Supporting Search-As-You-Type Using SQL in Databases

Guoliang Li, Jianhua Feng, *Member*, *IEEE*, and Chen Li, *Member*, *IEEE*

Abstract—A search-as-you-type system computes answers on-the-fly as a user types in a keyword query character by character. We study how to support search-as-you-type on data residing in a relational DBMS. We focus on how to support this type of search using the native database language, SQL. A main challenge is how to leverage existing database functionalities to meet the high-performance requirement to achieve an interactive speed. We study how to use auxiliary indexes stored as tables to increase search performance. We present solutions for both single-keyword queries and multikeyword queries, and develop novel techniques for fuzzy search using SQL by allowing mismatches between query keywords and answers. We present techniques to answer first-N queries and discuss how to support updates efficiently. Experiments on large, real data sets show that our techniques enable DBMS systems on a commodity computer to support search-as-you-type on tables with millions of records.

Index Terms—Search-as-you-type, databases, SQL, fuzzy search

1 Introduction

Many information systems nowadays improve user search experiences by providing instant feedback as users formulate search queries. Most search engines and online search forms support autocompletion, which shows suggested queries or even answers "on the fly" as a user types in a keyword query character by character. For instance, consider the Web search interface at Netflix,[1] which allows a user to search for movie information. If a user types in a partial query "`mad`," the system shows movies with a title matching this keyword as a prefix, such as "`Madagascar`" and "`Mad Men: Season 1.`" The instant feedback helps the user not only in formulating the query, but also in understanding the underlying data. This type of search is generally called *search-as-you-type* or *type-ahead search*.

Since many search systems store their information in a backend relational DBMS, a question arises naturally: how to support search-as-you-type on the data residing in a DBMS? Some databases such as Oracle and SQL server 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 new methods that can be used in all databases. One approach is to develop a separate application layer on the database to construct indexes, and implement algorithms for answering queries. While this approach has the advantage of achieving a high performance, its main drawback is duplicating data and indexes, resulting in additional hardware costs. Another approach is to use database extenders, such as DB2 Extenders, Informix DataBlades, Microsoft SQL Server Common Language Runtime (CLR) integration, and Oracle Cartridges, which allow developers to implement new functionalities to a DBMS. This approach is not feasible for databases that do not provide such an extender interface, such as MySQL. Since it needs to utilize proprietary interfaces provided by database vendors, a solution for one database may not be portable to others. In addition, an extender-based solution, especially those implemented in C/C++, could cause serious reliability and security problems to database engines.

In this paper we study how to support search-as-you-type on DBMS systems using the native query language (SQL). In other words, we want to use SQL to find answers to a search query as a user types in keywords character by character. Our goal is to utilize the built-in query engine of the database system as much as possible. In this way, we can reduce the programming efforts to support search-as-you-type. In addition, the solution developed on one database using standard SQL techniques is portable to other databases which support the same standard. Similar observation are also made by Gravano et al. [17] and Jestes et al. [23] which use SQL to support similarity join in databases.

A main question when adopting this attractive idea is: *Is it feasible and scalable?* In particular, can SQL meet the high-performance requirement to implement an interactive search interface? Studies have shown that such an interface requires each query be answered within 100 milliseconds [38]. DBMS systems are not specially designed for keyword queries, making it more challenging to support search-as-you-type. As we will see later in this paper, some important functionalities to support search-as-you-type require join operations, which could be rather expensive to execute by the query engine.

1. http://www.netflix.com/BrowseSelection.

- *G. Li and J. Feng are with the Tsinghua National Laboratory for Information Science and Technology, Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: {liguoliang, fengjh}@tsinghua.edu.cn.*
- *C. Li is with the Department of Computer Science, School of Information and Computer Sciences, University of California, Irvine, CA 92697-3435. E-mail: chenli@ics.uci.edu.*

Manuscript received 30 Dec. 2010; revised 6 May 2011; accepted 9 June 2011; published online 23 June 2011.
Recommended for acceptance by P. Ipeirotis.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2010-12-0704.
Digital Object Identifier no. 10.1109/TKDE.2011.148.

Authorized licensed use limited to: St. Xavier's Catholic College of Engineering. Downloaded on October 22,2024 at 09:44:08 UTC from IEEE Xplore. Restrictions apply.
1041-4347/13/$31.00 © 2013 IEEE Published by the IEEE Computer Society

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

The scalability becomes even more unclear if we want to support two useful features in search-as-you-type, namely *multikeyword search* and *fuzzy search*. In multikeyword search, we allow a query string to have multiple keywords, and find records that match these keywords, even if the keywords appear at different places. For instance, we allow a user who types in a query "`privacy mining rak`" to find a publication by "Rakesh Agrawal" with a title including the keywords "`privacy`" and "`mining,`" even though these keywords are at different places in the record. In fuzzy search, we want to allow minor mismatches between query keywords and answers. For instance, a partial query "`aggraw`" should find a record with a keyword "`agrawal`" despite the typo in the query. While these features can further improve user search experiences, supporting them makes it even more challenging to do search-as-you-type inside DBMS systems.

In this paper, we develop various techniques to address these challenges. In Section 3, we propose two types of methods to support search-as-you-type for single-keyword queries, based on whether they require additional index structures stored as auxiliary tables. We discuss the methods that use SQL to scan a table and verify each record by calling a user-defined function (UDF) or using the LIKE predicate. We study how to use auxiliary tables to increase performance.

In Section 4, we study how to support fuzzy search for single-keyword queries. We discuss a gram-based method and a UDF-based method. As the two methods have a low performance, we propose a new neighborhood-generation-based method, using the idea that two strings are similar only if they have common neighbors obtained by deleting characters. To further improve the performance, we propose to incrementally answer a query by using previously computed results and utilizing built-in indexes on key attributes.

In Section 5, we extend the techniques to support multikeyword queries. We develop a word-level incremental method to efficiently answer multikeyword queries. Notice that when deployed in a Web application, the incremental-computation algorithms do not need to maintain session information, since the results of earlier queries are stored inside the database and shared by future queries. We propose efficient techniques to progressively find the first-N answers in Section 6. We also discuss how to support updates efficiently in Section 7.

We have conducted a thorough experimental evaluation using large, real data sets in Section 8. We compare the advantages and limitations of different approaches for search-as-you-type. The results show that our SQL-based techniques enable DBMS systems running on a commodity computer to support search-as-you-type on tables with millions of records.

It is worth emphasizing that although our method shares an incremental-search idea with the earlier results in [24], developing new techniques in a DBMS environment is technically very challenging. A main challenge is how to utilize the limited expressive power of the SQL language (compared with other languages such as C++ and Java) to support efficient search. We study how to use the available resources inside a DBMS, such as the capabilities to build auxiliary tables, to improve query performance. An interesting observation is that despite the fact we need SQL queries with join operations, using carefully designed auxiliary tables, built-in indexes on key attributes, foreign-key constraints, and incremental algorithms using cached results, these SQL queries can be executed efficiently by the DBMS engine to achieve a high speed.

2 Preliminaries

We first formulate the problem of search-as-you-type in DBMS (Section 2.1) and then discuss different ways to support search-as-you-type (Section 2.2).

2.1 Problem Formulation

Let T be a relational table with attributes $A_1, A_2, \ldots, A_\ell$. Let $R = \{r_1, r_2, \ldots, r_n\}$ 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.

2.1.1 Search-as-You-Type for Single-keyword Queries

Exact Search: As a user types in a single partial (prefix) keyword w character by character, search-as-you-type on-the-fly finds the records that contain keywords with a prefix w. We call this search paradigm *prefix search*. Without loss of generality, each tokenized keyword in the data set and queries is assumed to use lower case characters. For example, consider the data in Table 1, $A_1 =$ title, $A_2 =$ authors, $A_3 =$ booktitle, and $A_4 =$ year. $R = \{r_1, \ldots, r_{10}\}$. r_3[booktitle] = "`sigmod`".

Authorized licensed use limited to: St. Xavier's Catholic College of Engineering. Downloaded on October 22,2024 at 09:44:08 UTC from IEEE Xplore. Restrictions apply.

$$W = \{\texttt{privacy}, \texttt{sigmod}, \texttt{sigir}, \ldots\}.$$

If a user types in a query "sig," we return records r_3, r_6, and r_9. In particular, r_3 contains a keyword "sigmod" with a prefix "sig."

Fuzzy Search: 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 "corel," record r_7 is a relevant answer since it contains a keyword "correlation" with a prefix "correl" similar to the query keyword "corel."

We use edit distance to measure the similarity between strings. Formally, the edit distance between two strings s_1 and s_2, denoted by $\mathrm{ed}(s_1, s_2)$, is the minimum number of single-character edit operations (i.e., insertion, deletion, and substitution) needed to transform s_1 to s_2. For example, ed (corelation, correlation) $= 1$ and ed (coralation, correlation) $= 2$. Given an edit-distance threshold τ, we say a prefix p of a keyword in W is *similar to* the partial keyword w if $\mathrm{ed}(p, w) \leq \tau$. We say a keyword d in W is *similar to* the partial keyword w if d has a prefix p such that $\mathrm{ed}(p, w) \leq \tau$. Fuzzy search finds the records with keywords similar to the query keywords.

2.1.2 Search-as-You-Type for Multikeyword Queries

Exact Search: Given a multikeyword query Q with m keywords $w_1, w_2, \ldots, w_m$, as the user is completing the last keyword w_m, we treat w_m as a partial keyword and other keywords as complete keywords.[2] As a user types in query Q character by character, search-as-you-type on-the-fly finds the records that contain the complete keywords and a keyword with a prefix w_m. For example, if a user types in a query "privacysig," search-as-you-type returns records r_3, r_6, and r_9. In particular, r_3 contains the complete keyword "privacy" and a keyword "sigmod" with a prefix "sig."

Fuzzy Search: Fuzzy search on-the-fly finds the records that contain keywords *similar to* the complete keywords and a keyword with a prefix *similar to* partial keyword w_m. For instance, suppose edit-distance threshold $\tau = 1$. Assuming a user types in a query "privicycorel," fuzzy type-ahead search returns record r_7 since it contains a keyword "privacy" similar to the complete keyword "privicy" and a keyword "correlation" with a prefix "correl" similar to the partial keyword "corel."

2.2 Different Approaches for Search-as-You-Type

We discuss different possible methods to support search-as-you-type and give their advantages and limitations.

The first method is to use a separate application layer, which can achieve a very high performance as it can use various programming languages and complex data structures. However, it is isolated from the DBMS systems.

The second method is to use database extenders. However, this extension-based method is "not safe" to the query engine, which could cause reliability and security problems to the database engine. This method depends on the API of the specific DBMS being used, and different DBMS systems have different APIs. Moreover, this method does not work if a DBMS system has no this extender feature, e.g., MySQL.

