International Journal of Scientific Research in Computer Science, Engineering and Information Technology
© 2017 IJSRCSEIT | Volume 2 | Issue 3 | ISSN : 2456-3307

Wildcard Search using Fuzzy Auto Completion

Ansari Aadil Salim[1], Ansari Shawana Aadil[2], Ansari Zeeshan Salim[3]
[1] Assistant Professor at MMANTC, Department of Computer Engineering, Malegaon, India
[2]Department of Electronics and Communication, SSVPS BSD COE, Dhule, India
[3]Department of Computer Engineering, SSVPS BSD COE, Dhule, India

ABSTRACT

Most popular information discovery method is through keyword search, as user does not need to know either the underlying structure of the data and a query language. The search engines available today provide keyword search on top of sets of documents. While traditional database management systems offer powerful query languages, they do not allow keyword-based search and we focus on how to support this type of search using the native database language, SQL. Searching in a relational in a relational database is not an easy task because the data present are complicated. Wildcard search is a search with a character that can be used to substitute for any other character(s) in a string. Fuzzy autocompletion method is used to generate the results by typing incomplete keyword character by character. In this paper we are combining the property of both the techniques wildcard search and the fuzzy autocompletion to generate the search results efficiently. Using the above method we present solutions for both single-keyword queries and multi keyword queries, and develop novel techniques for on the fly search using SQL by allowing mismatches between query keywords and answers. Experiments on large, real data sets show that our techniques enable DBMS systems to support on-the-fly search on tables with millions of records.

Keywords : Autocompletion, wildcard, databases, SQL, fuzzy, DBMS.

I. INTRODUCTION

Most information systems nowadays improve user search experiences by providing instant feedback as users formulate search queries. Many search engines and online search forms support auto completion this method shows suggested queries and answers “on the fly” means as a user types in a keyword query character one after the other. In relation to computer technology, "on the fly" means activities that is dynamic in nature rather than as the result of something that is statically predefined [13]. For example, consider the Web search interface at Youtube [19], it allows a user to search for videos related to different genres. If a user types in a partial query “Sta,” the system shows videos with a title matching this keyword as a prefix, such as “Star” and “StarWars.” The instant feedback helps the user to formulate the query also in understanding the underlying data. This type of search is generally called on-the-fly, search-as-you-type or type-ahead search. Since information in a search systems is stored in the backend relational DBMS, so one of the important challenge is to support search-as-you-type on the data residing in a DBMS. Databases such as SQL server and Oracle already support prefix search, and we could use this feature to do search-as-you-type. However, not all databases provide this feature. For this reason, we study different methods that can be used in all databases. Predictions in searching are possible search terms you can use that are related to the terms you’re typing and what other people are looking for ,so we are implementing this feature with Fuzzy technique. We are going to use Fuzzy autocompletion feature which can help in formulating queries in addition with the Wildcard. A wildcard character can be used to substitute for any other character(s) in a string which reduce the efforts of knowing the correct spelling of the keyword.

In this paper we study how to support type ahead search on DBMS systems using Wildcard with fuzzy autocompletion. Here we going to find answers to a search query as a user types in keywords character by character.

II. LITERATURE SURVEY

We have studied different techniques to search on SQL Relational databases below are some of them.

Nandi et al studied the problem of autocompletion at the level of a single "word", and also at level of a multi-word "phrase". They found two main challenges: one was with the number of phrases the second is that a "phrase", which is not a "word", it does not have a well-defined boundary, so that the autocompletion system has to decide not just what to predict, but also how far [1].

Li et al. studied the use of auxiliary indexes stored as tables to increase search performance. They demonstrated searching in a database using many different techniques and compared them to show that which can work better. They present solutions for single-keyword and multikeyword queries, and developed the technique for fuzzy search using SQL [13].

Li et al. proposed a search method DBease which make databases user-friendly and easily available. It allows users to explore data "on the fly" as they type in keywords. They developed various techniques to improve keyword search, form-based search, and SQL-based search for enhancing user experiences. Search as-you-type help user's on-the-fly to explore the underlying data. Form-based search can provide on-the-fly faceted search [8].

