another data source such as Wikipedia) to achieve context-aware ranking and recommendations.

3. DEMO DESCRIPTION

Construction and deployment. Our objective here is to illustrate that it is almost effortless and very efficient to deploy ColumbuScout over an existing database. We will deploy ColumbuScout over a relational database D with tens to hundreds of thousands of records, each with multiple attributes with different data types.

Query executions. In this part, our objective is to illustrate the flexibility, the scalability, and the effectiveness of the ColumbuScout system. At the time of the submission of this paper, we have already deployed two ColumbuScout instances over two separate, large databases: the Craigslist with 1.7 billion records; and the LinkedIn data with 12 million (larger) records. In both cases, we obtained the data through scraping (only) public and open pages from the Craigslist (in Florida) and the LinkedIn websites. Note that we only got a subset of available pages from these websites. But they are already massive. A query example from the LinkedIn data is shown in Figure 7. Both instances are deployed and running at the same server [4], and we even offer a query interface to search both data simultaneously using the same server!

We will also use a traditional, form-like query interface over the aforementioned database D, so that users can compare the search experiences between the ColumbuScout instance and the traditional form-like query interface over the same database D.

Updates. Lastly, we will demonstrate that our system supports dynamic updates efficiently. This is achieved via a set of records to be inserted into D, which triggers the updates to the ColumbuScout instance running on D we have initially built. We will then construct several queries to search for the newly inserted records.

Setup. Our demo needs a standard PC that connects to the Internet. We will pre-configure the PC to load the ColumbuScout system and the local database D, as well as a local web-server to host the constructed ColumbuScout instance over D. The other two running instances (over the Craigslist and the LinkedIn data) will be available via the Internet connection at our home server.

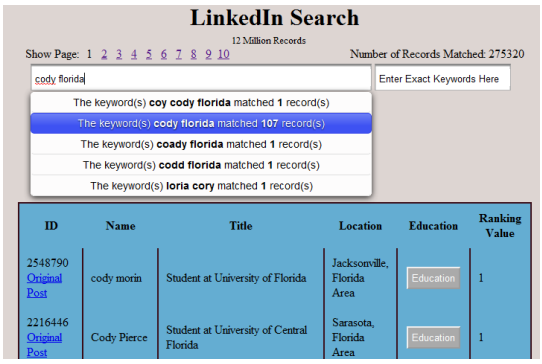

Figure 7: A *ColumbuScout* instance.

4. RELATED WORK

The ColumbuScout system builds upon previous work on supporting approximate string search [1, 9] (see an excellent tutorial in [7] and references therein) and search-as-you-type [11], by extending these techniques to incorporate the features such as search-engine-style recommendations and rankings. In order to integrate these various features seamlessly, efficiently, and effectively, we have introduced new designs and auxiliary indexing structures and storage organizations. We also plan to extend the ColumbuScout system to incorporate some of the ideas from keyword search in relational databases, where the focus is to find out how records (potentially from different tables) are connected through the query keywords [8], interested readers may refer to two excellent tutorials [5, 6] and references therein. These ideas have also been illustrated in some of the previous system demonstrations [2, 3, 10].

5. ACKNOWLEDGMENT

This work was partially supported by a grant for exploring non-conventional search from the Florida Department of Revenue and a REU grant from the National Science Foundation. The authors would like to thank Sudhir Aggarwal, Sharon Keri, Piyush Kumar, Manav Kundra, Kyle Stark, and Shiva Houshmand Yazdi for providing valuable feedback and the datasets used in our test.

6. REFERENCES

[1] The flamingo project. `http://flamingo.ics.uci.edu/`.
[2] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: enabling keyword search over relational databases. In *SIGMOD*, 2002.
[3] S. Bergamaschi, F. Guerra, S. Rota, and Y. Velegrakis. KEYRY: A keyword-based search engine over relational databases based on a hidden markov model. In *ER Workshops*, 2011.
[4] C. Hansen and F. Li. The ColumbuScout Project. `http://datagroup.cs.utah.edu/columbuscout.php`.
[5] Y. Chen, W. Wang, and Z. Liu. Keyword-based search and exploration on databases. In *ICDE*, 2011.
[6] Y. Chen, W. Wang, Z. Liu, and X. Lin. Keyword search on structured and semi-structured data. In *SIGMOD*, 2009.
[7] M. Hadjieleftheriou and C. Li. Efficient approximate search on string collections. *PVLDB*, 2(2):1660–1661, 2009.
[8] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, 2003.
[9] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.
[10] Y. Luo, W. Wang, and X. Lin. SPARK: A keyword search engine on relational databases. In *ICDE*, 2008.
[11] H. Wu, G. Li, C. Li, and L. Zhou. Seaform: Search-as-you-type in forms. *PVLDB*, 3(2):1565–1568, 2010.