International Journal of Engineering Research and General Science Volume 3, Issue 1, January-February, 2015
ISSN 2091-2730

Improving Efficiency in Keyword Search Using Exact and Fuzzy Methods

Nikhil Raj, Ashwani Kumar, Vibhans Kumar, Ravi Verma, Sonali Bhandurge

Under Graduate, Computer Engineering, MIT Academy of Engineering, Pune

nikhil.280247@outlook.com, +919175472189

Abstract— In today's emerging world, there is a great need to balance and lessen the gap between non-technical and technical ones. Most applications are being developed in order to make people life easier. Search is one of the most basic and important tool utilized by humans in everyday life. Search as-you-type feature allows you to get answers on the fly as a user fires a query character by character. So, we are going to simulate this type of search in our paper with the help of *exact* and *fuzzy* search methods. We will study how search as-you-type works on data lying in the backend database. But there are many challenges in the implementation of these methods, which include security issues, application compatibility in all platforms and the most important is the response time of application. We also study how to use indexes in tables that will increase overall performance of searching. We also make declarative solutions and techniques using exact and fuzzy search. Lastly we have tested our application on large and real time data with millions of records that shows far better good results.

Keywords— Exact Search, Fuzzy Search, Like and UDF methods, Gram based method, Incremental Computation method, Neighborhood Generation method, Inverted table method

INTRODUCTION

There are many information systems nowadays which provide autocomplete search by providing instant results as soon as possible. Many search engines and websites support autocomplete search, which provide multiple answers to the queries provided by the user. This feature is also known as answers "on the fly". For example, shopping sites helps user search different types of commodities with a single keyword search of user. If a user types in "*Refrigerators*", then the server may show multiple results with a title matching this keyword as a prefix. This type of search is popularly known as *type ahead search*.

Many databases, say, Oracle and SQL server support *type ahead* search. But there are many challenges which are to be considered and moreover all databases do not support this feature. Generally autocomplete search methods use three approaches which can be summarized as follows:

1) Constructing indexes on databases using separate application layer can be used to maintain indexes. However this feature has a benefit that it can be used to achieve performance, on the contrary it has a major problem of duplication of data and indexes that may result in additional hardware costs
2) Database extenders, say, Informix DataBlades, MS SQL Server CLR integration allow developers to provide some additional features, but the point of concern is non-availability of extender feature in databases such as SQL.
3) To use standard SQL techniques which are also portable to other databases. Gravano et al. [1] and Jestes et al.[2] made similar observations.

SEARCH CATEGORIES

In particular, there are two types of search which is mostly observed, namely *multikeyword search and fuzzy search.* In multikeyword search techniques, a user types in query containing multiple keywords, and find tuples that are similar to these keywords irrespective of the location of keywords. For example, if a user types in "Operating Machines" to find out a book by "R.K.Mishran" with a title including "Operating" and "Machines" irrespective of the locations. In fuzzy search, minor discrepancies may be present between entered query and actual results. For example, if a user types in "Mishrn" despite the word "Mishran", then also this type of search techniques can prove useful. Depending on these search techniques, multiple methods have been discussed later in the paper.

PREPARATORY MEASURE

We will first plan the problem of search-as-you-type in database management system and then we will discuss different ways to support search-as-you-type.

PROBLEM FORMULATION

Let us take *T* as a relational table with attributes A_1, A_2, A_3,……, A_l. Let $R = \{r_1, r_2, r_3, ….., r_n\}$ be the collection of records in *T*, and $r_i[A_j]$ denote the content of record r_i in attribute A_j. Let *W* be the set of tokenized keywords in *R*.

SEARCH-AS-YOU-TYPE FOR SINGLE-KEYWORD QUERIES

Exact search: When a user types in a single partial keyword *w*, search-as-you-type immediately finds records that contain keyword with a prefix as *w*. This type of search is known as prefix search or exact search. Consider the table *T* with given set of data. If any user types in a query "sig", it returns records r_3, r_6, r_9. In particular, r_3 contain a keyword "sigmod" with a prefix "sig".

TABLE 1
Table dblp: A Sample Publication Table (about "Privacy")