The third method is to use SQL. The SQL-based method is more compatible since it is using the standard SQL. Even if DBMS systems do not provide the search-as-you-type extension feature (indeed no DBMS systems provide such an extension), the SQL-based method can also be used in this case. Thus, the SQL-based method is more portable to a different platform than the first two methods.

In this paper, we focus on the SQL-based method and develop various techniques to achieve a high interactive speed.

3 Exact Search for Single Keyword

This section proposes two types of methods to use SQL to support search-as-you-type for single-keyword queries. In Section 3.1, we discuss no-index methods. In Section 3.2, we build auxiliary tables as index structures to answer a query.

3.1 No-Index Methods

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. There are two ways to do the checking: 1) Calling User-Defined Functions (UDFs). We can add functions into databases to verify whether a record contains the query keyword; and 2) Using the LIKE predicate. Databases provide a LIKE predicate to allow users to perform string matching. We can use the LIKE predicate to check whether a record contains the query keyword. This method may introduce false positives, e.g., keyword "publ*ic*ation" contains the query string "ic," but the keyword does not have the query string "ic" as a prefix. We can remove these false positives by calling UDFs. The two no-index methods need no additional space, but they may not scale since they need to scan all records in the table (Section 8 gives the results.).

3.2 Index-Based Methods

In this section, we propose to build auxiliary tables as index structures to facilitate prefix search. Some databases such as Oracle and SQL server already support prefix search, and we could use this feature to do prefix search. However, not all databases provide this feature. For this reason, we develop a new method that can be used in all databases. In addition, our experiments in Section 8.3 show that our method performs prefix search more efficiently.

Inverted-index table. Given a table T, we assign unique ids to the keywords in table T, following their alphabetical order. We create an inverted-index table I_T with records in the form $\langle kid, rid \rangle$, where kid is the id of a keyword and rid is the id of a record that contains the keyword. Given a complete keyword, we can use the inverted-index table to find records with the keyword.

Prefix table. Given a table T, for all prefixes of keywords in the table, we build a prefix table P_T with records in the form $\langle p, lkid, ukid \rangle$, 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 observation 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, given a prefix keyword w, we can use the prefix table to find the range of keywords with the prefix.

2. Our method can be easily extended to the case that every keyword is treated as a partial keyword.

Authorized licensed use limited to: St. Xavier's Catholic College of Engineering. Downloaded on October 22,2024 at 09:44:08 UTC from IEEE Xplore. Restrictions apply.

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}
...	...	...

For example, Table 2 illustrates the inverted-index table and the prefix table for the records in Table 1.[3] The inverted-index table has a tuple $\langle k_8, r_3\rangle$ since keyword k_8 ("sigmod") is in record r_3. The prefix table has a tuple $\langle$"sig"$k_7, k_8\rangle$ 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." The ids of keywords with a prefix "sig" must be in the range $[k_7, k_8]$.

Given a partial keyword w, we first 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. We use the following SQL to answer the prefix-search query w:

SELECT $T.^*$ FROM P_T, I_T, T
WHERE P_T.prefix $=$ "w" AND
$P_T.ukid \geq I_T.kid$ AND $P_T.lkid \leq I_T.kid$ AND
$I_T.rid = T.rid$.

For example, assuming a user types in a partial query "sig" on table dblp (Table 1), we issue the following SQL:

SELECT dblp.* FROM $P_{\text{dblp}}, I_{\text{dblp}}$, dblp
WHERE P_{dblp}.prefix $=$ "sig" AND
$P_{\text{dblp}}.ukid \geq I_{\text{dblp}}.kid$ AND $P_{\text{dblp}}.lkid \leq I_{\text{dblp}}.kid$ AND
$I_{\text{dblp}}.rid =$ dblp.rid,

which returns records r_3, r_6, and r_9. The SQL query first finds the keyword range $[k_7, k_8]$ based on the prefix table. Then it finds the records containing a keyword with ID in $[k_7, k_8]$ using the inverted-index table.

To answer the SQL query efficiently, we create built-in indexes on attributes *prefix*, *kid*, and *rid*. The SQL could first use the index on *prefix* to find the keyword range, and then compute the answers using the indexes on *kid* and *rid*. For example, assuming a user types in a partial query "sig" on table dblp (Table 1), we first get the keyword range of "sig" ($[k_7, k_8]$) using the index on *prefix* and then find records r_3, r_6, and r_9 using the index on *kid*.

4 Fuzzy Search for Single Keyword

4.1 No-Index Methods

Recall the two no-index methods for exact search in Section 3.1. Since the LIKE predicate does not support fuzzy search, we cannot use the LIKE-based method. We can use

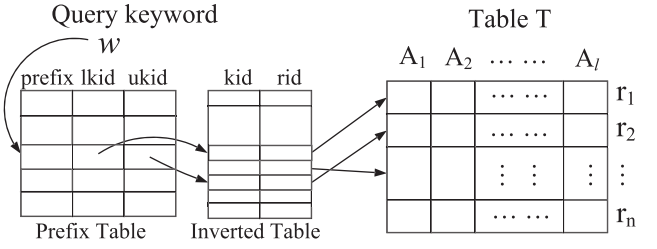


Fig. 1. Using inverted-index table and prefix table to support search-as-you-type.

UDFs to support fuzzy search. We use a UDF $\text{PED}(w, s)$ that takes a keyword w and a string s as two parameters, and returns the minimal edit distance between w and the prefixes of keywords in s. For instance, in Table 1,

$$\text{PED}(\text{"pvb"}, r_{10}[\text{title}]) = \text{PED}(\text{"pvb"}, \text{"privacy in database publishing "}) = 1$$

as r_{10} contains a prefix "pub" with edit distance of 1 to the query. We can improve the performance by doing early termination in the dynamic-programming computation [30] using an edit-distance threshold (if prefixes of two strings are not similar enough, then the two substrings cannot be similar), and devise a new UDF $\text{PEDTH}(w, s, \tau)$. If there is a keyword in string s having prefixes with an edit distance to w within τ, PEDTH returns true. In this way, we issue an SQL query that scans each record and calls UDF PEDTH to verify the record.

4.2 Index-Based Methods

This section proposes to use the inverted-index table and prefix table to support fuzzy search-as-you-type. Given a partial keyword w, we compute its answers in two steps. First we compute its similar prefixes from the prefix table P_T, and get the keyword ranges of these similar prefixes. Then we compute the answers based on these ranges using the inverted-index table I_T as discussed in Section 3.2. In this section, we focus on the first step: computing w's similar prefixes.

4.2.1 Using UDF

Given a keyword w, we can use a UDF to find its similar prefixes from the prefix table P_T. We issue an SQL query that scans each prefix 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 $\text{PEDTH}(w, P_T, \text{prefix}, \tau)$ 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:

"$\text{LENGTH}(P_T.\text{prefix}) \leq \text{LENGTH}(w) + \tau$AND
$\text{LENGTH}(P_T.\text{prefix}) \geq \text{LENGTH}(w) - \tau$".

4.2.2 Gram-Based Method

There are many q-gram-based methods to support approximate string search [17]. Given a string s, its q-grams are its

3. Here, we only use several keywords for ease of presentation.

Authorized licensed use limited to: St. Xavier's Catholic College of Engineering. Downloaded on October 22,2024 at 09:44:08 UTC from IEEE Xplore. Restrictions apply.

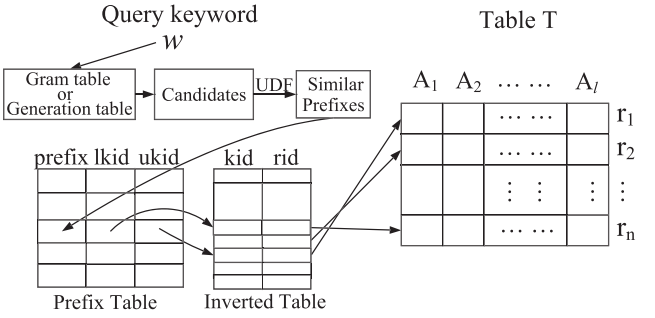


Fig. 2. Using the q-gram table and the neighborhood generation table to support fuzzy search.

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 $G^2(\texttt{pvldb}) = \{\texttt{pv}, \texttt{vl}, \texttt{ld}, \texttt{db}\}$ and $G^2(\texttt{vldb}) = \{\texttt{vl}, \texttt{ld}, \texttt{db}\}$. Strings s_1 and s_2 have an edit distance within threshold τ if

$$|G^q(s_1) \cap G^q(s_2)| \geq \max(|s_1|, |s_2|) + 1 - q - \tau * q \text{ [17]},$$

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 $\langle p, \texttt{qgram}\rangle$, where p is a prefix in the prefix table and qgram is a q-gram of p. Given a partial keyword w, we first find the prefixes in G_T with no smaller than $|w| + 1 - q - \tau * q$ grams in $G^q(w)$. We use the following SQL with "GROUP BY" command to get the candidates of ws similar prefixes:

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

As this method may involve false positives, we have to use UDFs to verify the candidates to get the similar prefixes of w. Fig. 2 illustrates how to use the gram-based method to answer a query. We can further improve the query performance by using additional filtering techniques, e.g., length filtering or position filtering [17].

It could be expensive to use "GROUP BY" in databases, and the q-gram-based method is inefficient, especially for large q-gram tables. Moreover, this method is rather inefficient for short query keywords [46], as short keywords have smaller numbers of q-grams and the method has low pruning power.

4.2.3 Neighborhood-Generation-Based Method

Ukkonen proposed a neighborhood-generation-based method to support approximate string search [45]. We extend this method to use SQL to support fuzzy search-as-you-type.

Given a keyword w, the substrings of w by deleting i characters are called "i-deletion neighborhoods" of w. Let $D_i(w)$ denote the set of i-deletion neighborhoods of w and $\hat{D}_\tau(w) = \cup_{i=0}^{\tau} D_i(w)$. For example, given a string "pvldb," $D_0(\texttt{pvldb}) = \{\texttt{pvldb}\}$, and $D_1(\texttt{pvldb}) = \{\texttt{vldb}, \texttt{pldb}, \texttt{pvdb}, \texttt{pvlb}, \texttt{pvld}\}$. Suppose $\tau = 1$, $\hat{D}_\tau(\texttt{pvldb}) = \{\texttt{pvldb}, \texttt{vldb}, \texttt{pldb}, \texttt{pvdb}, \texttt{pvlb}, \texttt{pvld}\}$. Moreover, there is a good property that given two strings s_1 and s_2, if $\text{ed}(s_1, s_2) \leq \tau$, $\hat{D}_\tau(s_1) \cap \hat{D}_\tau(s_2) \neq \phi$ as formalized in Lemma 1.

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
…	…	…

Lemma 1. *Given two strings* s_1, s_2, *if* $\text{ed}(s_1, s_2) \leq \tau$, $\hat{D}_\tau(s_1) \cap \hat{D}_\tau(s_2) \neq \phi$.

