Extending Autocompletion To Tolerate Errors

Surajit Chaudhuri
Microsoft Research
One Microsoft Way
Redmond, WA, USA
surajitc@microsoft.com

Raghav Kaushik
Microsoft Research
One Microsoft Way
Redmond, WA, USA
skaushi@microsoft.com

ABSTRACT

Autocompletion is a useful feature when a user is doing a look up from a table of records. With every letter being typed, autocompletion displays strings that are present in the table containing as their prefix the search string typed so far. Just as there is a need for making the lookup operation tolerant to typing errors, we argue that autocompletion also needs to be error-tolerant. In this paper, we take a first step towards addressing this problem. We capture input typing errors via edit distance. We show that a naive approach of invoking an offline edit distance matching algorithm at each step performs poorly and present more efficient algorithms. Our empirical evaluation demonstrates the effectiveness of our algorithms.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

General Terms

Design, Algorithms, Experimentation

Keywords

Autocompletion, Edit Distance

1. INTRODUCTION

Autocompletion is a ubiquitous feature found to be useful in several environments. As the user types, a list of appropriate completions is returned by such a feature. The goal is not only to reduce typing effort but also to help guide the user's typing.

Not surprisingly, autocompletion has found wide adoption as a feature in a variety of applications. This feature is available today in program editors such as Visual Studio, command shells such as the Unix Shell, search engines such as Google, Live and Yahoo, and desktop search. Autocompletion is also gaining popularity for mobile devices since it can assist the user in keying in contacts and text messages.

Autocompletion is also valuable in the database context as such a feature can help ensure data integrity by substantially reducing the probability of data entry errors. The specific scenario that is our focus is when a user is looking up a record from a table by entering a string, such as when a sales clerk is looking up a customer's name or one is looking up a product catalog online. For instance, `http://www.amazon.com` suggests completions when we are looking up a product item. Similarly, `Yahoo Finance` suggests completions when we are looking for a stock symbol or organization name.

One way of viewing autocompletion is as an *online* method of performing exact matching, since in the absence of autocompletion, a user would have to type out the string in its entirety and then match it against the table of records. In contrast to the online autocompletion process, we call the latter an *offline* lookup.

In an offline lookup setting, the data cleaning community has long recognized the need to go beyond exact matching. This is because the input string being typed can contain errors and differences in representation making exact matching inadequate. This observation has motivated the vast body of work on *record matching* [12].

In this paper, we argue that in the same way, the scope of autocompletion should be extended to tolerate errors and differences in representation. At first glance, this may not be obvious since even exact autocompletion does act as a hint in guiding the user's typing and to some extent reduces the likelihood of errors being committed in the first place. However, there are many important scenarios where error-tolerant autocompletion would add significant value. For example, consider the name `Schwarzenegger`. It is quite likely that a user looking up this name is likely to start with the prefix `Shwarz` or `Swarz` instead of the correct prefix `Schwarz`. In this case, the only opportunity for an exact autocompletion approach to suggest the correct completion is when the prefix `S` has been entered; but at this point, the number of completions is likely to be too high to be useful in a database of any realistic scale.

This problem is only exacerbated in domains such as product model numbers where we cannot rely on phonetics and intuition to guess the correct spelling. Further, even if we knew the correct spelling of a string, we may still mistype, especially when typing fast. There is no reason to believe that such typing errors are less likely at the beginning of the string. Error-tolerant autocompletion is also useful in maintaining data integrity since being quickly able to browse approximate completions can reduce duplicate entries.

Motivated by these observations, we study the problem of error-tolerant autocompletion. We begin by laying out the framework for exact autocompletion (Section 2) which we then extend to tolerate errors (Section 3). Just as we view exact autocompletion as an online version of exact lookup, we model the error-tolerant autocompletion problem as an online version of error-tolerant lookup. As is standard in data cleaning, in order to define what it means to tolerate errors, we need to use a notion of string similarity. While there are several sophisticated ways of modeling errors in data [12], in this paper, we take a first step by adopting the classical edit distance as our measure of similarity between strings. We generalize the notion of the extension of a string to tolerate edit errors. We show that it is possible to implement this strategy by using an vanilla edit distance matching algorithm at each step and argue that this can be quite expensive (Section 3.4). Accordingly, we propose two improved edit-tolerant autocompletion algorithms. The first is based on the state-of-the-art q-gram based edit distance matching algorithms (Section 4). The second is a trie-based algorithm that is even more suited to autocompletion (Section 5).