ID	Title	Authors	Booktitle	Year
r_1	K-Automorphism: A General Framework for Privacy Preserving Network Publication	Lei Zou, Lei Chen, M. Tamer Özsu	PVLDB	2009
r_2	Privacy-Preserving Singular Value Decomposition	Shuguo Han, Wee Keong Ng, Philip S. Yu	ICDE	2009
r_3	Privacy Preservation of Aggregates in Hidden Databases: Why and How?	Arjun Dasgupta, Nan Zhang, Gautam Das, Surajit Chaudhuri	SIGMOD	2009
r_4	Privacy-preserving Indexing of Documents on the Network	Mayank Bawa, Roberto J. Bayardo, Rakesh Agrawal, Jaideep Vaidya	VLDBJ	2009
r_5	On Anti-Corruption Privacy Preserving Publication	Yufei Tao, Xiaokui Xiao, Jiexing Li, Donghui Zhang	ICDE	2008
r_6	Preservation of Proximity Privacy in Publishing Numerical Sensitive Data	Jiexing Li, Yufei Tao, Xiaokui Xiao	SIGMOD	2008
r_7	Hiding in the Crowd: Privacy Preservation on Evolving Streams through Correlation Tracking	Feifei Li, Jimeng Sun, Spiros Papadimitriou, George A. Mihaila, Ioana Stanoi	ICDE	2007
r_8	The Boundary Between Privacy and Utility in Data Publishing	Vibhor Rastogi, Sungho Hong, Dan Suciu	VLDB	2007
r_9	Privacy Protection in Personalized Search	Xuehua Shen, Bin Tan, ChengXiang Zhai	SIGIR	2007
r_{10}	Privacy in Database Publishing	Alin Deutsch, Yannis Papakonstantinou	ICDT	2005

Fuzzy search: When a user types in a single partial keyword *w* which is basically prefix character by character, fuzzy search immediately finds the record with keyword similar to the query keyword. For example, if a user types in a query "correl", record r_7 is a relevant answer since it contain a keyword "correlation" with a prefix "correl" which is similar to the query keyword "corel". In fuzzy search we use edit distance to measure similarity between strings. The edit distance between two strings, say, *s1* & *s2*, denoted by *ed(s1,s2)*, is the minimum number of single-character edit operation needed to transform *s1* to *s2*. For example, ed(corelation, correlation) =1 and ed(coralation, correlation) =2. A prefix *p* of a keyword is similar to the partial keyword *w* if ed(p, w) ≤ edit distance threshold.

SEARCH-AS-YOU-TYPE FOR MULTI KEYWORD QUERIES

Exact search: Given a multi keyword query *Q* with *k* keywords w_1, w_2, w_3,…., w_k, as a user completed the last keyword w_k as a partial keyword and other keyword as a complete keywords. If a user type in a query "privacysig", search-as-you-type returns record r_3, r_6 and r_9.

Fuzzy search: Fuzzy search finds the record with keyword similar to the complete keyword and a keyword similar to the partial keyword w_m. Suppose edit distance is equal to one. Assuming that a user types in a query "privicycorel", fuzzy type ahead search

return record r_7 since it contains a keyword "privacy" similar to the complete keyword "privicy" and a keyword "correlation" with a prefix "correl" which is similar to the partial keyword "corel".

EXACT SEARCH FOR SINGLE KEYWORD

In this section we proposed two types of methods to use SQL to support Search-as-You-Type for single keyword Queries.

No-Index Method

A simple and straight forward way to support search-as-You-Type is to issue an SQL query that scans record and verifies whether the record is an answer to the query which can be implemented by two methods:

1) *Calling User-Defined Functions (UDFs):* We can add functions into database to verify whether a record contains the query keyword.
2) *Using the LIKE predicates:* Databases provide LIKE predicates to allow user to perform string matching. But this method may introduce false positive which can be removed by introducing UDF's.

Index-Based Method

In index-Based method we proposed building auxiliary tables as index structure to facilitate prefix search.

Inverted-index table: In table, we assign unique ids to the keyword in the table *T*, following their alphabetical order. We create an inverted-index table I_T with record in form <*kid, rid*>, where *kid* is the id of record that contain the keyword.

TABLE 2
The Inverted-Index Table and Prefix Table

(a) Keywords

kid	keyword
k_1	icde
k_2	icdt
k_3	preserving
k_4	privacy
k_5	publishing
k_6	pvldb
k_7	sigir
k_8	sigmod
k_9	vldb
k_{10}	vldbj
...	...

(b) Inverted-index Table

kid	*rid*
k_2	r_{10}
k_5	r_6
k_5	r_8
k_5	r_{10}
k_6	r_1
k_7	r_9
k_8	r_3
k_8	r_6
k_9	r_8
k_{10}	r_4
...	...

(c) Prefix Table

prefix	*lkid*	*ukid*
ic	k_1	k_2
p	k_3	k_6
pr	k_3	k_4
pri	k_4	k_4
pu	k_5	k_5
pv	k_6	k_6
pvl	k_6	k_6
sig	k_7	k_8
v	k_9	k_{10}
vl	k_9	k_{10}
...	...	...

Prefix table: For all prefixes of keywords in the table, we build a prefix table P_T with record in the form <*p, lkid, ukid*>, where *p* is a prefix of keyword, *lkid* is the smallest id of those keyword in table *T* having *p* as prefix and *ukid* is the largest id of those keyword in table *T* having *p* as prefix. So, given a prefix keyword *w*, we can use the prefix table to find the range of keyword with the prefix. For example Table 2 illustrate the inverted-index table and the prefix table for the record in Table 1.