Proof. We can use deletion operations to replace the substitution and insertion operations as follows: suppose we can transform s_1 to s_2 with d deletions, i insertions, and r substitutions, such that $\text{ed}(s_1, s_2) = d + i + r \leq \tau$. We can transform s_1 and s_2 to the same string by doing $d + r$ deletions on s_1 and $i + r$ deletions on s_2, respectively. Thus, $\hat{D}_\tau(s_1) \cap \hat{D}_\tau(s_2) \neq \phi$. □

We use this property as a filter to find similar prefixes of the query keyword w. We can prune all the prefixes if they have no common i-deletion neighborhoods with w. To this end, for prefixes in the prefix table P_T, we create a deletion-based neighborhood-generation table D_T with records in the form $\langle p, i\text{-deletion}, i\rangle$, where p is a prefix in the prefix table P_T and i-deletion is an i-deletion neighborhood of p ($i \leq \tau$). For example, Table 3 gives a neighborhood-generation table.

Given a query keyword w, we first find the similar prefixes in D_T which have i-deletion neighborhoods in $\hat{D}_\tau(w)$. Then we use UDFs to verify the candidates to get similar prefixes. Formally, we use the following SQL to generate the candidates of w s similar prefixes:

SELECT DISTINCT prefix FROM D_T
WHERE $D_T.i$-deletion IN $\hat{D}_\tau(w)$.

Assuming a user types in a keyword "pvldb," we find 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 efficient for short strings. However, 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 $\mathcal{O}(\min(n^\tau, 2^n))$ neighborhoods. It needs large space to store these neighborhoods.

As the three methods have some limitations, we propose an incremental algorithm which uses previous computed results to answer subsequence queries in Section 4.3.

4.3 Incrementally Computing Similar Prefixes

The previous methods have the following limitations. First, they need to find similar prefixes of a keyword from scratch. Second, they may need to call UDFs many times. In this section, we propose a character-level incremental method to find similar prefixes of a keyword as a user types character by character. Chaudhuri and Kaushik [14] and Ji et al. [24]

4. We need to use multisets to accommodate duplicated grams.

Authorized licensed use limited to: St. Xavier's Catholic College of Engineering. Downloaded on October 22,2024 at 09:44:08 UTC from IEEE Xplore. Restrictions apply.

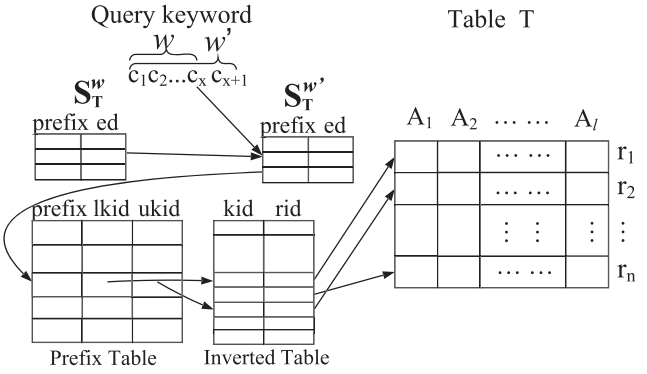


Fig. 3. Using character-level incremental method to support fuzzy search.

proposed to use a trie structure to incrementally compute similar prefixes. While adopting a similar incremental-search framework, we focus on the challenge of how to use SQL to do it. We develop effective index structures using auxiliary tables and devise pruning techniques to achieve a high speed. We develop novel techniques on how to use auxiliary tables, built-in indexes on key attributes, and pruning techniques. We also provide theoretical correctness.

4.3.1 Incremental-Computation Framework

Assume a user has typed in a keyword $w = c_1c_2 \cdots c_x$ character by character. For each prefix $p = c_1c_2 \cdots c_i (i \leq x)$, we maintain a similar-prefix table S_T^p with records in the form $\langle \texttt{prefix}, \mathrm{ed}(p, \texttt{prefix}) \rangle$, which keeps all the prefixes similar to p and their corresponding edit distances. As the similar-prefix tables are small (usually within 1,000 records), we can use in-memory tables to store them. The similar-prefix table is shared by different queries. If the table gets too big, we can periodically remove some of its entries. In other words, the incremental-computation algorithm does not need to maintain session information for different queries.

Suppose the user types one more character c_{x+1} and submits a new query $w' = c_1c_2 \cdots c_xc_{x+1}$. We use table S_T^w to compute $S_T^{w'}$ (Section 4.3.2), find the keyword ranges of similar prefixes in $S_T^{w'}$ by joining the similar-prefix table $S_T^{w'}$ and the prefix table P_T, and compute the answer of w' using the inverted-index table I_T (Fig. 3).

Based on similar-prefix table $S_T^{w'}$ of keyword w', we use the following SQL to answer the single-keyword query w':

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

TABLE 4
Similar-Prefix Table $S_{\mathrm{dblp}}^{\mathrm{vldb}}(\tau = 1)$

$prefix$	Edit Distance
vldb	0
vld	1
pvldb	1
vldbj	1
...	...

We can create indexes on the attribute prefix of the prefix table P_T and the similar-prefix table $S_T^{w'}$, and in this way the SQL can be executed efficiently.

For example, suppose $\tau = 1$. Assume a user has typed in a keyword "vld" and its similar-prefix table $S_{\mathrm{dblp}}^{\mathrm{vld}} = \{\langle \texttt{vl}, 1\rangle; \langle \texttt{vld}, 0\rangle; \langle \texttt{vldb}, 1\rangle; \langle \texttt{pvld}, 1\rangle\}$ has been computed and cached. Suppose the user types in one more character "b." We first generate the similar-prefix table $S_{\mathrm{dblp}}^{\mathrm{vldb}} = \{\langle \texttt{vldb}, 0\rangle; \langle \texttt{vld}, 1\rangle; \langle \texttt{pvldb}, 1\rangle; \langle \texttt{vldbj}, 1\rangle\}$ based on $S_{\mathrm{dblp}}^{\mathrm{vld}}$ (Section 4.3.2), as shown in Table 4, and then issue the following SQL to answer the query "vldb":

SELECT dblp.* FROM $S_{\mathrm{dblp}}^{\mathrm{vldb}}, P_{\mathrm{dblp}}, I_{\mathrm{dblp}}$, dblp
WHERE $S_{\mathrm{dblp}}^{\mathrm{vldb}}$.prefix $= P_{\mathrm{dblp}}$.prefix AND
$P_{\mathrm{dblp}}.ukid \geq I_{\mathrm{dblp}}.kid$ AND $P_{\mathrm{dblp}}.lkid \leq I_{\mathrm{dblp}}.kid$ AND
$I_{\mathrm{dblp}}.rid =$ dblp.rid,

which returns records r_1, r_4, and r_8.

4.3.2 Incremental Computation Using SQL

In this section, we discuss how to incrementally compute the similar-prefix table S_T^w.

Initialization. For the empty string ϕ, its similar-prefix table S_T^ϕ has all the prefixes with a length within the edit-distance threshold τ, and we can compute such prefixes and their corresponding edit distances using the following SQL:

SELECT prefix, LENGTH(prefix) AS ed FROM P_T
WHERE LENGTH(prefix) $\leq \tau$.

For example, consider the prefix table with prefixes {"ϕ, p, pv, pvl, pvld, pvldb, v, vl, vld, vldb"}. Suppose $\tau = 1$. This SQL returns $S_T^\phi = \{\langle \phi, 0\rangle, \langle \texttt{p}, 1\rangle, \langle \texttt{v}, 1\rangle\}$, as shown in Table 5a. We can precompute and materialize the similar-prefix table of the empty string.

Then, we discuss how to use S_T^w to compute $S_T^{w'}$. We have an observation that only those prefixes having a prefix in S_T^w could be similar-prefixes of w' based on Lemma 2.

Lemma 2. *If $\langle v', e'\rangle \in S_T^{w'}$, then $\exists \langle v, e\rangle \in S_T^w$ such that v is a prefix of v' and $e \leq e'$.*

TABLE 5
Similar-Prefix Tables of a Prefix Table with Strings in {"ϕ, v, vl, vld, vldb, p, pv, pvl, pvld, pvldb"}

(a) S_T^ϕ

prefix	ed	from	op
ϕ	0	ϕ	m
p	1	ϕ	i
v	1	ϕ	i

(b) S_T^{v}

prefix	ed	from	op
ϕ	1	ϕ	d
v	0	ϕ	m
pv	1	p	m
vl	1	ϕ	i
p	1	ϕ	s

(c) S_T^{vl}

prefix	ed	from	op
v	1	v	d
vl	0	v	m
pvl	1	pv	m
vld	1	v	i

(d) S_T^{vld}

prefix	ed	from	op
vl	1	v	d
vld	0	vl	m
pvld	1	pvl	m
vldb	1	vl	i

($\tau = 1$. *For ease of presentation, we add two columns "from" and "op," where column "from" denotes where the record is derived from, and column "op" denotes operations—m:match, d:deletion, i:insertion, s:substitution.)*

Authorized licensed use limited to: St. Xavier's Catholic College of Engineering. Downloaded on October 22,2024 at 09:44:08 UTC from IEEE Xplore. Restrictions apply.

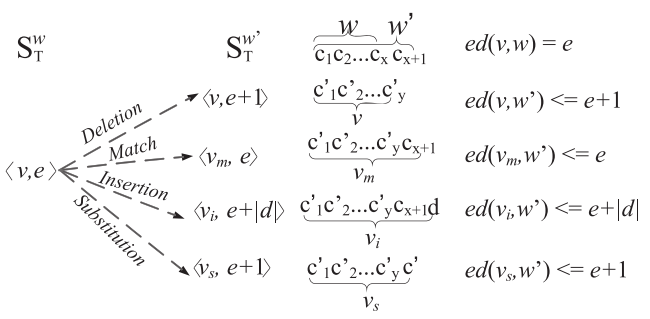


Fig. 4. Incrementally computing similar prefixes. (d is a string with length $|d|$ no larger than $\tau - e$.)

Proof. Consider a transformation from node v' to keyword w' with $\mathrm{ed}(v', w')$ operations. In the transformation, we consider the last match case between two characters in v' and w'. Let p_v and p_w be, respectively, the prefixes of v' and w' before the last match characters (including the two characters).