Feng et al proposed a concept called Compact Steiner Tree (CSTree), which can be used to approximate the Steiner tree problem for answering top-k keyword queries efficiently. To benefit from its indexing and query-processing capability their proposed technique can be implemented using a standard relational RDBMS. This techniques was implemented in MYSQL, which can provide built-in keyword-search capabilities using SQL [12].

Ji et al. proposed a information-access paradigm, called "interactive, fuzzy search," the system which searches the data during the typing work as the user types in query keywords. Autocomplete interfaces was extended by allowing keywords to appear in multiple attributes (in an arbitrary order) of the underlying data; and finding related records that have keywords matching query keywords approximately. Their framework allowed users to explore data as they type [4].

Chaudhari et al. proposed a method capture input typing errors via edit distance. They show that a native approach of invoking an offline edit distance matching algorithm at each step performs poorly and present more efficient algorithms. Their study demonstrated the effectiveness of algorithms. However they focused on the algorithmic aspects of error-tolerant autocompletion which are relevant regardless of the specific application [5].

Li et al. proposed a novel approach to keyword search in the relational world, called Tastier. A system that can bring instant gratification to users by supporting type-ahead search, it find answers "on the fly" as the user types in query keywords. To find appropriate results on-the-fly by joining tuples in the database they developed an index structures and algorithms. They improved the query performance by grouping relevant tuples and pruning irrelevant tuples efficiently also proposed a method to answer a query efficiently by predicting highly relevant complete queries for the user. Their main challenge is to achieve a interactive high speed for large volume of data in multiple tables, so that within milliseconds, a query can be answered efficiently [6].

Qin et al. proposed a system by using SQL to compute all the interconnected tuple structures for a given keyword query. To control the size of the structures they used three different types of interconnected tuple structures. The main idea behind their approach is tuple reduction. A middleware free approach to compute such m-keyword queries on RDBMSs using SQL only. Their middleware free approach makes it possible to fully utilize the functionality of RDBMSs to support keyword queries in the same framework of RDBMSs [7].

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

ID	Title	Authors	Book title	Year
r1	K-Automorphism: A Framework for Privacy Preserving Network Publication	Lei ,Lei Chen,M. Tamer Ozsu	PVDLB	2009
r2	Privacy Preserving Singular Value Decomposition	Shuguo , Wee Keong ,Philip S.Yu	ICDE	2009
r3	Privacy Preservation Aggregates on Hidden Database:Why and How?	Arjun Dasgupta, Gautam Das, Surajit Chuadhari	SIGMOD	2009
r4	Privacy-preserving Indexing of Documents on the Network	Mayank Bawa, Roberto J. Bayard, , Jaideep Vaidya	VLDBJ	2009
r5	On Anti Corruption Privacy Preserving Publication	Yufei Tao, Xiaokui Xiao, Jiexing Li,Donghui Zhang	ICDE	2008
r6	Preservation of Proximity Privacy in Publishing Numerical Sensitive Data	Jiexing Li, Yufei Tao, Xiaokui Xiao,	SIGMOD	2008
r7	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
r8	The Boundary Between Privacy and Utility in Data Publishing	Vibhor Rastogi, Sungho Hong ,Dan Suciu	VLDB	2007
r9	Privacy Protection in Personalized Search	Xuehea Shen ,Bin Tan, Chen Xiang Zhai	SIGIR	2007
r10	Privacy on Database Publishing	Alin Deutsch, Yannis Papakonstantinou	ICDT	2005

It becomes even more unclear if we want to use features in search-as-you-type, namely multikeyword search and fuzzy search. In multikeyword search, a query string have multiple keywords, and we find the records which matches these keywords, even if the keywords appear at other or different places. Consider, when a user types in a query "Hiding and "Privacy" he will get the resultant string with the record r7.