Often, when only a few characters of the lookup string have been entered, there are too many completions for autocompletion to be useful. We thus consider an alternate *buffered* strategy that performs autocompletion only after a few input characters have been entered (Section 5.2). We use pre-computation to handle kick start the autocompletion. By hashing characters to a small number of bits and exploiting the fact that we are performing pre-computation only for short strings, we control the amount of state needed for pre-computation. We show that pre-computation coupled with the trie has formal guarantees for edit distance matching (Section 5.3). Not surprisingly, we find that pre-computation further improves the performance of the trie-based algorithm significantly.

Often, the records in the table being looked up have an application specific *static* score. For example, in a table of product records, the static score could be used to reflect the popularity of the product, say based on the number of recent purchases. It is natural to factor this in addition to the edit distance in ordering the autocompletion output. We show how to extend our algorithms to return only the top-l extensions exploiting such an ordering (Section 5.4).

We then conduct a detailed empirical evaluation of the techniques we propose over real data sets (Section 6). Among other things, we empirically show the effectiveness of error tolerant autocompletion by measuring the additional key strokes saved over and above exact autocompletion. By comparing edit-tolerant autocompletion with offline edit-distance matching, we also study the additional overhead incurred in invoking an online approach to edit distance matching. We show that while our solution does impose an additional overhead as expected, this is substantially better than the additional overhead imposed by exact autocompletion when compared to exact string lookup.

We discuss related work in Section 7 and conclude in Section 8. Finally, we note that despite the availability of error-tolerant autocompletion in a limited number of products, little is known about their underlying algorithms. To the best of our knowledge, this is the first piece of work that systematically studies error-tolerant autocompletion in the context of database lookups.

2. AUTOCOMPLETION: STATE OF THE ART

At its core, autocompletion is about suggesting valid completions of a partially entered lookup string with the intention of minimizing and guiding the user's typing. The concept of autocompletion is not new. There are various approaches studied in prior work on autocompletion. There is a vast body of work on *predictive* autocompletion [4, 8, 16] where the idea is to use information retrieval techniques, language models and learning to suggest potential completions.

In contrast, our focus is on the scenario where there is a table T of strings being looked up and thus completions are suggested based on matches in T. This is the form of autocompletion supported by `http://www.amazon.com` and `Yahoo Finance` for instance. The most common form of such autocompletion is based on exact matching. In this section, we formalize some of the key concepts involved in exact autocompletion.

2.1 Autocompletion Interface

Autocompletion is an *online* problem where at any point, there is a partially typed string s that we call the *lookup string*. In response, the autocompletion produces a list $Completions(s)$. The lookup string is modified via some user *move*. In general, the user can modify the string using any of a large space of moves — the user can append a character, insert or delete a character either at the end of the string or anywhere in the string, choose one of the suggested completions, and invoke a lookup. In this paper, we focus primarily on the following user moves.

- $Append(c)$: append a character to the end of the lookup string s,
- $Choose(s')$ where $s' \in Completions(s)$: choose one of the suggested completions.

We choose these moves since these are the most common moves made by the user. We defer a study of other moves to future work.

2.2 Autocompletion Strategy

There are multiple ways in which exact autocompletion may be performed. We call these possibilities *autocompletion strategies.*

Perhaps the simplest strategy is to return all strings in T that are extensions of the lookup string. When the number of characters in the lookup string is very small — at the extreme case when only the first character has been entered — the number of extensions can be too large to be useful. This motivates an alternate strategy where we perform autocompletion only after a minimum number of characters have been input.

Another exact autocompletion approach is to return at each point all strings in T that contain the partially entered lookup string as a substring. This is the strategy supported by `Yahoo Finance` for instance. As with the prefix based approach above, we could consider doing this only after the lookup string has a minimum number of characters.

In general, the autocompletion strategy could be quite complex as illustrated by prior work on multi-word autocompletion [2, 15, 16].