If $p_v = p_w = \phi$,[5] we have $e' = \mathrm{ed}(v', w') = \max(|v'|, |w'|)$. Let v and w, respectively, denote the prefixes of v' and w' without the last characters. We have $e = \mathrm{ed}(v, w) = \max(|v'|, |w'|) - 1$. Thus, v is a prefix of v', $e \leq e'$, and $\langle v, e\rangle$ must be in S_T^w. Otherwise $p_v = p_w \neq \phi$. Let v and w, respectively, denote the prefixes of p_v and p_w without the last characters. We have $e = \mathrm{ed}(v, w) = \mathrm{ed}(v', w') = e'$. Thus, v is a prefix of v', $e \leq e'$, and $\langle v, e\rangle$ must be in S_T^w. □

Based on this property, for a new prefix query $w' = wc_{x+1}$ by concatenating query w and a new character c_{x+1}, we construct $S_T^{w'}$ by checking each record in S_T^w as follows: for $\langle v, e\rangle \in S_T^w(\mathrm{ed}(v, w) = e)$, we consider the following basic edit operations (as illustrated in Fig. 4).

Deletion. We can transform $w' = wc_{x+1}$ to v by first transforming w to v (with e operations) and then deleting c_{x+1} from w'. The transformation distance (the number of operations in the transformation) is $e + 1$.[6] If $e + 1 \leq \tau$, v is similar to w'. We get such prefixes using the following SQL:

SELECT prefix, $\mathrm{ed} + 1$ as ed FROM S_T^w WHERE $\mathrm{ed} < \tau$.

For example, suppose $w' =$ "v", $w = \phi$, and S_T^ϕ is the one shown in Table 5a. Since only $\langle \phi, 0\rangle \in S_T^\phi$ satisfies the SQL condition, this SQL returns $\langle \phi, 1\rangle$. Apparently ϕ is similar to the new query "v" with an edit distance 1.

Match. Let v_m be the concatenated string of string v and character c_{x+1}, i.e., $v_m = vc_{x+1}$. We can transform w' to v_m by first transforming w to v (with e operations) and then matching the last character. Thus, the transformation distance is e. As $e \leq \tau$, v is similar to w', and we can get all such similar prefixes using the following SQL:

SELECT P_T.prefix, ed FROM S_T^w, P_T
WHERE P_T.prefix = CONCAT(S_T^w.prefix, c_{x+1}),

where CONCAT(s, t) concatenates two strings s and t. In the SQL, S_T^w.prefix corresponds to v and P_T.prefix corresponds to v_m. In our example, as $\langle \phi, 0\rangle \in S_T^\phi$ and $\langle \mathrm{p}, 1\rangle \in S_T^\phi$ satisfy the SQL condition, this SQL returns $\langle \mathrm{v}, 0\rangle$ and $\langle \mathrm{pv}, 1\rangle$.

5. The operation between the empty string and the root node is also a match case.

6. For each similar string of w', we can compute its real edit distance to w' by keeping the smallest one as discussed later.

Insertion. If v_m is in P_T, the strings with v_m as a prefix could also be similar to w'. For each such string v_i, we can transform w' to v_i by first transforming w' to v_m (with e operations) and then inserting $|v_i| - |v_m| = |v_i| - |v| - 1$ characters after v_m. Thus, the transformation distance is $e + |v_i| - |v| - 1$. If $e + |v_i| - |v| - 1 \leq \tau$, v_i is similar to w'. We can get such similar prefixes using the following SQL:

SELECT P_T.prefix, LENGTH(P_T.prefix)-
LENGTH(S_T^w.prefix)-1+ed AS ed FROM S_T^w, P_T
WHERE LENGTH(P_T.prefix)> LENGTH
(S_T^w.prefix)+1 AND
ed + LENGTH (P_T.prefix)-LENGTH(S_T^w.prefix)-1
$\leq \tau$ AND
SUBSTR(P_T.prefix, 1, LENGTH (S_T^w.prefix) + 1) =
CONCAT (S_T^w.prefix,c_{x+1}),

where SUBSTR(str, pos, len) returns a substring with "len" characters from the string str, starting at position pos. In the SQL statement, S_T^w.prefix, ed, and P_T.prefix, respectively, correspond to v, e, and v_i. In our running example, as only $\langle \phi, 0\rangle \in S_T^\phi$ satisfies the SQL condition, this SQL returns $\langle \mathtt{v1}, 1\rangle$. Note that "v" matches the query keyword and we can do an insertion "1" after "v," thus "v1" is similar to the query "v" with an edit distance of 1.

Substitution. Let v_s be the concatenated string of v and character c' where $c' \neq c_{x+1}$, i.e., $v_s = vc'$. We can transform w' to v_s by first transforming w to v (with e operations) and then substituting c_{x+1} for c'. Thus, the transformation distance is $e + 1$. If $e + 1 \leq \tau$, v_s is similar to w', and we can get all such prefixes using the following SQL:

SELECT P_T.prefix, ed+1 AS ed FROM S_T^w, P_T
WHERE $\mathrm{ed} < \tau$
AND SUBSTR(P_T.prefix, 1, LENGTH(P_T.prefix)-1) =
S_T^w.prefix AND P_T.prefix != CONCAT(S_T^w.prefix, c_{x+1}).

In the SQL, P_T.prefix and S_T^w.prefix, respectively, correspond to v_s and v. In our running example, as $\langle \phi, 0\rangle \in S_T^\phi$, this SQL returns $\langle \mathrm{p}, 1\rangle$ since we can use "p" to replace "v."

We insert the results of these SQL queries into the similar-prefix table $S_T^{w'}$. During the insertion, it is possible to add multiple pairs $\langle v', e'_1\rangle$ and $\langle v', e'_2\rangle$ for the same similar prefix v'. In this case, only the one with the smallest edit operation should be inserted in $S_T^{w'}$. The reason is that we only keep the minimum number of edit operations to transform the string v' to the string w'. To this end, we can first insert all strings generated by the above SQL queries, and then use the following SQL to prune the nodes with larger edit distances by doing a postprocessing:

DELETE FROM $S_T^{w'}$ AS S_1, $S_T^{w'}$ AS S_2
WHERE S_1.prefix = S_2.prefix AND S_1.ed > S_2.ed.

Theorem 1 shows the correctness of our method.

Theorem 1. *For a query string $w = c_1c_2 \cdots c_x$, let S_T^w be its similar-prefix table. Consider a new query string $w' = c_1c_2 \cdots c_xc_{x+1}$. $S_T^{w'}$ is generated by the above SQL queries based on S_T^w. 1) Completeness: Every similar prefix of the new query string w' will be in $S_T^{w'}$. 2) Soundness: Each string in $S_T^{w'}$ is a similar prefix of the new query string w'.*

Proof. 1) We first prove the completeness. We prove it by induction. This claim is obviously true when $w = w_0 = \epsilon$.

Authorized licensed use limited to: St. Xavier's Catholic College of Engineering. Downloaded on October 22,2024 at 09:44:08 UTC from IEEE Xplore. Restrictions apply.

Suppose the claim is true for $w_x = w$ with x characters. We want to prove this claim is also true for a new query string w_{x+1}, where $w_{x+1} = w' = w_x c_{x+1}$.

Suppose v is a similar prefix of w_{x+1}. If $v = \epsilon$, then by definition $\mathrm{ed}(v, w_{x+1}) = \mathrm{ed}(\epsilon, w_{x+1}) = x + 1 \leq \tau$, and $x \leq \tau - 1 < \tau$. Thus, $\mathrm{ed}(v, w_x) = \mathrm{ed}(\epsilon, w_x) = x \leq \tau$, and v is also a similar prefix of w_x. When we consider this node v, we add the pair $\langle v, x + 1\rangle$ (i.e., $\langle v, \mathrm{ed}(v, w_{x+1})\rangle$) into $S_T^{w_{x+1}}$.

Now consider the case where the similar prefix v of w_{x+1} is not the empty string. Let $v = n_{y+1} = n_y d$, i.e., it has $y + 1$ characters, and is concatenated from a string n_y and a character d. By definition, $\mathrm{ed}(n_{y+1}, w_{x+1}) \leq \tau$. We want to prove that $\langle n_{y+1}, \mathrm{ed}(n_{y+1}, w_{x+1})\rangle$ will be added to $S_T^{w_{x+1}}$.

Based on the idea in the classic dynamic-programming algorithm, we consider the following four cases in the minimum number of edit operations to transform n_{y+1} to w_{x+1}.

Case 1: Deleting the last character c_{x+1} from w_{x+1}, and transforming n_{y+1} to w_x. Since $\mathrm{ed}(n_{y+1}, w_{x+1}) = \mathrm{ed}(n_{y+1}, w_x) + 1 \leq \tau$, we have $\mathrm{ed}(n_{y+1}, w_x) \leq \tau - 1 < \tau$. Thus, n_{y+1} is a similar prefix of w_x. Based on the induction assumption, $\langle n_{y+1}, \mathrm{ed}(n_{y+1}, w_x)\rangle$ must be in $S_T^{w_x}$. From the node n_{y+1}, our method considers the deletion case when it considers the node n_y, and adds $\langle n_{y+1}, \mathrm{ed}(n_{y+1}, w_x) + 1\rangle$ to $S_T^{w_{x+1}}$, which is exactly $\langle n_{y+1}, \mathrm{ed}(n_{y+1}, w_{x+1})\rangle$.

Case 2: Substituting the character d of n_{y+1} for the last character c_{x+1} of w_{x+1}. Since $\mathrm{ed}(n_{y+1}, w_{x+1}) = \mathrm{ed}(n_y, w_x) + 1 \leq \tau$, we have $\mathrm{ed}(n_y, w_x) \leq \tau - 1 < \tau$. Thus, n_y is a similar prefix of w_x. Based on the induction assumption, $\langle n_y, \mathrm{ed}(n_y, w_x)\rangle$ must be in $S_T^{w_x}$. From node n_y, our method considers the substitution case when it considers this child node (n_{y+1}) of the node n_y, and adds $\langle n_{y+1}, \mathrm{ed}(n_y, w_x) + 1\rangle$ to $S_T^{w_{x+1}}$, which is exactly $\langle n_{y+1}, \mathrm{ed}(n_{y+1}, w_{x+1})\rangle$.

Case 3: The last character c_{x+1} of w_{x+1} matching the character d of n_{y+1}. Since $\mathrm{ed}(n_{y+1}, w_{x+1}) = \mathrm{ed}(n_y, w_x) \leq \tau$, then n_y is a similar prefix of n_x. Based on the induction assumption, $\langle n_y, \mathrm{ed}(n_y, w_x)\rangle$ must be in $S_T^{w_x}$. From node n_y, our method considers the match case when it considers this child node (n_{y+1}) of the node n_y, and adds $\langle n_{y+1}, \mathrm{ed}(n_y, w_x)\rangle$ to $S_T^{w_{x+1}}$, which is exactly $\langle n_{y+1}, \mathrm{ed}(n_{y+1}, w_{x+1})\rangle$.