III. DATA SETS

We are using DBLP datasets in this research work. The *dblp computer science bibliography* is reference for bibliographic information on major computer science publications. It has com eup from an early small scale experimental web server to a popular open-data service for the computer science community. As of May 2016, *dblp* indexes over 3.3 million publications, published by more than 1.8 million authors. To this end, dblp indexes about than 32,000 journal volumes, more than 31,000 conference or workshop proceedings, and more than 23,000 monographs[50].

We collected datasets from dblp and got total of 207792 records whose size is greater than 1.5 gigabytes which we got after extraction. The raw data which we got was in the xml format .we then filtered it in the excel format(.xlsx) and used this datasets in our work.

IV. PROPOSED METHOD

Our project is influenced from the existing search system in various DBMS systems like Oracle, Microsoft SQL server, MySQL etc. Many database system provides prefix searching capability but it is not supported by all the types of database special in the relational database. In the existing sytem for type-ahead or on-the-fly search the authors had developed a sytem which uses two different types of methods for searching a data using relevant keyword conventional method was called as Exact Search in which in which one has type the exact words of the particular record .Consider an example if we make search as "privacy." in a database it will try to find the exact copy word and gives the records where the given word was present and other was using Fuzzy algorithm. Fuzzy helps to find the results if we type the words incorrectly [13]. Both the method was again classified into many different method.

The Problem with the exact search was it takes more time for finding the results and an ignorant user will always find it difficult to search in the given database, one can say that the user was "left in dark". So it is better to neglect this method. The Other Problem was not with the fuzzy algorithm but with the underutilization of the fuzzy algorithm, because it was just useful when we type the word incorrectly. Many Studies have shown that the fuzzy can be useful for finding answer in a database, whenever it will be used different manner. If resources of the Fuzzy are properly

utilized it can generate efficient results at reduced time.

We proposed a system which will makes the efficient use of fuzzy algorithm. We developed the system with "autocompletion feature" which will find the relevant record. It can find proper answer just by typing few number of keywords or we call it few character it will autocomplete our queries at reduced time interval. With the help of this autocompletion feature an unknown user also can make an effective search. So we are adding this autocompletion feature in some existing fuzzy technique.

One of the most important concept of Wildcard we are going to use in combination with the fuzzy autocompletion feature. Wildcard allows minor mismatches in the resultant queries which can produce an effective results.

In this paper our main aim is to support search-as-you-type on DBMS systems using the native query language (SQL). We are going to use SQL to find answers to a search query as a user types in keywords character by character. Our goal is to make effective use of the built-in query engine of the database system as much as possible. In this way, we can reduce the coding headaches to support search-as-you-type.

We applied wildcard techniques in combination with the fuzzy autocompletion to enhance the previous existing results .

3.1 Introduction To Wildcard Technique

A **wildcard** is usually a character that may be used in a **search** term to represent one or more other characters. The two most commonly used **wildcards** are: An asterisk (*) may be used to specify any number of alphanumeric characters [52] [53]. Wildcards allow you to bring various words by using the same common characters without having to type each possibility as a separate keyword. We can use Wildcards or some other universal characters to show the spot in your keyword where system can replace any character. We can use a question mark (**?**) in the place of a character in our search term to indicate that any character can come up at that specific spot i.e same position in the search term.

For example, if we type **advis?r** as search term, we get the results which will include advis***o***r and advis***e***r. In our search term if two or more question marks comes together , then the system makes replacement based on the number of question marks included consecutively in the search term.

For example, if user type **??clude** term for search, your results will include , ***oc***clude, ***in***clude and ***ex***clude but not preclude (because preclude would have required your search term to be **???clude**).
Here are some different kinds of example using wildcard characters in queries:
i) The wildcard query 'f?rm' will match all of the words 'form'", 'firm', and farm -word that begins with an 'f', is followed by any other character, and ends with the 'rm' characters.
ii) The wildcard query '??rm' will match all of the words from the previous bullet, as well as words like 'worm', 'harm', 'term', , and so on - any four character word that ends with the characters 'rm'.
iii)The wildcard query '??ow*rm' will match terms such as 'glowworm', 'slowworm', and 'snowstorm' - any word that begins with any two characters, followed by the character sequence 'ow', followed by any number of other characters, and ending in the character sequence 'rm'.
iv)the wildcard query '*rm' will match all of the words in the previous bullets, as well as words like 'terraform', 'wheatgerm', 'alarm', 'bookworm', 'brainstorm', and so on. Because the '*' wildcard character can represent any sequence composed of any characters, the expression '*rm' matches any term that ends in 'rm'. It may also include matching terms Linux command 'rm', because the wildcard '*' can also match zero characters. The regular expression '?rm' would not match the Linux command 'rm', because the '?' wildcard operator must match a character in the specified position.