Suppose given a partial keyword *w*, we first get its keyword range *[lkid,ukid]* using prefix table P_T, and then find records that have keywords in the range through the inverted-index table I_T. For example, if a user enters the keyword "sig", then the SQL query first finds out keyword range [k_7,k_8] based on P_T. Next, it finds the records containing a keyword with ID in [k_7, k_8] using I_T. We can use SQL to answer the prefix search query w:

SELECT T* FROM P_T, I_T, T WHERE P_T.prefix = "w" AND P_T.ukid ≥ I_T.kid AND P_T.lkid ≤ I_T.kid AND I_T.rid = T.rid.

FUZZY SEARCH FOR SINGLE KEYWORD

No-Index Methods

In fuzzy search LIKE predicate is not supported, so we take UDF to implement no-index methods. We use PED(w,s) that takes keyword w and string s as two parameters and returns minimal edit distance between w and the prefixes of the keywords in s. For example, in Table 1,

PED(‘‘pvb’’, r_{10}[title])=PED(‘‘pvb’’; ‘‘privacy in database publishing ’’)=1.

Here the edit distance is 1 to the query where r_{10} bears a prefix “pub”. PED(w,s) returns true when keyword in string s has prefixes with edit distance within τ(edit distance threshold).

INDEX-BASED METHODS

In this method I_T and P_T are utilized to support fuzzy search. In the first stage, from prefix table P_T, we calculate its similar prefixes and range of the keywords is obtained from these similar prefixes. Then with the help of I_T, we calculate the answer depending upon these ranges.

USING UDF

UDF can be used to find similar prefixes from the prefix table P_T when a keyword w is given. The underlying SQL query helps to scan each prefix in P_T and a call is made to the UDF to notice whether prefix is similar to w:

SELECT $T.*$ FROM P_T, I_T, T WHERE PEDTH(ω, P_T , $prefix$, τ) AND $P_T.ukid \geq I_T.kid$ AND $P_T.lkid \leq I_T.kid$ AND $I_T.rid = T.rid$.

Performance can be improved by utilization of length filtering which can be done by adding the following clause to the where clause:

“LENGTH($P_T.prefix$) ≤ LENGTH(ω) +τ AND

LENGTH($P_T.prefix$) ≥LENGTH(ω) – τ”.

GRAM-BASED METHOD

Approximate string search is supported by many q-gram-based methods. String s is been given as input then its q-grams are its substrings with length q. We assume that $G^q(s)$ represents set of its q-grams and the size of $G^q(s)$ is been represented by $|G^q(s)|$. For instance, for “pvldb” and “vldb”, have G^2(pvldb)={ pv,vl,ld,db} and G^2(vldb)={vl,ld,db}. From the above example we say that strings s_1 and s_2 pose edit distance within threshold τ if

$|G^q(s_1) \cap G^q(s_2)| \geq \max(|s1|, |s2|) + 1 - q - \tau * q,$

This implementation is named as count filtering. But creating I_T and P_T is not enough, we even need to create a q-gram table G_T with records in the manner (p,$q$gram) when there is a necessity to find similar prefixes of the query keyword w, where p is referred as prefix in the prefix table and $q$gram referred as q-gram of p. When a partial keyword w is given as input, the initial step is to search the prefixes in $G_T$with no smaller than $|\omega| + 1 - q - \tau * q$grams in $G^q(\omega)$.

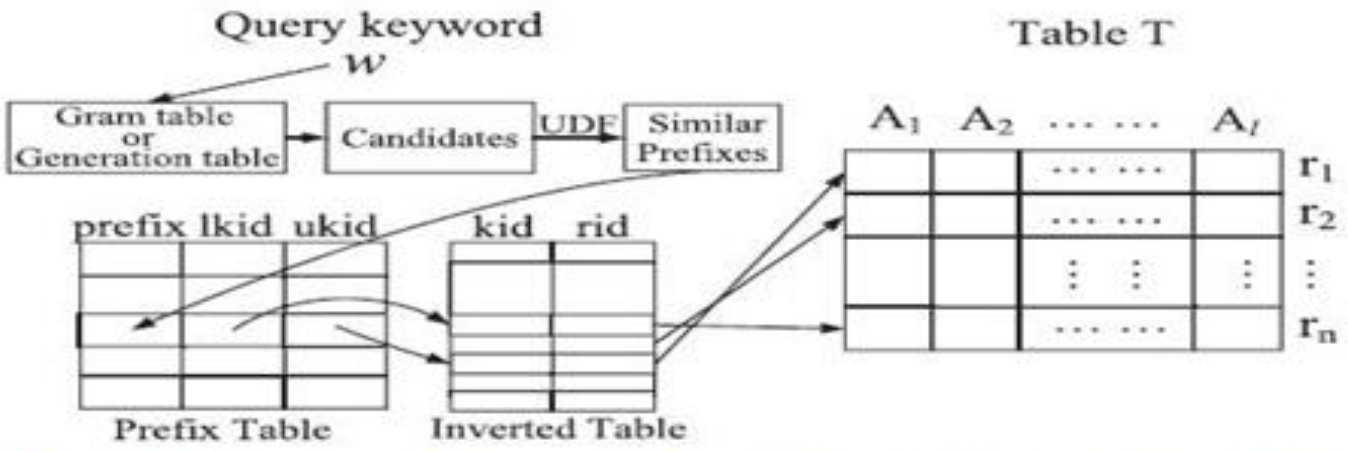


Fig. 1 . Using the q-gram table and the neighborhood generation table to support fuzzy search.