Case 4: Transforming n_y to w_{x+1} and inserting character d of n_{y+1}. For each transformation from n_y to w_{x+1}, we consider the last character c_{x+1} of w_{x+1}. First, we can show that this transformation cannot delete the character c_{x+1}, since otherwise we can combine this deletion of c_{x+1} and the insertion of d into one substitution, yielding another transformation with a smaller number of edit operations, contradicting to the minimality of edit distance. Thus, we can just consider two possible operations on the character c_{x+1} in this transformation. 1) Matching c_{x+1} for the character of an ancestor n_a of n_{y+1}: in this case, since $\mathrm{ed}(n_{y+1}, w_{x+1}) = \mathrm{ed}(n_{a-1}, w_x) + y - a + 1 \leq \tau$, we have $\mathrm{ed}(n_{a-1}, w_x) \leq \tau$, and n_{a-1} is a similar prefix of w_x. Based on the induction assumption, $\langle n_{a-1}, \mathrm{ed}(n_{a-1}, w_x)\rangle$ must be in $S_T^{w_x}$. From node n_{a-1}, the algorithm considers the matching case, and adds $\langle n_{y+1}, \mathrm{ed}(n_{a-1}, w_x) + y - a + 1\rangle$ to $S_T^{w_{x+1}}$, which is $\langle n_{y+1}, \mathrm{ed}(n_{y+1}, w_{x+1})\rangle$. 2) Substituting c_{x+1} for the character of an ancestor n_a of n_{y+1}: in this case, instead of substituting c for the character of n_a and inserting the character d, we can insert the character of n_a and substitute c_{x+1} for the last character d. Then we get another transformation with the same number of edit operations. (Characters c and d cannot be the same, since otherwise the new transformation could have fewer edit operations, contradicting to the minimality of edit distance.) We use the same argument in "Case 2" to show that our method adds $\langle n_{y+1}, \mathrm{ed}(n_{y+1}, w_{x+1})\rangle$ to $S_T^{w_{x+1}}$.

In summary, for all cases the algorithm adds $\langle n_{y+1}, \mathrm{ed}(n_{y+1}, w_{x+1})\rangle$ to $S_T^{w_{x+1}}$.

2) Then we prove the soundness. By definition, a transformation distance of two strings in each added tuple by the algorithm is no less than their edit distance. That is, $\mathrm{ed}(n, w') \leq \tau$. Thus, n must be a similar prefix of p. □

4.3.3 Improving Performance Using Indexes

As we can create indexes on the attribute prefix of the prefix table P_T and the similar-prefix table S_T^w, the SQL statements for the deletion and match cases can be efficiently executed using the indexes. However, for the SQL of the substitution case, the SQL contains a statement $\mathrm{SUBSTR}(P_T.\mathrm{prefix}, 1, \mathrm{LENGTH}(P_T.\mathrm{prefix}) - 1) = S_T^w.\mathrm{prefix}$. Although we can create an index on the attribute prefix of the similar prefix table S_T^w, if there is no index to support the predicate $\mathrm{SUBSTR}(P_T.\mathrm{prefix}, 1, \mathrm{LENGTH}\ (P_T.\mathrm{prefix}) - 1)$, it is rather expensive to execute the SQL. To improve the performance, we can alter table P_T by adding an attribute $\mathtt{parent} = \mathrm{SUBSTR}(P_T.\mathrm{prefix}, 1, \mathrm{LENGTH}(P_T.\mathrm{prefix}) - 1)$, and create a table $P_T\langle \mathtt{prefix}, lkid, ukid, \mathtt{parent}\rangle$. Using this table, we propose an alternative method and use the following the SQL for the substitution case:

SELECT P_T.prefix, ed+1 AS ed FROM S_T^w, P_T
WHERE $P_T.\mathtt{parent} = S_T^w$.prefix AND $\mathrm{ed} < \tau$ AND
P_T.prefix! = CONCAT $(S_T^w.\mathrm{prefix}, c_{x+1})$.

We can create an index on attribute parent of prefix table P_T to increase search performance.

Similarly, it is inefficient to execute the SQL for the insertion case as it contains a complicated statement

$$\mathrm{SUBSTR}\big(P_T.\mathrm{prefix}, 1, \mathrm{LENGTH}(S_T^w.\mathrm{prefix}) + 1\big) = \mathrm{CONCAT}(S_T^w.\mathrm{prefix}, c_{x+1}).$$

Next we discuss how to improve the SQL using indexes. Let Y_0 be the similar-prefix table of the results of the SQL for the match case. For insertions, we need to find the similar strings with prefixes in Y_0. Let Y_{i+1} be the similar-prefix table composed of prefixes by appending one more character to those prefixes in Y_i for $0 \leq i \leq \tau - 1$. Obviously $\cup_{i=1}^{\tau} Y_i$ is exactly the result of the SQL for the insertion case. Note that Y_0 can be efficiently computed as we can use the SQL for the match case to generate it. Iteratively, we compute Y_{i+1} based on Y_i using the following SQL:

SELECT P_T.prefix, ed+1 AS ed FROM Y_i, P_T
WHERE $P_T.\mathtt{parent} = Y_i$.prefix AND $\mathrm{ed} < \tau$.

We can create indexes on the parent attribute of the prefix table P_T and the prefix attribute of Y_i to improve the performance. In our running example, $Y_0 = \{\langle \mathtt{v}, 0\rangle; \langle \mathtt{pv}, 1\rangle\}$. As only $\langle \mathtt{v}, 0\rangle \in Y_0$ satisfies the SQL condition, this SQL returns $\langle \mathtt{vl}, 1\rangle$. Thus, we can use several SQL statements that can be efficiently executed to replace the original complicated SQL statement for the insertion case.

Authorized licensed use limited to: St. Xavier's Catholic College of Engineering. Downloaded on October 22,2024 at 09:44:08 UTC from IEEE Xplore. Restrictions apply.

5 Supporting Multikeyword Queries

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

5.1 Computing Answers from Scratch

Given a multikeyword query Q with m keywords $w_1, w_2, \ldots, w_m$, there are two ways to answer it from scratch. 1) *Using the* "INTERSECT" *Operator:* a straightforward way is to first compute the records for each keyword using the previous methods, and then use the "INTERSECT" operator to join these records for different keywords to compute the answers. 2) *Using Full-text Indexes:* we first use full-text indexes (e.g., CONTAINS command) to find records matching the first $m-1$ complete keywords, and then use our methods to find records matching the last prefix keyword. Finally, we join the results. These two methods cannot use the precomputed results and may lead to low performance. To address this problem, we propose an incremental-computation method.

5.2 Word-Level Incremental Computation

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

Exact search. 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
$P_T.ukid \geq I_T.kid$ AND $P_T.lkid \leq I_T.kid$ AND
$I_T.rid = C_Q.rid$ AND $C_Q.rid = T.rid$.

For example, suppose a user has typed in a query $Q =$ "privacysigmod" and we have created a temporary table $C_Q = \{r_3, r_6\}$. Then the user types in a new keyword "pub" and submits a new query $Q' =$ "privacysigmodpub." We check whether records r_3 and r_6 contain a keyword with the prefix "pub." Using C_Q, we find that only r_6 contains a keyword "publishing" with the prefix "pub."

Fuzzy search. As an example, we consider the character-level incremental method. We first compute $S_T^{w_{m+1}}$ using the character-level incremental method for the new keyword w_{m+1}, and then use $S_T^{w_{m+1}}$ to answer the query. Based on the temporary table C_Q, we use the following SQL query to answer Q':

SELECT $T.^*$ FROM $S_T^{w_{m+1}}, P_T, I_T, C_Q, T$
WHERE $S_T^{w_{m+1}}$.prefix $= P_T$.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 $C_Q.rid = T.rid$.

If the user modifies the keyword w_m of query Q to w'_m and submits a query with keywords $w_1, w_2, \ldots, w_{m-1}, w'_m$, we can use the cached result of query $w_1, w_2, \ldots, w_{m-1}$ 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.

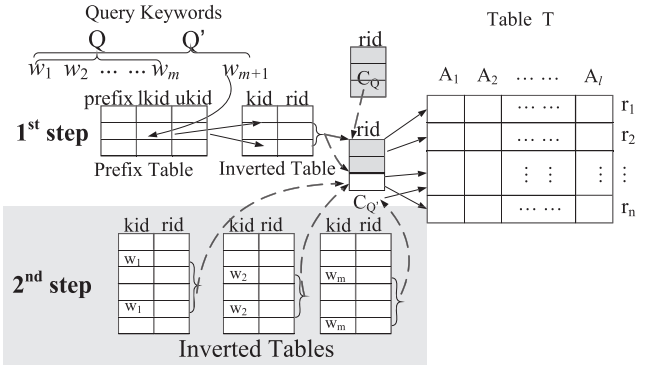


Fig. 5. Incrementally computing first-N answers.

6 Supporting First-N Queries

The previous methods focus on computing all the answers. As a user types in a query character by character, we usually give the user the first-N (any-N) results as the instant feedback. This section discusses how to compute the first-N results.

Exact first-N queries. For exact search, we can use the "LIMIT N" syntax in databases to return the first-N results. For example, MYSQL uses "LIMIT n_1, n_2" to return n_2 rows starting from the n_1th row. As an example, we focus on how to extend the method based on the inverted-index table and the prefix table (Section 3.2). Our techniques can be easily extended to other methods.

For a single-keyword query, we can use "LIMIT $0, N$" to find the first-N answers. For example, assume a user types in a keyword query "sig." To compute the first-2 answers, we issue the following SQL:

SELECT dblp.* FROM $P_{\text{dblp}}, I_{\text{dblp}}$, dblp
WHERE P_{dblp}.prefix $=$ "sig" AND
$P_{\text{dblp}}.ukid \geq I_{\text{dblp}}.kid$ AND $P_{\text{dblp}}.lkid \leq I_{\text{dblp}}.kid$ AND
$I_{\text{dblp}}.rid =$ dblp.rid
LIMIT $0, 2$,

which returns records r_3 and r_6.

For multikeyword queries, if we use the "INTERSECT" operator, we can use the "LIMIT" operator to find the first-N answers. But it is not straightforward to extend the word-level incremental method to support first-N queries, since the cached results of a query Q with keywords $w_1, w_2, \ldots, w_m$ have N records, instead of all the answers. For a query Q' with one more keyword w_{m+1}, we may not get N answers for Q' using the cached results C_Q, and need to continue to access records from the inverted-index table. To address this issue, we first use the incremental algorithms as discussed in Section 5 to answer Q' using C_Q. Let $R(C_Q, Q')$ denote the results. If the temporary table C_Q has smaller than N records or $R(C_Q, Q')$ has N records, $R(C_Q, Q')$ is exactly the answers. Otherwise, we continue to access records from the inverted-index table. Fig. 5 shows how to incrementally find first-N results.

We progressively access the records that contain the first keyword w_1. Suppose we have accessed the records of w_1

Authorized licensed use limited to: St. Xavier's Catholic College of Engineering. Downloaded on October 22,2024 at 09:44:08 UTC from IEEE Xplore. Restrictions apply.

from the 0th row to the $(\eta-1)$st row for answering query Q. Then for query Q', we continue to access the records of w_1 from the ηth row. To get enough answers, we access θ records for each SQL query, where θ is a parameter depending on the keyword distribution (usually set to $m * N$).