Often, we can order the completions by leveraging an application specific *static score* assigned to each string in T. For example, if T represents a table of products and we store a log of lookup queries posed against T, then we can use the static score to reflect the popularity of a product, say based on the number of recent purchases. Alternately, the static score can be used to bias the lookup toward newer products. Another example is when T consists of author names and we use the static score to bias one subject area over others — an application that is targeted toward database users can use the static score to assign a higher preference to database authors.

A given autocompletion strategy can be supported by a variety of algorithms. For instance, if our strategy is to return all extensions of the lookup string, we could (a) at each point issue an offline prefix lookup using say a B-Tree that finds all extensions in T, or (b) use a *trie* to find all extensions in an online fashion.

2.3 Efficiency Considerations

An important goal of an autocompletion system is to be responsive — it must look instantaneous to the user. Prior work [14] has shown that this implies a maximum response time of 100ms. In a client-server setting, this 100ms bound includes not only the autocompletion time, but other overheads such as the communication overheads. Thus, it is desirable to make the per-character autocompletion cost itself minimal.

As noted above, we view autocompletion as an online method of performing the underlying lookup. We expect the online algorithm to be slower than an offline lookup. For instance, finding an exact match by hashing a string in its entirety is substantially faster than looking up a trie character by character. Thus, the benefits of autocompletion come with the price of invoking an online algorithm.

3. INCORPORATING ERROR TOLERANCE

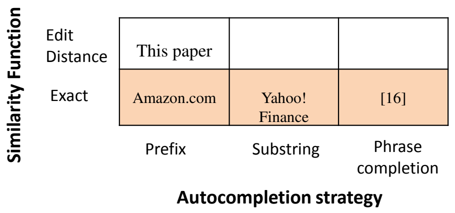

Figure 1: Autocompletion Methods

Our goal as argued in Section 1 is to extend autocompletion to be error tolerant. In Section 2, we considered various ways in which autocompletion could be performed. The question arises how error tolerance influences these options. In general, *any* of these methods can be extended to be error-tolerant. As shown in prior work on record matching [12], error tolerance can be achieved in many ways, for example by choosing different similarity functions. Any of these similarity functions can be used to make autocompletion error-tolerant. This leaves us with a two-dimensional space of options plotted in Figure 1. The figure shows examples of prior work covering exact autocompletion approaches. Note that only one similarity function (edit distance) is shown in the Figure for illustration. In general, the space of similarity functions is much larger.

In this paper, we focus on extending the prefix based approach to be error tolerant. The similarity function we choose is the classic edit distance. Even though our techniques generalize to also handle substring matching, we omit the details owing to a lack of space. In this section, we review the definition of edit distance, extend the concept of string extensions to tolerate string edits via the notion of k-Extensions. We then discuss some of the properties of k-Extensions which provide the basis for all algorithms we propose.

3.1 Edit Distance Based Matching

We choose edit distance as our notion of tolerating errors in this paper. Given two strings s_1 and s_2, we are allowed to insert and delete characters as well as replace one character with a different one. The minimum number of moves we need to perform on s_1 such that the result is equal to s_2 is the *edit distance* between the two strings, denoted $ed(s_1, s_2)$. We use the phrase "edit distance within k" to refer to the expression $ed(s_1, s_2) \leq k$.

Example 1. The string `Shwarzenegger` has edit distance 1 to the string `Schwarzenegger`. In order to transform `Shwarzenegger` to `Schwarzenegger`, we need to insert character `c` after `S`. □

Fix a string s and a threshold k. The *(offline) edit lookup* operation returns all strings $r \in T$ that are within edit distance k in increasing order of edit distance.

3.2 k-Extension

We now extend the concept of string extensions to be tolerant of string edits. Figure 2 shows an example of our intuition.

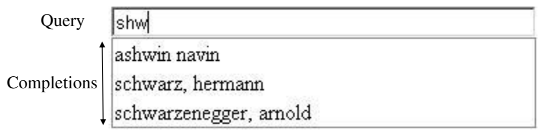

Figure 2: Error Tolerant Autocompletion

We formalize this intuition through the following definition.

DEFINITION 1. *A string s_1 is defined to be a k-*prefix *of string s_2, denoted $s_1 \prec^k s_2$ if there is some extension of s_1 that is within edit distance k of s_2. s_2 is called a k-*extension *of s_1. The smallest k such that s_2 is a k-extension of s_1 is called the* extension distance *of s_2 given s_1.*

We illustrate this definition with an example.