To obtain the candidates of w's similar prefixes, the following SQL with "GROUP BY" command is mentioned:

SELECT $P_T.prefix$ FROM G_T, WHERE $G_T.prefix = P_T.prefix$ AND $G_T.q$gram IN $G^q(\omega)$ GROUP BY $G_T.prefix$ HAVING COUNT($G_T.q$gram)= $|\omega| + 1 - q - \tau * q$.

False positives may be introduced in this method hence we make use of UDFs to check the candidates to obtain similar prefixes. It could be inefficient to use q-gram based method, and utilization of "GROUP BY" could be expensive in database for mostly large q-gram tables. This method is even inefficient for short query keyword [3] it has low pruning power as short keywords have less number of q-grams. Length filtering may be added to improve performance [1].

NEIGHBORHOOD-GENERATION-BASED METHOD

TABLE 3
Neighborhood-Generation Table ($\tau = 1$)

prefix	*i-deletion*	*i*
vldb	vldb	0
vldb	ldb	1
vldb	vdb	1
vldb	vlb	1
vldb	vld	1
...	...	...

This method was proposed by Ukkonen to support approximate string searching [4]. If a keyword w is given, the substring of w by eliminating or deleting i characters called as "i-deletion neighborhoods" of w. Set of i-deletion neighborhoods of ω is given by $D_i(\omega)$ and $\widehat{D\tau}(\omega) = \bigcup_{i=0}^{r} D_i(\omega)$. For instance, for a string "pvldb", D_0(pvldb) = {pvldb}, and D_1(pvldb) = {vldb, pldb, pvdb, pvlb, pvld}. Assume τ=1, $\widehat{D}\tau$(pvldb) = {pvldb, vldb, pldb, pvdb, pvlb, pvld}. If a user enters an input keyword "pvldb", its prefixes are calculated in D_τ that have i-deletion neighborhoods in {pvldb, vldb, pldb, pvdb, pvlb, pvld}. From this, we come to know that "vldb" is similar to "plvdb" and their edit distance is 1. So we can say that this method is efficient for short strings but for long strings this method is inefficient and especially when the edit distance threshold is even large. To store neighborhoods large space is been required. All these three methods of fuzzy search have some or the other disadvantages. So to overcome these disadvantages an incremental algorithm is proposed which makes use of previously calculated result to answer subsequent queries.

INCREMENTALLY COMPUTING SIMILAR PREFIXES

Chaudhari and Kaushik [5] and Ji et al. [6] proposed to compute similar prefix incrementally. If a user types in a keyword $w = c_1\ c_2\ \ldots\ c_x$ character by character, then for each prefix $p = c_1\ c_2\ \ldots\ c_i\ (i<=x)$, we maintain a similar-prefix table $S^p{}_T$ with records in the form *(prefix, ed(p,prefix))*, which has all the prefixes similar to p and corresponding edit distances. Due to small similar prefix tables, in-memory tables can be used to store them. This similar prefix table may be shared by different queries. To avoid table from getting too big, periodically some of its entries may be removed. So, the incremental-computation algorithm does not maintain session information for different queries. Suppose the user types one more character c_{x+1} and enters a new query $w' = c_1\ c_2\ \ldots\ c_x c_{x+1}$, then we use table $S^w{}_T$ to calculate $S^{w'}{}_T$ and find the range of keywords of similar prefixes in $S^{w'}{}_T$, by joining the similar-prefix table $S^{w'}{}_T$ and the prefix table P_T, and compute the answer of w' using I_T. The following SQL query can be used to answer single keyword query w':

SELECT T.* FROM $S^{w'}{}_T, P_T, I_T, T$ WHERE $S^{w'}{}_T.prefix=P_T$.prefix AND $P_T.ukid \leq I_T.kid$ AND $P_T.lkid \leq I_T.kid$ AND $I_T.rid = T.rid$.

SIMPLE TECHNIQUES FOR FINDING MULTIQUERIES

In this section, we have used techniques to support multi-keyword queries.

COMPUTING ANSWER FROM SCRATCH

Let us assume a multi keyword query Q with m keywords. It can be done in two ways:

1) *By Using Intersect Operator:* In this we first find the records & then by using INTERSECT Operator join these records for multi keywords to compute the answer
2) *By Using Command text:* In this we first find the record by using CONTAINS command to find records which match the first *m-1* complete keywords & then use our methods to match the last prefix keyword .