Next we discuss how to assign η iteratively. Initially, for $m = 1$, that is, the query Q has only one keyword and Q' has two keywords. When answering Q, we have visited N records for w_1, and we need to continue to access θ records starting from the Nth record. Thus, we set $\eta = N$. If we cannot get N answers, we set $\eta = \eta + \theta$ until we get N results (or we have accessed all of the records).

For example, assume a user types in a query "privacyic" character by character, and $N = 2$. When the user types in keyword "privacy," we issue the following SQL:

SELECT dblp.* FROM $P_{\text{dblp}}, I_{\text{dblp}}$, dblp
WHERE P_{dblp}.prefix = "privacy" AND
$P_{\text{dblp}}.ukid \geq I_{\text{dblp}}.kid$ AND $P_{\text{dblp}}.lkid \leq I_{\text{dblp}}.kid$ AND
$I_{\text{dblp}}.rid =$ dblp.rid
LIMIT $0, 2$,

which returns records r_1 and r_2. We compute and cache $C_Q = \{r_1, r_2\}$. When the user types in another keyword "ic" and submits a query "privacyic," we first use C_Q to answer the query and get record r_2. As we want to compute first-2 results, we need to issue the following SQL:

SELECT dblp.* FROM $P_{\text{dblp}}, I_{\text{dblp}}$, dblp
WHERE P_{dblp}.prefix = "privacy" AND
$P_{\text{dblp}}.ukid \geq I_{\text{dblp}}.kid$ AND $P_{\text{dblp}}.lkid \leq I_{\text{dblp}}.kid$ AND
$I_{\text{dblp}}.rid =$ dblp.rid
LIMIT $2, 2*2$
INTERSECT
SELECT dblp.* FROM $P_{\text{dblp}}, I_{\text{dblp}}$, dblp
WHERE P_{dblp}.prefix = "ic" AND
$P_{\text{dblp}}.ukid \geq I_{\text{dblp}}.kid$ AND $P_{\text{dblp}}.lkid \leq I_{\text{dblp}}.kid$ AND
$I_{\text{dblp}}.rid =$ dblp.rid,

which returns records r_5. Thus, we get first-2 results (records r_2 and r_5) and terminate the execution.

Fuzzy first-N queries. The above methods cannot be easily extended to support fuzzy search, as they cannot distinguish the results of exact search and fuzzy search. Generally, we need to first return the "best results" with smaller edit distances. To address this issue, we propose to progressively compute the results. As an example, we consider the character-level incremental method (Section 4.3).

For a single-keyword query w, we first get the results with edit distance 0. If we have gotten N answers, we terminate the execution; otherwise, we progressively increase the edit-distance threshold and select the records with edit-distance thresholds $1, 2, \ldots, \tau$, until we get N answers.

For example, suppose $\tau = 2$. Considering a keyword "vld," to get the first-3 answers, we first issue the following SQL:

SELECT dblp.* FROM $S^{\text{vld}}_{\text{dblp}}, P_{\text{dblp}}, I_{\text{dblp}}$, dblp
WHERE $S^{\text{vld}}_{\text{dblp}}$.ed $= 0$ AND $S^{\text{vld}}_{\text{dblp}}$.prefix $= P_{\text{dblp}}$.prefix
AND $P_{\text{dblp}}.ukid \geq I_{\text{dblp}}.kid$ AND $P_{\text{dblp}}.lkid \leq I_{\text{dblp}}.kid$
AND $I_{\text{dblp}}.rid =$ dblp.rid
LIMIT $0, 3$,

TABLE 6
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 ($\lvert q\rvert = 2$)	902 MB	329 MB
Avg. size of the similar-prefix table	3 KB	2 KB

which returns records r_4 and r_8. As we only get two results, we increase the edit-distance threshold to 1, and issue a new SQL, which returns record r_1. Thus, we get first-3 results and terminate the execution. We do not need to consider the case that $\tau = 2$ as we have gotten the first-3 results.

7 Supporting Updates Efficiently

We can use a trigger to support data updates. We consider insertions and deletions of records.

Insertion. Assume a record is inserted. We first assign it a new record ID. For each keyword in the record, we insert the keyword into the inverted-index table. For each prefix of the keyword, if the prefix is not in the prefix table, we add an entry for the prefix. For the keyword-range encoding of each prefix, we can reserve extra space for prefix ids to accommodate future insertions. We only need to do global reordering if a reserved space of the insertion is consumed.

Deletion. Assume a record is deleted. For each keyword in the record, in the inverted-index table we use a bit to denote whether a record is deleted. Here we use the bit to mark the record to be deleted. We do not update the table until we need to rebuild the index. For the range encoding of each prefix, we can use the deleted prefix ids for future insertions.

The ranges of ids are assigned-based inverse document frequency (idf) of keywords. We use a larger range for a keyword with a smaller idf. In most cases, we can use the kept extra space for update. But in the worst case, we need to rebuild the index. The problem of the range selection and analysis is beyond the scope of this paper.

8 Experimental Study

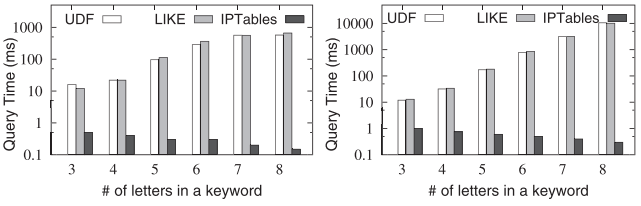
We implemented the proposed methods on two real data sets. 1) "DBLP": It included 1.2 million computer science publications.[7] 2) "MEDLINE": It included 5 million biomedical articles.[8] Table 6 summarizes the data sets and index sizes. We see that the size of inverted-index table and prefix table is acceptable, compared with the data set size. As a keyword may have many deletion-based neighbors, the size of prefix-deletion table is rather large. The size of q-gram table is also larger than that of our method, since a substring

7. http://dblp.uni-trier.de/xml/.
8. http://www.ncbi.nlm.nih.gov/pubmed/.

Authorized licensed use limited to: St. Xavier's Catholic College of Engineering. Downloaded on October 22,2024 at 09:44:08 UTC from IEEE Xplore. Restrictions apply.

TABLE 7
Ten Example Keyword Queries

Q_1	similarity	Q_6	database
Q_2	similarity join	Q_7	database search
Q_3	similarity search	Q_8	database keyword search
Q_4	similarity size	Q_9	xml keyword
Q_5	entity extraction	Q_{10}	xml keywore search



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

Fig. 6. Exact-search performance for answering single-keyword queries (varying keyword length).

has multiple overlapped q-grams. Note that the size of similar-prefix table is very small as it only stores similar prefixes of a keyword.

We used 1,000 real queries for each data set from the logs of our deployed systems. We assumed the characters of a query were typed in one by one. Table 7 gives ten example queries.

We used a Windows 7 machine with an Intel Core 2 Quad processor (X5450 3.00 GHz and 4 GB memory). We used three data bases, MYSQL, SQL Server 2005, and Oracle 11g. By default, we used MYSQL in the experiments. We will compare different data bases in Section 8.3.

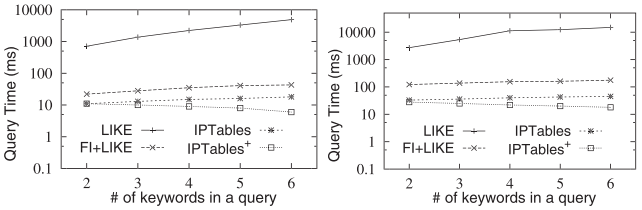
8.1 Exact Search

Single-keyword queries. We implemented three methods for single-keyword queries: 1) using UDF; 2) using the LIKE predicate; and 3) using the inverted-index table and the prefix table (called "IPTables"). We compared the performance of the three methods to compute the first-N answers. Unless otherwise specified, $N = 10$. Fig. 6 shows the results.

We see that both the UDF-based method and the LIKE-based method had a low search performance as they needed to scan records. IPTables achieved a high performance by using indexes. As the keyword length increased, the performance of the first two methods decreased, since the keyword became more selective, and the two methods needed to scan more records in order to find the same number (N) of answers. As the keyword length increased, IPTables had a higher performance, since there were fewer complete keywords for the query and the query needed fewer join operations.

Multikeyword queries. We implemented six methods for multikeyword 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 inverted-index table and prefix table (IPTables);
6. using the word-level incremental method (called "IPTables$^+$")



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

Fig. 7. Exact-search performance of answering multikeyword queries (varying keyword numbers).

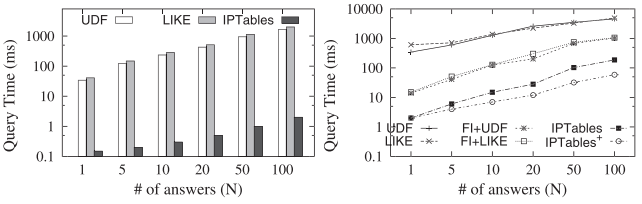


Fig. 8. Exact-search performance of computing first-N answers by varying different N values. (a) Single keywords—DBLP. (b) Multi-keywords—MEDLINE.

Fig. 7 shows the results. We only show the results of four methods as the UDF-based method and the LIKE-based method achieved similar results and the two methods using full-text indexes got similar results.

We see that the LIKE-based method had the worst performance. The method using the full-text indexes achieved a better performance. For example, on the MEDLINE data set, the LIKE-based method took 5,000 ms to answer a query, and the latter method reduced the time to 150 ms. IPTables$^+$ achieved the highest performance. It could answer a query within 10 ms for the DBLP data set and 30 ms for the MEDLINE data set, as it used an incremental method to find first-N answers and did not scan all records.

Varying the number of answers N. We compared the performance of the methods to compute first-N answers by varying the number of first results. Fig. 8 shows the experimental results. We can see that IPTables achieved the highest performance for single-keyword queries and IPTables$^+$ outperformed other methods for multiple-keyword queries, for different N values. For example, IPTables computed 100 answers for single-keyword queries within 2 ms on the DBLP data set and IPTables$^+$ computed 100 answers for multikeyword queries within 60 ms on the MEDLINE data set. This difference shows the advantages of our index structures and our incremental algorithms.