The Above wildcard method will be used in every Fuzzy autocompletion technique which we are going to see

3.2 Fuzzy Autocompletion

Let's say we have five people. David, Damarcus, Daniel, Dustin, and Russ. when a user types in *d*. We would match **Du**stin and **D**amarcu**s**. Likewise, if we typed in *us*, we would get an output of D**us**tin, Damarc**us**, and R**us**s. At this point, it's sort of a no-brainer. Your input-based regular expression can be created such as

```
var people = ['David', 'Damascus','Daniel',
'Dustin','Russ'];
function matchPeople(input) {
var reg = new
RegExp(input.split('').join('\\w*').replace(/\W/, "")',
'i');
return people.filter(function(person) {
 if (person.match(reg)) {
return person;
}
});
}
```

Another example is from the table no.1 as a user types in a single partial keyword w character by character, fuzzy search on-the-fly finds records with keywords similar to the query keyword. In Table 1, assuming a user types in a query "dat," record r_8 is a relevant answer since it contains a keyword "Data" with a "dat" partial word it will return with the record r_3 r_6 r_8 $r_{10.}$

V. WILDCARD USING FUZZY

AUTOCOMPLETION.

In our proposed method we are combining the features of Wildcard with the Fuzzy autocompletion and it will be applied in the following methods-

4.1 Using Single Keyword:

i) No Index Method - User Defined Function
ii) Index Method - UDF, Gram Based, Neighborhood Method.

4.2 **Using Multi Keyword:**

i) Using Intersect Operator
ii) Using Full Text Indexes
iii) Word Level Incremental Computation

PRELIMINARIES

We first formulate the problem of on-the-fly search in DBMS and then discuss different approach to support search .
Problem Formulation

Consider T be a relational table having attributes A1, A2, A3 . . ., Al. Set of records R= {r1, r2, r3 . . . , rn} be in T, and the content of record ri in attribute Aj denoted by $r_{i,}[A_j]$. Let W be the set of tokenized keywords in R.

4.1 Using Single Keyword:

4.1.1 No Index Method – UDF

USER DEFINED FUNCTION (UDF)

A straightforward way to support search-as-you-type is to issue an SQL query that scans each record and verifies whether the record is an answer to the query. Functions can be added into databases to verify whether a record contains the query keyword Consider an Example if we are searching for a query keyword "ri" so we can get results from record which contain the "tric" anywhere in the word that might be an electric or Trick . It can be prefix or postfix or can be Infix. It will find the resultant anwer very fast and the precision also increases as the number of keyword will increase. We use a UDF PXD (w,s) that takes a keyword w and a string s as two parameters, and returns the minimal extend distance between w and the prefixes of keywords in s.

PXD(‘‘pri’’,r_{10}[title])

= PXD(‘‘pri’’, ‘‘privacy in database publishing)=4

as r10 contains a prefix "pri" with extend distance of 4 to the query. We can improve the performance by doing early termination in the dynamic-programming computation

4.1.1 Index Based Method

As index structures we an build auxiliary tables to have prefix search. SQL server and Oracle are databases that already support prefix search .we use this to do prefix search. However, this feature is not provided by all database. So we developed a new method to be used in all database.

Inverted-index table. We developed this table and assign ids to the keywords in table T by their alphabetical order then added an inverted-index table I_T with records in the form (kid, rid), where "kid" is id of a keyword and "rid" is id of a record which have the

keyword. If given a complete keyword, we can use the inverted-index table to find records with the keyword.