Example 2. In Figure 2, observe that each of the strings `Ashwin Navin`, `Schwarzenegger, Arnold` and `Schwarz, Hermann` is a 1-extension of `Shw`.

The extension of the input string that yields edit distance 1 to `Schwarzenegger` is `Shwarzenegger`. The extension distance of `Schwarzenegger` given `Shwarz` is 1. □

If instead of using string extensions, we were to perform edit-tolerant substring matching, then in Figure 2, we would return additional strings such as `Graeme Swann`.

We consider two autocompletion strategies in this paper. The first is as follows.

DEFINITION 2. *Fix an edit threshold k. The* Full k-Extension Strategy *is as follows: At* each *point as the lookup string is modified by appending characters, our goal is to return all k-extensions in the order of increasing extension distance.*

Often, when only a few characters of the lookup string have been entered, there are too many completions for autocompletion to be useful. We therefore consider a *Buffered Strategy* where we are required to return the k-extensions, but only after the lookup string has a minimum number b of characters. We refer to this number b as the *transition length.*

As with exact autocompletion, we can leverage the application specific *static score* associated with each string in T, if available. We assume an overall scoring function that is *monotonic* [6] in both the edit distance and the static score. We can return completions ordered by this overall scoring function, independent of the specific function chosen so long as it is monotonic.

Finally, as argued in Section 2.3, we expect online autocompletion to be slower than offline matching. However, our goal is to make this efficiency gap comparable to the case of exact autocompletion.

3.3 Properties of k-Extensions

We now discuss some properties of k-Extensions. We first establish the relationship between extensions and prefixes when allowing for string edits. We use this relationship to show how to compute the pairwise extension distance defined above by adapting the classic dynamic programming based edit distance computation algorithm [17]. This discussion forms the basis for all algorithms in this paper.

3.3.1 Relating Extensions to Prefixes

First we show the following equivalence which forms the basis for all algorithms we propose in this paper.

PROPERTY 1. $s_1 \prec^k s_2$ *if and only if there is some prefix* s_2' *of* s_2 *such that* s_1 *and* s_2' *are within edit distance* k.

We illustrate this property with an example below.

Example 3. As noted earlier, the string `Schwarzenegger` is a 1-extension of `Shwarz`. In this case, the prefix `Schwarz` is within edit distance 1 of `Shwarz`. □

3.3.2 Pairwise Extension Distance

We develop our algorithm to solve this problem by first considering the following more basic problems.

- *Extension Distance:* Given strings s_1 and s_2, compute the extension distance of s_2 given s_1.
- *k-Extension Distance:* Given strings s_1 and s_2, check if the extension distance of s_2 given s_1 is at most k and if so, compute the extension edit distance.

These problems can be solved by a straightforward adaptation of the standard dynamic programming algorithm for computing the edit distance between s_1 and s_2 [17]. We first review this algorithm. Suppose that the two strings under consideration are s_1 and s_2. We place the two strings on a matrix D with s_1 top-down and s_2 left-to-right, and incrementally compute the edit distance between all prefixes of s_1 and s_2. Figure 3 illustrates the dynamic programming matrix for the two strings s_1="Jon" and s_2="Johnny". Let i be the index of the rows in the dynamic programming matrix and j be the index of the columns. The row numbers increase as we go down whereas the column numbers increase from left to right. Both begin at 0. The numbers in parentheses in Figure 3 indicate the row and column numbers. The number entered in cell $D(i,j)$ denotes the edit distance between the prefixes ending at i and j respectively. The recurrence relation that completes D is as follows [17].

	(0)	J(1)	o(2)	h(3)	n(4)	n(5)	y(6)
(0)	0	1	2	3	4	5	6
J(1)	1	0	1	2	3	4	5
o(2)	2	1	0	1	2	3	4
n(3)	3	2	1	1	1	2	3

Figure 3: Pairwise Edit Distance

$$\begin{aligned} D(i,j) = \min(&D(i-1,j)+1, \\ &D(i,j-1)+1, \\ &D(i-1,j-1)+\delta(i,j)) \end{aligned}$$

where $\delta(i,j)$ is 0 or 1 according as the i^{th} character of s_1 and the j^{th} character of s_2 are equal. For example,

$$\begin{aligned} D(\text{“Jo”},\text{“Joh”}) = D(2,3) = \min(&D(1,3)+1, D(2,2)+1, \\ &D(1,2)+\delta(2,3)) = \min(3,1,2) = 1 \end{aligned}$$

Note that in this process, we find the edit distance between s_1 and all prefixes of s_2 — this is captured in the last row of D. We can then find the prefix of s_2 with the smallest edit distance from s_1 by finding the smallest entry in the last row. Using Property 1, we can see that this yields the extension distance. We can see from Figure 3 that even though the edit distance is 3, the extension distance is 1 for s_1 and s_2 as defined above.