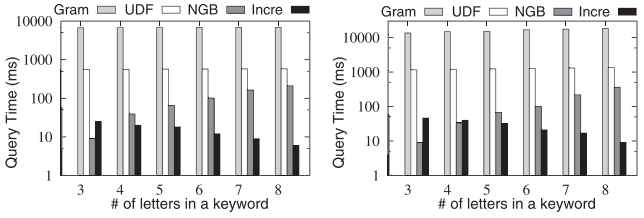
8.2 Fuzzy Search

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 UDF;
2. using the gram-based method (called "Gram") described in [30][9];

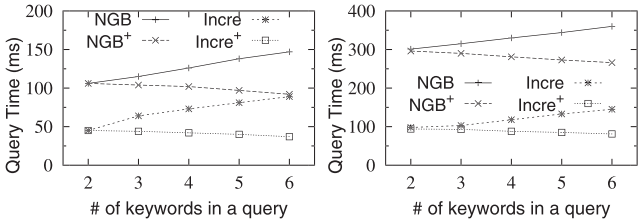
9. We set $q = 2$ and used the techniques of count filtering, length filtering, and position filtering.

Authorized licensed use limited to: St. Xavier's Catholic College of Engineering. Downloaded on October 22,2024 at 09:44:08 UTC from IEEE Xplore. Restrictions apply.



(a) DBLP (b) MEDLINE

Fig. 9. Fuzzy-search performance of computing similar keywords for single-keyword queries by varying the query keyword length ($\tau = 2$).



(a) DBLP (b) MEDLINE

Fig. 10. Fuzzy-search performance (overall) of computing first-N answers for multikeyword queries by varying the keyword number in a query ($\tau = 2$).

3. using the neighborhood-generation-based method (called "NGB"); and
4. using the character-level incremental algorithms (called "Incre") to compute similar keywords for a given query keyword using the prefix table as discussed in Section 4.2.

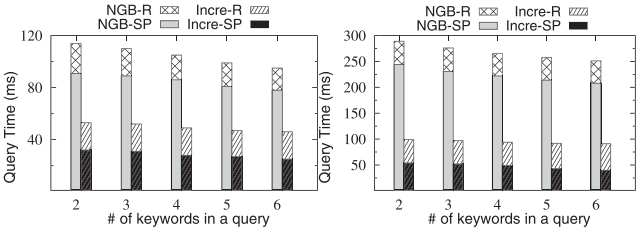
Fig. 9 shows the results.

As the keyword length increased, the running time of Gram, UDF, and NGB increased while that of Incre decreased. The main reason was the following: first, the UDF-based method needed more time for computing edit distances for longer strings. Second, long strings have many more i-deletion neighborhoods, and NGB needed longer time to find an i-deletion neighborhood of the query string from the deletion table. Third, there were more grams for longer strings and Gram needed longer time to process large numbers of grams. Besides the inverted-index table and prefix table, Gram and NGB maintained additional indexes. Fourth, Incre can incrementally compute the similar prefixes and longer strings have a smaller number of similar prefixes, thus its running time decreased.

Multikeyword queries. We evaluated the performance of different methods to compute first-N answers for multikeyword queries. Gram and the UDF-based methods were too slow to support search-as-you-type. We implemented two algorithms using NGB and Incre to find similar keywords on top of the prefix table, and then computed the answers based on the inverted-index table. For multikeyword queries, we also implemented their word-level incremental algorithms, called NGB$^+$ and Incre$^+$, respectively.

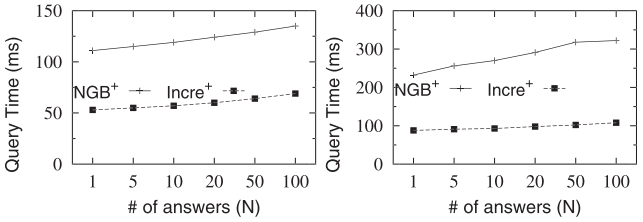
Fig. 10 shows the results. We see that the word-level incremental algorithms can improve the performance for multikeyword queries by using previously computed results to answer queries. For example, Incre$^+$ achieved a very high performance; it could answer a query within 50 ms for the DBLP data set and 100 ms for the MEDLINE data set.

In addition, we evaluated the running time in two steps: 1) finding similar keywords (called "NGB-SP" and "Incre-SP"); 2) computing first-N answers (called "NGB-R" and "Incre-R"). Fig. 11 shows the results. We see that NGB needed more time for finding similar keywords and Incre needed nearly the same amount time for the two steps. For example, on the MEDLINE data set, NGB needed 200 ms to compute similar prefixes and 50 ms to compute the answers for queries with six keywords. Incre reduced the time to 50 ms for computing similar prefixes.



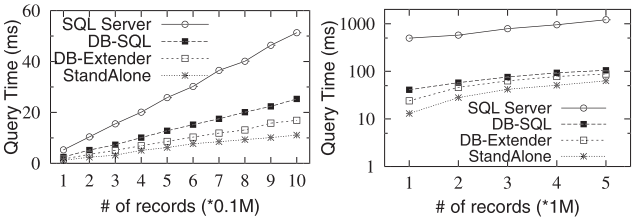
(a) DBLP (b) MEDLINE

Fig. 11. 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. 12. Fuzzy-search performance of computing first-N answers by varying different N values.



(a) Exact search (DBLP) (b) Fuzzy search (MEDLINE)

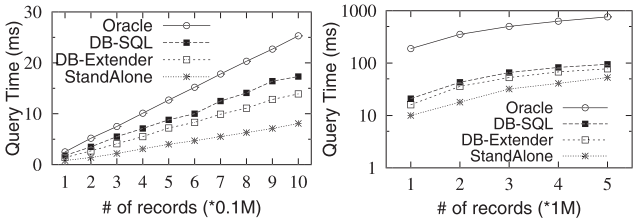
Fig. 13. Comparison of different methods (SQL Server).

Varying the number of returned results (N). We compared the performance of different algorithms to compute first-Nanswers by varying N. Fig. 12 shows the results. We can see that both Incre$^+$ and NGB$^+$ can efficiently compute the first-N answers for different N values.

8.3 Comparisons of Different Approaches

We compared different methods to support search-as-you-type. The first method uses existing built-in functionalities (e.g., full-text indexes and the CONTAINS command) in Oracle and SQL Server. We used their indexes to maximize their performance. The second method builds a separate application layer on the DBMS using techniques in [24], namely "StandAlone." For the extender-based method, we implemented the proposed techniques in [24] and added them as extenders of Oracle Cartridge in Oracle 11g and CLR in Microsoft SQL Server. The fourth method is based on SQL. We evaluated the scalability for both exact search and fuzzy search. We set $\tau = 2$ and $N = 100$. Figs. 13 and 14 show the results.

Authorized licensed use limited to: St. Xavier's Catholic College of Engineering. Downloaded on October 22,2024 at 09:44:08 UTC from IEEE Xplore. Restrictions apply.



(a) Exact search (DBLP) (b) Fuzzy search (MEDLINE)

Fig. 14. Comparison of different methods (Oracle).

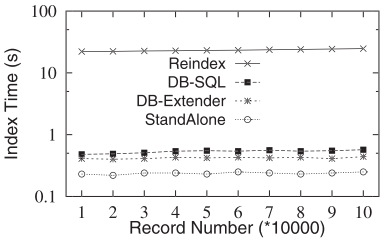


Fig. 15. Performance of updates (10,000 records).

We see that the StandAlone method achieved the highest performance as it used in-memory data index structures, while the SQL-based method (DB-SQL) had a lower but comparable performance. DB-SQL outperformed the built-in support of Oracle and SQL Server, since we can incrementally compute answers using effective index structures, which is very important for search-as-you-type. For exact search, all these methods achieved a high performance and scaled well as the data set increased. For example, for exact search, DB-SQL could answer a query within 2 ms for 100,000 records and 18 ms for one million records. For fuzzy search,[10] DB-SQL outperformed the built-in support in Oracle and SQL Server by 1-2 orders of magnitude. DB-SQL scaled well as data sizes increased, which reflects the superiority of our techniques. For example, the methods using built-in capabilities of Oracle and SQL Server took about 1,000 milliseconds to answer a query. DB-SQL could answer a query within 20 ms for one million records and 100 ms for five million records. This is because they cannot incrementally answer a query. Note that Oracle and Microsoft took fuzzy search as a black box and we do not know how they support fuzzy search. Here we took them as a baseline. The results suggested that practitioners need to consider both the performance and other system aspects to decide the best approach for search-as-you-type.

8.4 Data Updates

We tested the cost of updates on the DBLP data set. We first built indexes for 1 million records, and then inserted 10,000 records at each time. We compared the performance of the three methods on inserting 10,000 records. Fig. 15 shows the results. 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.

9 Related Work

Autocompletion and search-as-you-type. An autocompletion system can predict a word or phrase that a user may type in next based on the partial string the user has already typed [40]. Nandi and Jagadish studied phrase prediction, which took the query string as a single keyword and computed all sentences with a phrase of the query string. Bast et al. proposed HYB indexes to support search-as-you-type [4], [5], [6]. Ji et al. [24] extended autocompletion to support fuzzy, full-text instant search. Chaudhuri and Kaushik [14] studied how to find similar strings interactively as users type in a query string, and they did not study how to support multikeyword queries. Li et al. [34] studied search-as-you-type on a data base with multiple tables by modeling relational data as graphs. Qin et al. [42] proposed to use SQL to answer traditional keyword search, taking the query keywords as complete keywords. Li et al. [32] proposed to suggest SQL queries based on keywords. Different from existing studies [24], we study how to use SQL to support search-as-you-type. We proposed to use the available resources inside a DBMS and develop effective pruning techniques to improve the performance.

Approximate string search and similarity join. There have been recent studies to support efficient approximate string search [9], [26], [3], [13], [11], [18], [30], [31], [27], [50], [19], [45], which, given a set of strings and a query string, all strings in the set that are similar to the query string. Many studies used gram-based index structures to support approximate string search (e.g., [30], [28], [18]). The experiments in [24] and [14] showed that these approaches are not as efficient as trie-based methods for fuzzy search. Similarity joins are extensively studied [17], [3], [7], [12], [43], [48], [49], [46], which given two sets of strings, find all similar string pairs from the two sets. Gravano et al. [17] proposed to use DBMS capabilities to support fuzzy joins of strings. Their methodology (q-gram-based techniques) has a low performance to support search-as-you-type (the experimental results in Section 8). Jestes et al. [23] proposed to use min-hash to improve performance. Chaudhuri et al. [10] studied data cleansing operators in Microsoft SQL Server. There are also some studies on estimating selectivity of approximate string queries [20], [28], [29] and approximate entity extraction [1], [9], [47].

Keyword search in data bases. There are many studies on keyword search in data bases [22], [2], [8], [25], [36], [37], [35], [44], [15], [16], [37], [21], [39], [41], and [33].

Our work complements these earlier studies by investigating how to support search-as-you-type inside DBMS. To our best knowledge, our work is the first study on supporting search-as-you-type inside a DBMS using SQL, even supporting multikeyword queries and fuzzy search.