Prefix table. For all prefixes of keywords we build a prefix table P_T with records (p, lkid, ukid), where p is a prefix of a keyword, lkid is the smallest id of those keywords in the table T having p as a prefix, and ukid is the largest id of those keywords having p as a prefix. An interesting thing is that a complete word with p as a prefix must have an ID in the keyword range (lkid, ukid), and each complete word in the table T with an ID in this keyword range must have a prefix p. Thus, for given a prefix keyword w, we can use the prefix table to find the range of keywords with the prefix.

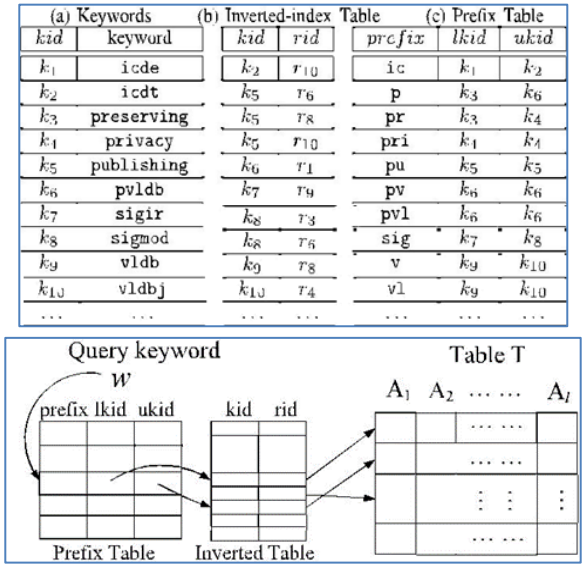

TABLE 2
The Inverted-Index Table and Prefix table

For example, Table 2 shows the prefix table and the inverted-index table for the records in Table 1. The inverted-index table has a tuple (k_8, r_3) since keyword k_8 ("sigmod") is in record r_3. since keyword k_7 . ("sigir") is the minimal id of keywords with a prefix "sig," and keyword k_8 ("sigmod") is the maximal id of keywords with a prefix "sig" as the prefix table has a tuple (''sig''k_7; k_8)." The ids of keywords with a prefix "sig" must be in the range (k_7, k_8) .If we type a partial keyword w, we get its keyword range (lkid, ukid) using the prefix table P_T , and then find the records that have a keyword in the range through the inverted-index table I_T as shown in Fig. 1.

Index Based UDF

Given a keyword w, we use a UDF to find its similar prefixes from the prefix table P_T . Each prefix in a SQL query can be scan in P_T and calls the UDF to check if the prefix is similar to w. We issue the following SQL query to answer the prefix-search query w:

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

We can use length filtering to improve the performance, by adding the following clause to the where clause:

"LENGTH(P_T .prefix) LENGTH(w) + AND
LENGTH(P_T .prefix) LENGTH(w)- $_T$ ".

Gram-Based Method

There are many q-gram-based methods to support approximate string search. Given a string s, its q-grams are its

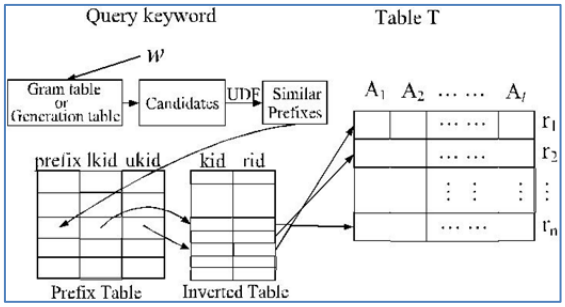

Figure 1. q-gram table and the neighborhood generation table to support fuzzy search

substrings with length q. Let $G^q(s)$ denote the set[4] of its q-grams and | $G^q(s)$ | denote the size of $G^q(s)$. For example, for "pvldb" and "vldb," we have G^2 (pvldb) = {pv, vl, ld, db} and G^2 (vldb) = {vl, ld, db}. Strings s_1 and s_2 have an edit distance within threshold $_T$ if

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

where $|s_1|$ and $|s_2|$ are the lengths of string s_1 and s_2, respectively. This technique is called count filtering. To find similar prefixes of a query keyword w, besides maintaining the inverted-index table and the prefix table, we need to create a q-gram table G_T with records in the form <p, qgram>, where p is a prefix in the prefix table and qgram is a q-gram of p.