Due to lack of precomputed results, these two methods may lead to low performance and thus, to overcome these problems we propose an incremental computation method.

INCREMENTAL COMPUTATION BY USING WORD BY WORD

In this method, suppose a user types a query Q with key word as $x_1, x_2, \ldots, x_n$, a temporary table C_Q is created to reserve the record ids of query Q. At that instance, if a user types in a new keyword w_{m+1} and submits a new query Q' with $x_1, x_2 \ldots\ldots x_n, x_{n+1}$ then a temporary table C_Q is utilized to increment the answer.

Exact Search: We check whether the tuples in C_Q contain keywords with prefix w_{m+1} of new query Q'. The SQL query for the same is:

SELECT T.* FROM C_Q, P_T, I_T, T WHERE P_T.prefix=" w_{m+1} " AND $P_T.ukid \geq I_T.kid$ AND $P_T.lkid \leq I_T.kid$ AND $I_T.rid = C_Q.rid$ AND $T.rid = C_Q.rid$.

For example, if a user types in query Q = "privacysigmod" and there is a temporary table $C_Q = \{r_3, r_6\}$. At the same instance if the user types in a new keyword "pub" and issues a new query Q' = "privacysigmodpub", in that case it is checked whether records r_3, r_6 contain a keyword with prefix "pub". With the help of C_Q, we observe that only r_6 contains a keyword "publishing" with prefix "pub".

Fuzzy Search: Firstly $S^{wm+1}{}_T$ is calculated using character level incremental method and then use $S^{wm+1}{}_T$ to answer the query. With the help of temporary table C_Q, the SQL query is:

SELECT T.* FROM $S^{wm+1}{}_T, P_T$, I_T, T WHERE P_T.prefix= $S^{wm+1}{}_T$ .prefix AND $P_T.ukid \geq I_T.kid$ AND $P_T.lkid \leq I_T.kid$ AND $I_T.rid = C_Q.rid$ AND $T.rid = C_Q.rid$.

PROVIDE UPDATES EFFECTIVE

To make updates effective, we have also to consider insertion and deletion of records.

1) *Insertion:* When a record is entered, we first assign the record with a record ID. For each keyword, we add the keyword in P_T. For each new prefix, we insert it into the prefix table. In this way we reserve the space for the prefix keywords.

2) *Deletion:* If a record is deleted, there should be an indication that the prefix is deleted from the prefix table for which we use a bit. If the bit is marked, then the record is deleted. But updating of table is only done when index has to be rebuilt. So, we have to better update the table until we need to restore the indexes.

EXPERIMENTAL SURVEY

After implementing the proposed method in two real data sets (DBLP & MEDLINE), we summarize the data sets and index size into Table 4. From Table 4, we can say that size of I_T and P_T is acceptable as compared to the data size. In a keyword, substring has many overlapped q-grams, so the size of q-grams table is larger. As similar prefix table stores similar prefix of a keyword, so its size is very small. From the log of our deployed system we used 1000 real queries for each data set and assume characters are typed one by one. We work on a Windows 7 OS with a Intel Core 2 Quad processor (X5450 3.00 GHz and 4 GB memory) and three data bases, MYSQL, SQL Server 2005 and Oracle 11g.

TABLE 4
Data Sets and Index Costs

Data Set	MEDLINE	DBLP
# of Records (millions)	5	1.2
Database size	1.5 GB	450 MB
Avg. # of words per record	7.7	17.1
Max. # of words per record	62	172
Min. # of words per record	2	2
# of distinct keywords (millions)	0.7	0.4
Index-construction CPU Time	46 secs	9 secs
Index-construction IO Time	102 secs	18 secs
Size of the inverted-index table	604 MB	126 MB
Size of the prefix table	70 MB	30 MB
Size of the prefix-deletion table ($\tau = 2$)	4.2 GB	1.3 GB
Size of the q-gram table ($\|q\| = 2$)	902 MB	329 MB
Avg. size of the similar-prefix table	3 KB	2 KB