10 Conclusion and Future Work

In this paper, we studied the problem of using SQL to support search-as-you-type in data bases. We focused on the challenge of how to leverage existing DBMS functionalities to meet the high-performance requirement to achieve

10. In Oracle one can set a fuzzy score between 0 and 80 to do fuzzy search, we used the default value 60.

Authorized licensed use limited to: St. Xavier's Catholic College of Engineering. Downloaded on October 22,2024 at 09:44:08 UTC from IEEE Xplore. Restrictions apply.

an interactive speed. To support prefix matching, we proposed solutions that use auxiliary tables as index structures and SQL queries to support search-as-you-type. We extended the techniques to the case of fuzzy queries, and proposed various techniques to improve query performance. We proposed incremental-computation techniques to answer multikeyword queries, and studied how to support first-N queries and incremental updates. Our experimental results on large, real data sets showed that the proposed techniques can enable DBMS systems to support search-as-you-type on large tables.

There are several open problems to support search-as-you-type using SQL. One is how to support ranking queries efficiently. Another one is how to support multiple tables.

ACKNOWLEDGMENTS

Guoliang Li and Jianhua Feng are partially supported by the National Natural Science Foundation of China under Grant No. 61003004 and 61272090, the National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302206, and National S&T Major Project of China under Grant No. 2011ZX01042-001-002. Chen Li is partially supported by the NIH grant 1R21LM010143-01A1 and the NSF grant IIS-1030002. Chen Li declares financial interest in Bimaple Technology Inc., which is commercializing some of the techniques used in this publication.

REFERENCES

[1] 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.

[2] 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.

[3] A. Arasu, V. Ganti, and R. Kaushik, "Efficient Exact Set-Similarity Joins," *Proc. 32nd Int'l Conf. Very Large Data Bases (VLDB '06),* pp. 918-929, 2006.

[4] 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.

[5] 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.

[6] 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.

[7] R.J. Bayardo, Y. Ma, and R. Srikant, "Scaling up all Pairs Similarity Search," *Proc. 16th Int'l Conf. World Wide Web (WWW '07),* pp. 131-140, 2007.

[8] 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.

[9] K. Chakrabarti, S. Chaudhuri, V. Ganti, and D. Xin, "An Efficient Filter for Approximate Membership Checking," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '08),* pp. 805-818, 2008.

[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. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani, "Robust and Efficient Fuzzy Match for Online Data Cleaning," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '03),* pp. 313-324, 2003.

[12] S. Chaudhuri, V. Ganti, and R. Kaushik, "A Primitive Operator for Similarity Joins in Data Cleaning," *Proc. 22nd Int'l Conf. Data Eng. (ICDE '06),* pp. 5-16, 2006.

[13] S. Chaudhuri, V. Ganti, and R. Motwani, "Robust Identification of Fuzzy Duplicates," *Proc. 21st Int'l Conf. Data Eng. (ICDE),* pp. 865-876, 2005.

[14] 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.

[15] B.B. Dalvi, M. Kshirsagar, and S. Sudarshan, "Keyword Search on External Memory Data Graphs," *Proc. VLDB Endowment,* vol. 1, no. 1, pp. 1189-1204, 2008.

[16] B. Ding, J.X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin, "Finding Top-K Min-Cost Connected Trees in Data Bases," *Proc. IEEE 23rd Int'l Conf. Data Eng. (ICDE '07),* pp. 836-845, 2007.

[17] 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.

[18] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava, "Fast Indexes and Algorithms for Set Similarity Selection Queries," *Proc. IEEE 24th Int'l Conf. Data Eng. (ICDE '08),* pp. 267-276, 2008.

[19] M. Hadjieleftheriou, N. Koudas, and D. Srivastava, "Incremental Maintenance of Length Normalized Indexes for Approximate String Matching," *Proc. 35th ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '09),* pp. 429-440, 2009.

[20] M. Hadjieleftheriou, X. Yu, N. Koudas, and D. Srivastava, "Hashed Samples: Selectivity Estimators for Set Similarity Selection Queries," *Proc. VLDB Endowment,* vol. 1, no. 1, pp. 201-212, 2008.

[21] H. He, H. Wang, J. Yang, and P.S. Yu, "Blinks: Ranked Keyword Searches on Graphs," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '07),* pp. 305-316, 2007.

[22] V. Hristidis and Y. Papakonstantinou, "Discover: Keyword Search in Relational Data Bases," *Proc. 28th Int'l Conf. Very Large Data Bases (VLDB '02),* pp. 670-681, 2002.

[23] 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.

[24] 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.

[25] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar, "Bidirectional Expansion for Keyword Search on Graph Data Bases," *Proc. 31st Int'l Conf. Very Large Data Bases (VLDB '05),* pp. 505-516, 2005.

[26] M.-S. Kim, K.-Y. Whang, J.-G. Lee, and M.-J. Lee, "N-Gram/2l: A Space and Time Efficient Two-Level N-Gram Inverted Index Structure," *Proc. 31st Int'l Conf. Very Large Data Bases (VLDB '05),* pp. 325-336, 2005.

[27] N. Koudas, C. Li, A.K.H. Tung, and R. Vernica, "Relaxing Join and Selection Queries," *Proc. 32nd Int'l Conf. Very Large Data Bases (VLDB '06),* pp. 199-210, 2006.

[28] H. Lee, R.T. Ng, and K. Shim, "Extending Q-Grams to Estimate Selectivity of String Matching with Low Edit Distance," *Proc. 33rd Int'l Conf. Very Large Data Bases (VLDB '07),* pp. 195-206, 2007.

[29] H. Lee, R.T. Ng, and K. Shim, "Power-Law Based Estimation of Set Similarity Join Size," *Proc. VLDB Endowment,* vol. 2, no. 1, pp. 658-669, 2009.

[30] C. Li, J. Lu, and Y. Lu, "Efficient Merging and Filtering Algorithms for Approximate String Searches," *Proc. IEEE 24th Int'l Conf. Data Eng. (ICDE '08),* pp. 257-266, 2008.

[31] C. Li, B. Wang, and X. Yang, "VGRAM: Improving Performance of Approximate Queries on String Collections Using Variable-Length Grams," *Proc. 33rd Int'l Conf. Very Large Data Bases (VLDB '07),* pp. 303-314, 2007.

[32] 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.

[33] 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.

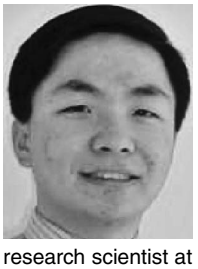
[34] G. Li, S. Ji, C. Li, and J. Feng, "Efficient Type-Ahead Search on Relational Data: A Tastier Approach," *Proc. 35th ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '09),* pp. 695-706, 2009.

Authorized licensed use limited to: St. Xavier's Catholic College of Engineering. Downloaded on October 22,2024 at 09:44:08 UTC from IEEE Xplore. Restrictions apply.

[35] G. Li, B.C. Ooi, J. Feng, J. Wang, and L. Zhou, "EASE: An Effective 3-in-1 Keyword Search Method for Unstructured, Semi-Structured and Structured Data," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '08),* pp. 903-914, 2008.
[36] 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.
[37] Y. Luo, X. Lin, W. Wang, and X. Zhou, "Spark: Top-K Keyword Query in Relational Data Bases," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '07),* pp. 115-126, 2007.
[38] R.B. Miller, "Response Time in Man-Computer Conversational Transactions," *Proc. AFIPS '68: Fall Joint Computer Conf., Part I,* pp. 267-277, 1968.
[39] S. Mitra, M. Winslett, W.W. Hsu, and K.C.-C. Chang, "Trustworthy Keyword Search for Compliance Storage," *VLDB J.—Int'l J. Very Large Data Bases,* vol. 17, no. 2, pp. 225-242, 2008.
[40] A. Nandi and H.V. Jagadish, "Effective Phrase Prediction," *Proc. 33rd Int'l Conf. Very Large Data Bases (VLDB '07),* pp. 219-230, 2007.
[41] L. Qin, J. Yu, and L. Chang, "Ten Thousand Sqls: Parallel Keyword Queries Computing," *Proc. VLDB Endowment,* vol. 3, no. 1, pp. 58-69, 2010.
[42] 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.
[43] S. Sarawagi and A. Kirpal, "Efficient Set Joins on Similarity Predicates," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '04),* pp. 743-754, 2004.
[44] T. Tran, H. Wang, S. Rudolph, and P. Cimiano, "Top-K Exploration of Query Candidates for Efficient Keyword Search on Graph-Shaped (RDF) Data," *Proc. IEEE Int'l Conf. Data Eng. (ICDE '09),* pp. 405-416, 2009.
[45] E. Ukkonen, "Finding Approximate Patterns in Strings," *J. Algorithms,* vol. 6, no. 1, pp. 132-137, 1985.
[46] 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.
[47] W. Wang, C. Xiao, X. Lin, and C. Zhang, "Efficient Approximate Entity Extraction with Edit Distance Constraints," *Proc. 35th ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '09),* pp. 759-770, 2009.
[48] C. Xiao, W. Wang, and X. Lin, "Ed-Join: An Efficient Algorithm for Similarity Joins with Edit Distance Constraints," *Proc. VLDB Endowment,* vol. 1, no. 1, pp. 933-944, 2008.
[49] C. Xiao, W. Wang, X. Lin, and H. Shang, "Top-K Set Similarity Joins," *Proc. IEEE Int'l Conf. Data Eng. (ICDE '09),* pp. 916-927, 2009.
[50] C. Xiao, W. Wang, X. Lin, and J.X. Yu, "Efficient Similarity Joins for Near Duplicate Detection," *Proc. 17th Int'l Conf. World Wide Web (WWW '08),* 2008.

Guoliang Li received the PhD degree in computer science from Tsinghua University in 2009, and the bachelor's degree in computer science from Harbin Institute of Technology in 2004. He is an assistant professor in the Department of Computer Science, Tsinghua University, Beijing, China. His research interests include integrating data bases and information retrieval, data cleaning, data base usability, and data integration.

Jianhua Feng received the BS, MS, and PhD degrees in computer science and technology from Tsinghua University. He is currently working as a professor in the Department of Computer Science and Technology at Tsinghua University. His main research interests include data bases, native XML data bases, and keyword search over structured data. He is a member of the ACM and the IEEE, and a senior member of China Computer Federation (CCF).



Chen Li received the BS and MS degrees in computer science from Tsinghua University, China, in 1994 and 1996, respectively, and the PhD degree in computer science from Stanford University in 2001. He is an associate professor in the Department of Computer Science at the University of California, Irvine. He received a US National Science Foundation (NSF) CAREER Award in 2003 and a few other NSF grants and industry gifts. He was once a part-time visiting research scientist at Google. His research interests include the fields of data management and information search. He is the founder of Bimaple.com. He is a member of the IEEE.

▹ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

Authorized licensed use limited to: St. Xavier's Catholic College of Engineering. Downloaded on October 22,2024 at 09:44:08 UTC from IEEE Xplore. Restrictions apply.