Neighborhood Based Method

Ukkonen proposed a neighborhood-generation-based method to support approximate string search [14]. We

have extended their method to use SQL to support fuzzy on the fly search. Given below Table 3 gives a neighborhood-generation table. Consider when a user types in a keyword "pvldb," we get the prefixes in D_T that have i-deletion neighborhoods in {"pvldb," "vldb," "pldb," "pvdb," "pvlb," "pvld"}. Here we find "vldb" similar to "pvldb" with edit distance 1.

This method is good for short strings. But on the otherhand, it is inefficient for long strings, especially for large edit-distance thresholds, because given a string with length n, it has $\binom{n}{i}$- deletion neighborhoods and totally O(min(n^T, 2^n)) neighborhoods. It needs large space to store these neighborhoods.

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

TABLE 3 Neighborhood-Generation Table ($_T$ = 1)

4.2 Multikeyword Queries

In this section, we study efficient techniques to support multikeyword queries.

4.2.1 Computing Answers from Scratch

Here we have Intersect operator and Full text Indexes techniques.

Given a multikeyword query Q with m keywords w_1 , w_2 , w_3 ., w_m there are two ways to answer it from scratch.

"INTERSECT" Operator:

This is a Simple way to first compute the records for each and every keyword using the previous methods, and then to join these records for different keywords to compute the answers use the "INTERSECT" operator

Full-text Indexes:

Full-text indexes CONTAINS command) to find records matching the first m complete keywords, and then use our methods to find records matching the last prefix keyword. Lastly, we join the results. These two methods cannot use the precomputed results and may lead to low performance. To solve this problem we used Word level incremental computation

4.2.2 Word-Level Incremental Computation

We used previously generated results to incrementally answer a query. Consider a user has typed in a query Q with keywords w_1, w_2 .. w_m, we create a temporary table C_Q to cache the record ids of query Q. If the user types in a new keyword $w_{mþ1}$ and submits a new query Q' with keywords w_1 , w_2 .. w_m, w_{m+1} .We use temporary table C_Q to incrementally answer the new query.

As an example, we focus on the method that uses the prefix table and inverted-index table. As C_Q contains all results for query Q, we check whether the records in C_Q contain keywords with the prefix w_{m+1} of new query Q'. We issue the following SQL query to answer keyword query Q' using C_Q:

SELECT T * FROM P_T I_T, C_Q , T
WHERE P_T .prefix = "w_{m+1}" AND
P_T . ukid ≥ I_T .kid AND P_T .lkid ≤ I_T .kid AND
I_T .rid = C_Q:rid AND C_Q.rid = T.rid.

In Fuzzy search as an example, we can consider the character-level incremental method. We compute S_T^{Wm+1} first by using the character-level incremental method for the new keyword $w_{mþ1}$, and then use $S_T^{wmþ1}$ to answer the query. Based on the temporary table C_Q, we use the following SQL query to answer Q'
SELECT T: FROM S_T^{Wm+1} , P_T , I_T, C_Q, T.
WHERE S_T^{Wm+1} .prefix = P_T .prefix AND
P_T .ukid ≥ I_T .kid AND P_T .lkid ≤ I_T . kid AND
I_T.rid = C_Q.rid AND C_Q.rid = T.rid.

If the user modifies the keyword w_m of query Q to $w'_{m'}$ and submits a query with keywords w_1, w_2 .. w_{m-1},$w'_{m'}$, we can use the cached result of query w_1, w_2 .. w_{m-1} to answer the new query using the above method. Similarly, if The user arbitrarily modifies the query, we can easily extend this method to answer the new query.