Now consider the *k-Extension Distance* problem. Intuitively, we can ignore parts of the matrix D where the value is guaranteed to be larger than k. The following property of matrix D follows from the recurrence relation above and lets us formalize this intuition.

PROPERTY 2. *(1)* $D(i,0) = i$ *and* $D(0,j) = j$, *and (2)* $D(i,j) \geq D(i-1,j-1)$.

Now define the *c-diagonal* to be all cells such that $i - j = c$ (c can be negative). By Property 2, it follows that we only need to track the entries of D in diagonals $-k$ through k — all other cells in D are guaranteed to have values larger than k. For each cell in these diagonals, we need to store the edit distance only if it is at most k (otherwise we store ∞). The recurrence relation can be used to compute the edit distance so long as it is at most k. Then we read off the minimum value in the last row as before to compute the extension distance. Figure 4 illustrates this for the case $k = 1$. Observe that this algorithm takes $O(kn)$ time where n is the length of s_1.

Finally, we can see that the above algorithm is naturally incremental. Adding a new character to s_1 corresponds to

	(0)	J(1)	o(2)	h(3)	n(4)	n(5)	y(6)
(0)	0	1					
J(1)	1	0	1				
o(2)		1	0	1			
n(3)			1	1	1		

Figure 4: 1-Extension Distance

adding a new row to the matrix D. By the form of the recurrence relation, we see that the old entries of D do not change and that the $2k+1$ entries for this row can be computed left to right from the old entries of D in constant time per new entry.

3.4 Baseline Algorithm for Edit-Tolerant Autocompletion

We can use Property 1 coupled with any offline edit distance matching algorithm to implement both the *Full* and *Buffered* strategies. We first introduce some notation to talk about the prefixes of strings. Given a string r, the set consisting of r and all its prefixes is denoted $\bar{r}$. Given a table of strings T, the set of strings in T along with all their prefixes is denoted $\bar{T}$.

We index all strings in $\bar{T}$. At any point in the autocompletion, we invoke the offline algorithm to find matching strings in $\bar{T}$ and then return their corresponding extensions. We call this the *baseline* algorithm. It is sketched in Algorithm 3.1 for the *Full* strategy and can be trivially extended to handle the *Buffered* strategy.

Algorithm 3.1 Baseline Algorithm

```
Input:   String $s$ input incrementally with new
         character being appended at each step and edit threshold $k$.
Output:  At each step, all $k$-extensions
1. Build an index for offline edit matching over all strings in $\bar{T}$
2. For each character $c$ of the input
3.     Use the index to find all $r \in T$ such that
       some prefix of $r$ is within edit distance $k$ of $s$
```

The baseline algorithm has serious shortcomings. If we performed no autocompletion, we would invoke offline matching on the entire lookup string. This matching would occur over the strings in T. In contrast, the baseline algorithm invokes offline matching on a much larger set, namely $\bar{T}$. Further, this matching is performed for each character appended.

There is considerable room for improvement over this approach, by exploiting (1) the structure of the set of being indexed, namely $\bar{T}$, and (2) the commonality among the successive lookups which only differ in one character. This is the focus of the next two sections.

4. Q-GRAM BASED ALGORITHM

We now discuss our q-gram based autocompletion algorithms. Q-gram based techniques constitute the state-of-the-art algorithms for offline edit distance matching [1, 3, 9, 21]. We first briefly review these algorithms before introducing our extensions for autocompletion.

4.1 Review of Offline Q-Gram Based Lookup

A q-gram of a string s is a contiguous substring of s of length q. The q-gram set is the bag of all q-grams of s. Intuitively, if the edit distance between two strings s and r is small, then the overlap between the corresponding q-gram sets must be large. Formally if $ed(r,s) \le k$ then the (bag) intersection between their q-gram sets must be at least $(max(|r|,|s|)-q+1)-q\cdot k$ where $|r|$ and $|s|$ denote the lengths of r and s respectively [9].