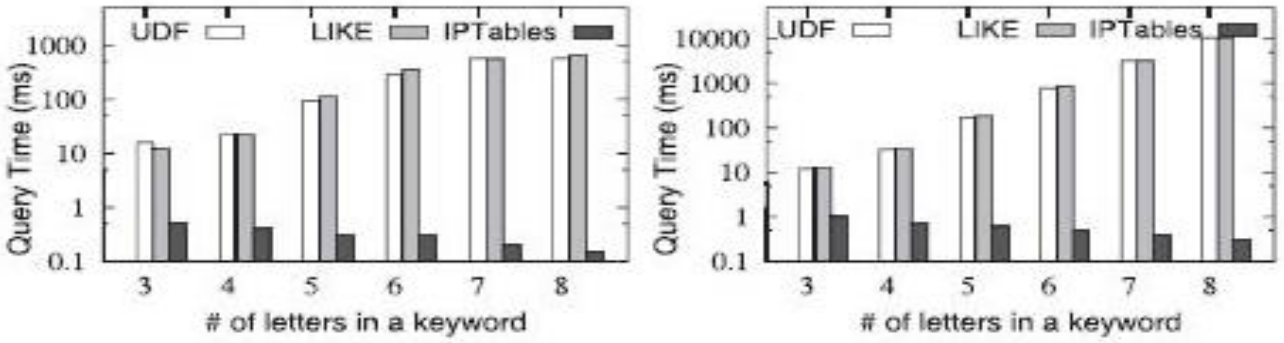
EXACT SEARCH

Single Keyword Queries: In our system for single-keyword queries, we implemented three methods: 1) using UDF, 2) using LIKE predicate and 3) using the IP tables (inverted index and prefix table). Fig. 2 shows the results.

UDF based method and LIKE based method had a low search performance as compared to the IP table as in UDF & LIKE based method, they need to scan records where as in IP table they uses indexes. As the length of keyword increases, the performance of UDF and LIKE based method decreases as they need to scan more records in order to find the same number (*N*) of answers whereas IP tables had a higher performance as there are fewer complete keywords & fewer join operations for the query.

Multi-Keyword queries: We used 6 methods to implement multi keyword queries:

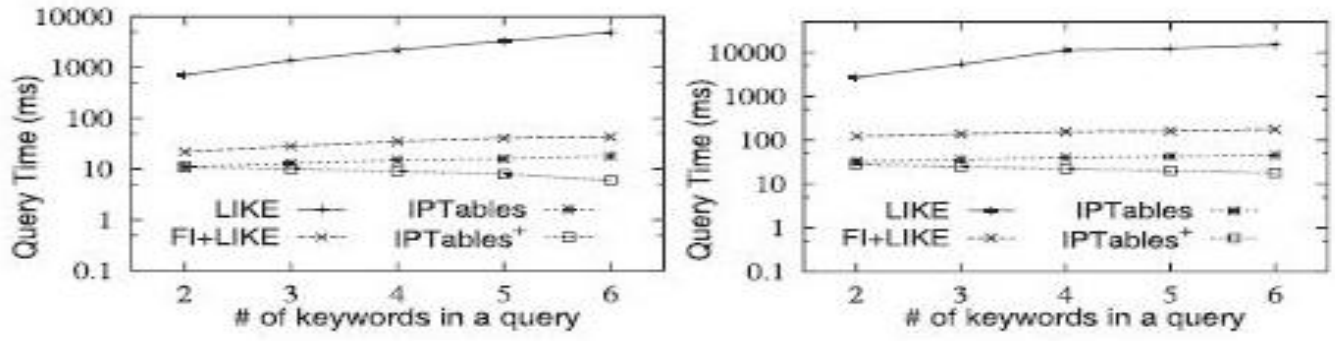
1. Using UDF.
2. Using the LIKE predicate.
3. Using full-text indexes and UDF (called "FI+UDF").
4. Using full-text indexes and the LIKE predicate (called "FI+LIKE").
5. Using the IPT tables (inverted- index table prefix table).
6. Using the IPT tables+ (called word- level incremental method).



(a) Find N answers(DBLP) (b) Find N answers(MEDLINE)

Fig. 2. Exact-search performance for answering single-keyword queries (varying keyword length).

From figure 3, we see that LIKE based method has low performance. Full-text indexes gives better performance. For example, on the MEDLINE data set, LIKE based method took higher time and later on method took less time and IPT table+ achieved the highest performance.



(a) Find N answers(DBLP) (b) Find N answers(MEDLINE)

Fig. 3. Exact-search performance of answering multikeyword queries (varying keyword numbers).

Varying the number of answers N: We compared the performance of methods to compute first *N* answers by varying the results which is shown in Fig 4. From the figure, we observe that IP table had highest performance for the single keyword whereas IP table+ outperformed other methods for the multi-keyword for different values of N.