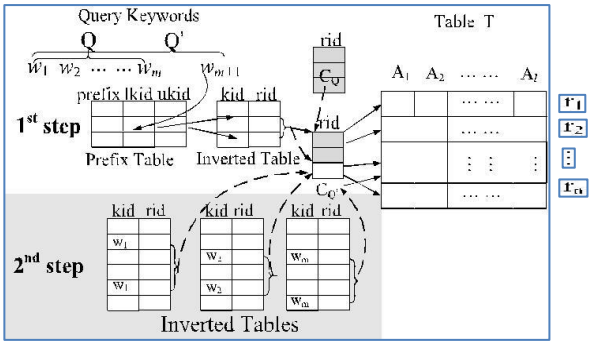

Figure 3. Incrementally computing first-N answers

VI. EXPERIMENTAL STUDY

We implemented the proposed methods on real "DBLP" data set . It included more than 2 million computer science publications. We are generating a Log file which gives us the last used technique in the search performed which gives us the time and space required for producing the results example,
Fuzzy-Index-UDF - TIME(ms): 782 and SPACE: 1082456

We used a Windows 10 machine with an Intel Core i3-processor (4005U 1.70 GHz and 4 GB memory). We used the data base SQL Server 2014 Management Studio.

VII. RESULTS

Wildcard Search using Fuzzy Autocompletion For Single keyword

Single-keyword queries. We first evaluated the performance of different methods to compute similar keywords of single-keyword queries. We implemented four methods:

1. using No Index UDF,
2. using Index UDF
3. using the gram-based method (called "Gram") ,
4. using the neighborhood-generation-based method (called "NGB");

Memory Usage (MB)		
Fuzzy Single Keyword Search	Simple	Wildcards
No-index (UDF)	243	231
Index (UDF)	198	172
Index (Gram-Based)	239	239
Index (Neighborhood)	241	236

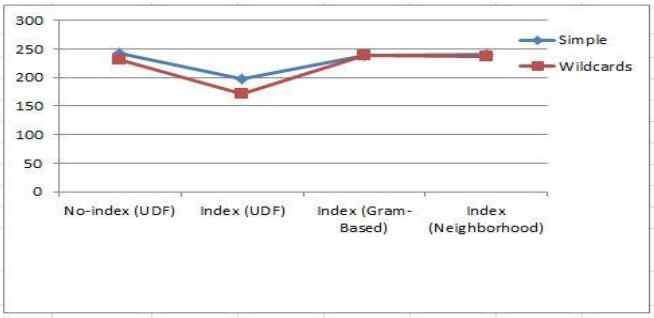

Figure 2. Graph of Single keyword Search –Simple Fuzzy vs Wilcard using Fuzzy Autocompletion

So here are the Comparison of the simple fuzzy search and Wildcard with Fuzzy Autocompletion . The performance time is same but the memory utilized by Wildcard is less.

Wildcard Search using Fuzzy Autocompletion For Multi keyword

Multikeyword queries.

For multikeyword we used three different methods

1.Intersect Operator
2.Full Text Indexes
3.Word Level Incremental

So here are the Comparison of the simple fuzzy search and our technique Wildcard with Fuzzy Autocompletion

Memory Usage (MB)		
Fuzzy Multikeyword Search	Simple	Wildcards
Computing Answers from scratch (INTERSECT)	298	272
Computing Answers from scratch (Full Text)	262	257
Word Level Incremental Computation	371	365

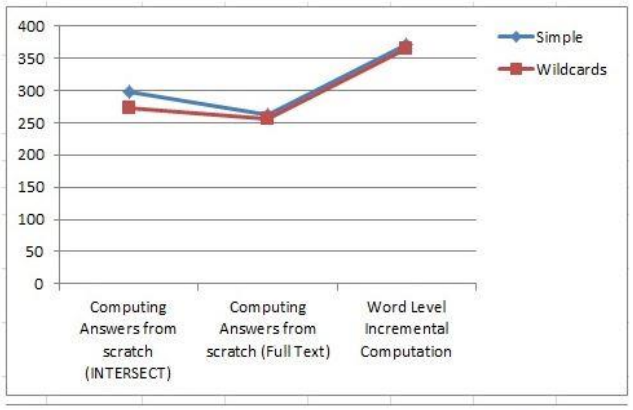

Figure 3. Multi keyword search Graph of Wildcard Fuzzy Autocompletion vs Simple Fuzzy

We can see that the memory utilized by our proposed is less than the simple fuzzy and UDF perform better than the other technique.