Example 4. The edit distance between `Shwarzenegger` and `Schwarzenegger` is 1. Consider their 1-gram sets which is the set of all characters in the strings. Their intersection size is 13 which is larger than or equal to $(\max(13,14)-1+1)-1\cdot 1=13$.

This relationship is used to invoke a set-similarity based matching. The detail of set-similarity matching that is relevant here is that most previously proposed algorithms are based on *signature schemes* [1]. The idea is to create a set of signatures for each string based on its q-gram set. The signature scheme must have the property that whenever two strings are within edit distance k, they share at least one common signature. Examples of signature schemes are *Prefix-Filter* [3, 21], *PartEnum* [1] and *Locality Sensitive Hashing*(LSH) [7]. The index consists of an inverted list that maps a signature to all strings that generate this signature. At lookup time, signatures are generated for the lookup string and the *union* of all the corresponding rid-lists is taken. Each string in this union is then passed through a *verification* where we check whether its edit distance to the lookup string is indeed within k. This verification step is necessary since the signature lookup can generate false positives.

4.2 Q-Gram Based Autocompletion

Fix a signature scheme *Sig*. Consider a string $r \in T$. Recall from Property 1 that we need to consider returning r whenever *some* prefix of r is within edit distance k of the lookup string.

We illustrate the problem of using the q-gram approach coupled with the baseline algorithm shown in Figure 3.1 through an example. Suppose that T consists of the single string r = `Schwarzenegger`. Suppose also that the signature scheme *Sig* returns all 1-grams. We have one inverted list per character of the string `Schwarzenegger`. Each of these lists contains all prefixes of r that contain the respective character. For instance, the list for character `'S'` contains all (non-empty) prefixes of r. This is shown in the column called “Baseline List” in Figure 5. Now consider the lookup string `Shwarz`. Under the baseline algorithm, we consider each string in the inverted list of `'S'` for verification. Thus, we invoke the k-Extension Distance algorithm discussed in Section 3.3.2 for every prefix of the string `Schwarzenegger`.

Signature	Baseline List	Modified List
`'S'`	{`S, Sc, Sch, ..., Schwarzenegger`}	{`Schwarzenegger`}
...	...	...

Figure 5: Q-gram approach

We improve upon this as follows. We modify the signature scheme to obtain a signature scheme Sig' where $Sig'(r) = \cup_{r' \in \bar{r}} Sig(r')$. Since the strings in $\bar{r}$ have substantial overlap, they generate many common signatures.

Unlike the baseline approach, these common signatures are represented only once. We build an inverted index over the signatures generated by Sig'. The inverted index for the character `'S'` in our example above consists of the single string `Schwarzenegger` — this is shown in the column marked "Modified List" in Figure 5.

The verification phase can be optimized by exploiting the commonality among the strings in $\bar{r}$. We noted in Section 3.3.2 that the k-Extension Distance computation between r and s actually performs the k-Extension Distance computation for all pairs of strings in $\bar{r}, \bar{s}$. By Property 1, we can perform verification in one invocation of the dynamic programming algorithm described in Section 3.3.2.

We can further optimize the q-gram based algorithm by exploiting the the fact that successive lookup strings only differ in one character. At each step, we avoid re-scanning the lists of signatures that we have already examined. For example, since the signatures for `Sh` and `Shw` contain the character `'S'`, the list for `'S'` is accessed only once.

These optimizations lead to significant improvements in the running time of the q-gram based algorithm. However, despite these optimizations, the number of strings being retrieved at each step for verification can be significant leading to poor performance as we will see in Section 6. Ideally, we would like to "transition" from the results in one step to the results in the next. We achieve this in an automaton-style traversal over a trie.

5. TRIE-BASED ALGORITHM

We now discuss our trie-based autocompletion algorithms. The idea is to transition from the k-extensions in one step to the next just as we do in an automaton. This also results in a novel trie-based algorithm for offline edit distance matching that processes the lookup string character by character (see Section 7 for a detailed discussion of related work).