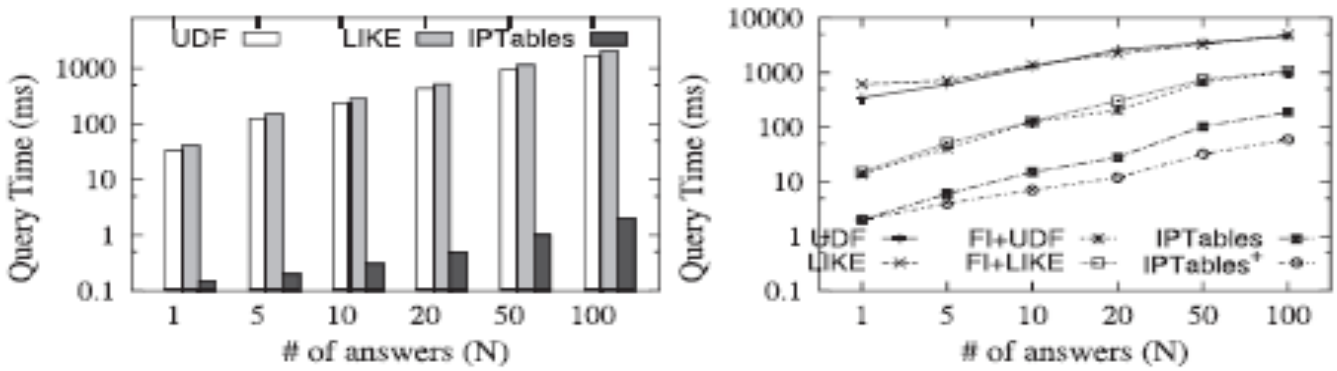


Fig. 4. Exact-search performance of computing first-N answers by varying different N values. (a) Single keywords—DBLP. (b) Multi-keywords—MEDLINE.

FUZZY SEARCH

Single-keyword queries: These queries have been implemented by these four methods:

1. Using UDF.
2. Using Gram based method.
3. Using NGB method (called neighborhood-generation-based method).
4. Using INCRE method (called character-level incremental method).

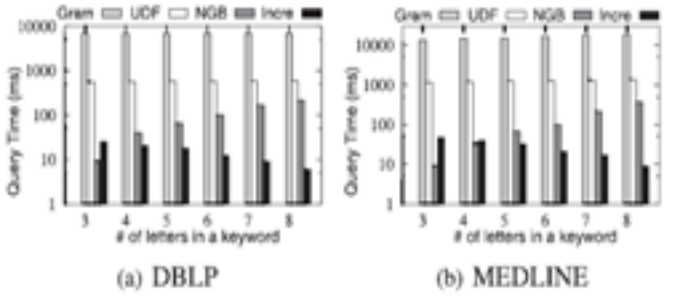
From Fig 5, we conclude that the running time of Gram, NGB & UDF increases while running time of INCRE method decreases. This is due to the fact that UDF took more time for computing long strings, NGB took more time for i-deletion of longer strings and more time was required for large number of grams.

Multi-keyword queries: For multi-keyword queries, we did not use UDF & Gram as they were too slow. We implemented two methods INCRE & NGB to find similar keywords. For multi-keyword queries, we also implemented their word level incremental called NGB+ & INCRE+. From Fig 6. it is clear that INCRE+ had highest performance . We also evaluated the running time in 2 steps:

1. Finding similar keywords (called NGB-SP & INCRE-SP).
2. Computing first *N* answers (called NGB-R & INCRE-R).

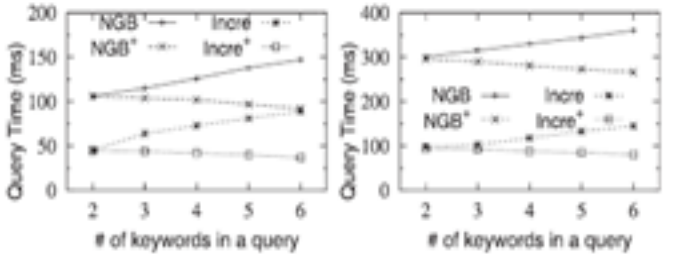
From Fig 7, we can say that NGB & INCRE nearly took the same amount of time.

Varying the number of returned results (N): Fig 8 shows the result for the first –*N* answering by varying *N* from which we can conclude that INCRE+ & NGB+ can efficiently compute.



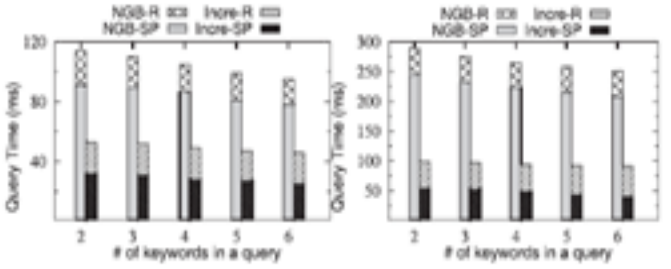
(a) DBLP (b) MEDLINE

Fig. 5. Fuzzy-search performance of computing similar keywords for single-keyword queries by varying the query keyword length ($\tau = 2$).



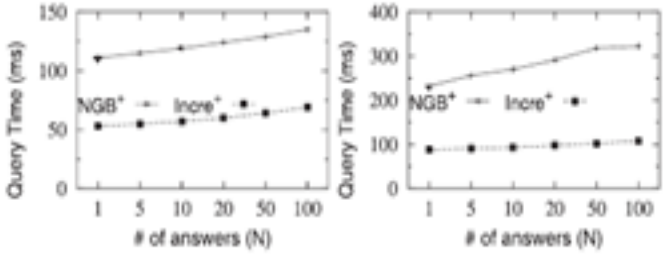
(a) DBLP (b) MEDLINE

Fig. 6 Fuzzy-search performance (overall) of computing first-N answers for multikeyword queries by varying the keyword number in a query ($\tau = 2$).