VIII. DATA UPDATES

We tested the cost of updates on the DBLP data set. We first built indexes for 1.5 million records, and then inserted 10,000 records at each time. We compared the performance of the different methods on inserting 207792 records. It took more than 40 seconds to reindex the data, while our incremental-indexing method only took 0.5 seconds.

Summary. 1) In order to achieve a high speed, we have to rely on index-based methods. 2) The approach using inverted-index tables and the prefix tables can support prefix, fuzzy search, and achieve the best performance among all these methods and outperform the built-in methods in SQL Server and Oracle. 3) Our SQL-based method can achieve a high interactive speed and scale well.

IX. CONCLUSION

Keyword search in different scenarios enables information discovery without requiring from the user to know the schema of the database. With use of Fuzzy autocompletion we get the resultant answer in less time as compared with Simple Fuzzy. Undergoing with the techniques we get a conclusion that Wildcard method when combined with Fuzzy autocompletion method on different search techniques reduces the memory required for storing the data. We enhanced the existing methods of Fuzzy and developed a novel approach of Wildcard using Fuzzy Autocompletion which performs very well by reducing time and memory required and give more precise results.

X. REFERENCES

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

[2]. 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.

[3]. H. Bast and I. Weber, “Type Less, Find More: Fast Autocompletion Search with a Succinct Index,” Proc. 29th Ann. Int’l ACM SIGIR Conf. Research and Development in Information Retrieval (SIGIR ’06), pp. 364-371, 2006.

[4]. 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.

[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]. G. Li, S. Ji, C. Li, and J. Feng, “Efficient Type-Ahead Search on Relational Data: A Tastier Approach,” Proc. 35th ACM SIGMOD Int’l Conf. Management of Data (SIGMOD ’09), pp. 695-706, 2009.

[7]. L. Qin, J.X. Yu, and L. Chang, “Keyword Search in Data Bases: The Power of Rdbms,” Proc. 35th ACM SIGMOD Int’l Conf. Management of Data (SIGMOD ’09), pp. 681-694, 2009.

[8]. G. Li, J. Fan, H. Wu, J. Wang, and J. Feng, “Dbease: Making Data Bases User-Friendly and Easily Accessible,” Proc. Conf. Innovative Data Systems Research (CIDR), pp. 45-56, 2011.

[9]. 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.

[10]. S. Chaudhuri, K. Ganjam, V. Ganti, R. Kapoor, V. Narasayya, and T. Vassilakis, “Data Cleaning in Microsoft SQL Server 2005,” Proc. ACM SIGMOD Int’l Conf. Management of Data (SIGMOD ’05), pp. 918-920, 2005.

[11]. S. Agrawal, K. Chakrabarti, S. Chaudhuri, and V. Ganti, "Scalable Ad-Hoc Entity Extraction from Text Collections," Proc. VLDB Endowment, vol. 1, no. 1, pp. 945-957, 2008.
[12]. G. Li, J. Feng, X. Zhou, and J. Wang, "Providing Built-in Keyword Search Capabilities in Rdbms," VLDB J., vol. 20, no. 1, pp. 1-19, 2011.
[13]. G. Li, J. Feng and Chen Li," Supporting Search-As-You-Type Using SQL in Databases," IEEE transactions on knowledge and data engineering, vol. 25, no. 2, february 2013
[14]. E. Ukkonen, "Finding Approximate Patterns in Strings," J. Algorithms, vol. 6, no. 1, pp. 132-137, 1985.
[15]. http://dblp.uni-trier.de/
[16]. http://www.dustindiaz.com/autocomplete-fuzzy-matching
[17]. http://en.wikipedia.org/wiki/Wildcard_character
[18]. http://www.w3schools.com/sql/sql_wildcards.asp
[19]. http://www.youtube.com