(a) DBLP (b) MEDLINE

Fig. 7. Fuzzy-search performance (2 steps) of computing first-N answers for multiple-keyword queries by varying keyword numbers in a query ($\tau = 2$).



(a) DBLP (b) MEDLINE

Fig. 8. Fuzzy-search performance of computing first-N answers by varying different N values.

RELATED WORK

The function of auto completion feature is to predict answers to query on the basis of previous parial word typed by the user [7]. This feature was studied in detail by Nandi and Jagadish, also known as phrase prediction. Many other researchers like Bast et al. proposed HYB indexing techniques [8] [9] to support autocomplete searching. Nowadays keyword search has become very important in databases. Recently many techniques have been studied on keyword search [10] [11] [12]. There have been recent developments to support approximate string searching like gram based methods but these are not as better as tri structure in fuzzy search techniques. Our study on search as you type feature includes these earlier studies by thorough investigation of different related methods.

ACKNOWLEDGEMENT

We have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals and organizations. I would like to extend my sincere thanks to all of them.

The completion of any inter-disciplinary project depends upon cooperation, coordination and combined efforts of several sources of knowledge. We are grateful to **Prof. Pranali Lokhande, Department of Computer Engineering** for her even willingness to give us valuable advice and direction; whenever we approached her with a problem. We are thankful to her for providing immense guidance for this project. Our thanks and appreciations also go to our colleagues in developing the project and people who have willingly helped me out with their abilities.

CONCLUSION

In this paper, we work on the problem of using SQL to support search as you type in databases. We discuss on the challenge of how to meet high performance in existing databases. We use prefix matching via auxiliary tables as index structures & SQL queries to support search as you type. We use fuzzy search to improve performances. We use incremental –computation method to answer multikeyword queries and study about incremental updates. Our result on large, real data sets showed that our method can enable DBMS system to support search as you type on large tables.

However, there are several problems to support search as you type using SQL. For example, one is how to support queries efficiently and other is how to support multiple tables.

REFERENCES:

[1] L. Gravano, P.G. Ipeirotis, H.V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava, "Approximate String Joins in a Data Base (Almost) for Free," Proc. 27th Int'l Conf. Very Large Data Bases (VLDB '01), pp. 491-500, 2001.

[2] J. Jestes, F. Li, Z. Yan, and K. Yi, "Probabilistic String Similarity Joins," Proc. Int'l Conf. Management of Data (SIGMOD '10), pp. 327- 338, 2010.

[3] J. Wang, G. Li, and J. Feng, "Trie-Join: Efficient Trie-Based String Similarity Joins with Edit-Distance Constraints," Proc. VLDB Endowment, vol. 3, no. 1, pp. 1219-1230, 2010.

[4] E. Ukkonen, "Finding Approximate Patterns in Strings," J. Algorithms, vol. 6, no. 1, pp. 132-137, 1985.

[5] S. Chaudhuri and R. Kaushik, "Extending Autocompletion to Tolerate Errors," Proc. 35th ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '09), pp. 433-439, 2009.

[6] S. Ji, G. Li, C. Li, and J. Feng, "Efficient Interactive Fuzzy Keyword Search," Proc. 18th ACM SIGMOD Int'l Conf. World Wide Web (WWW), pp. 371-380, 2009.

[7] A. Nandi and H.V. Jagadish, "Effective Phrase Prediction," Proc. 33rd Int'l Conf. Very Large Data Bases (VLDB '07), pp. 219-230, 2007.

[8] H. Bast, A. Chitea, F.M. Suchanek, and I. Weber, "ESTER: Efficient Search on Text, Entities, and Relations," Proc. 30th Ann. Int'l ACM SIGIR Conf. Research and Development in Information Retrieval (SIGIR '07), pp. 671-678, 2007.

[9] H. Bast and I. Weber, "The Complete Search Engine: Interactive, Efficient, and Towards IR & DB Integration," Proc. Conf. Innovative Data Systems Research (CIDR), pp. 88-95, 2007.

[10] S. Agrawal, S. Chaudhuri, and G. Das, "DBXplorer: A System for Keyword-Based Search over Relational Data Bases," Proc. 18th Int'l Conf. Data Eng. (ICDE '02), pp. 5-16, 2002.

[11] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword Searching and Browsing in Data Bases

Using Banks," Proc. 18th Int'l Conf. Data Eng. (ICDE '02), pp. 431- 440, 2002.

[12] F. Liu, C.T. Yu, W. Meng, and A. Chowdhury, "Effective Keyword Search in Relational Data Bases," Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '06), pp. 563-574, 2006.

[13] Wikipedia, "en.wikipedia.org/wiki/Approximate_string_matching"