5.1 Full k-Extension Strategy

We organize the set of strings in T as a trie. We represent the transitions as edge labels. Figure 6 shows an example trie. Owing to potential edit errors at any given time, we could be at multiple nodes in the trie. The algorithm maintains the set of all prefixes of the strings in the database that are within edit distance k. We call the corresponding nodes in the trie *valid*. It can be shown that for any k-extension of the lookup string, there at most $2k + 1$ prefixes that are within distance k, and that these prefixes correspond to a contiguous path in the trie.

The pseudo-code is shown in Algorithm 5.1. As characters are appended to the input, we make transitions roughly corresponding to how we populate the edit matrix. We think of the input string as populating the rows of the edit matrix D. Before the next character is appended, we have the values for row i populated in the `valid` buffer in Algorithm 5.1. We are trying to populate the entries for the next row which is captured in the buffer `new-valid`.

Recall that the cells in row $i + 1$ are populated left to right. In the trie, this corresponds to a top-down traversal. Cell $D(i+1, j)$ is influenced by the values in $D(i+1, j-1)$, $D(i, j-1)$ and $D(i, j)$. In the trie, the value of $D(i+1, j-1)$ is stored in `new-valid` whereas the value of $D(i, j-1)$ is stored in `valid`. The influence of $D(i+1, j-1)$ and $D(i, j-1)$ is captured in Steps 12-16. The influence of $D(i, j)$ is captured in Steps 8-9. Note that we only consider marking

Algorithm 5.1 Trie-Based Autocompletion

```
Input:   String $s$ input incrementally with new
         character being appended at each step and edit threshold $k$.
Output:  At each step, all $k$-extensions
     Let valid denote the buffer storing current valid nodes
     Let new-valid denote the buffer storing the next set of valid
     nodes
     Let $root$ denote the root of the trie
 1. Add(valid, $root$, 0)
 2. For $i = 1 \ldots k$
 3.   For each node $nd$ at distance $i$ from $root$
 4.     Add(valid, $nd$, $i$)
 5. For each character $c$ of the input
        Traverse the nodes in valid in queue order
 6.     For each valid node $nd$
 7.       Let $l_{curr}$ = Distance(valid, $nd$)
 8.       If $l_{curr} + 1 \leq k$
 9.         Add(new-valid, $nd, l_{curr} + 1$)
10.       If $nd$ has a child $nd'$ through edge label $c$
11.         Add(new-valid, $nd', l_{curr}$)
12.       Let $l_{new}$ = Distance(new-valid, $nd$)
13.       Let $l = \min(l_{curr}, l_{new})$
14.       If $l + 1 \leq k$
15.         For each child $nd'$ of $nd$
16.           Add(new-valid, $nd', l + 1$)
17.     Swap new-valid and valid
18.     Clear new-valid
19.     Return leaf nodes reachable from nodes in valid

Procedure Add(buffer, $nd, l$)
 1. If $(nd, l')$ is already present in buffer
 2.   Replace $l'$ with $l$ if $l < l'$
 3. Else
 4.   Enqueue $nd$ in buffer
 5.   Set the distance of $nd$ in buffer to be $l$

Procedure Distance(buffer, $nd$)
 1. If $(nd, l)$ is present in buffer
 2.   Return $l$
 3. Else
 4.   Return $\infty$
```

a node valid if the edit distance is at most k. Further, since it is possible to reach a cell in multiple ways, we keep track of the minimum distance (Procedure *Add*).

Note that this procedure requires that we process the valid nodes in a top-down order where the parent of a node is processed before its children. The valid nodes are maintained in a buffer that has two components — a queue to maintain these nodes in top-down order, and a hash table to store the edit distance for each valid node. By the way Algorithm 5.1 operates, the top-down order is maintained without the need for an explicit sort.

The algorithm is initialized for the empty input string. This corresponds to marking the root and all nodes reachable within distance k from the root as valid (Steps 1-4). Finally, just as for the exact case, we return all leaf nodes reachable from the valid nodes. Note that the edit distance ordering can be ensured by sorting the node distances *before* retrieving the leaf nodes reachable from them.

Figure 6 illustrates how the algorithm operates on a database consisting of three strings — `Johnny, Josef` and `Bond`. The input string being entered is `Jonn`. The shaded nodes denote the valid nodes and the edit distances are shown beside them.

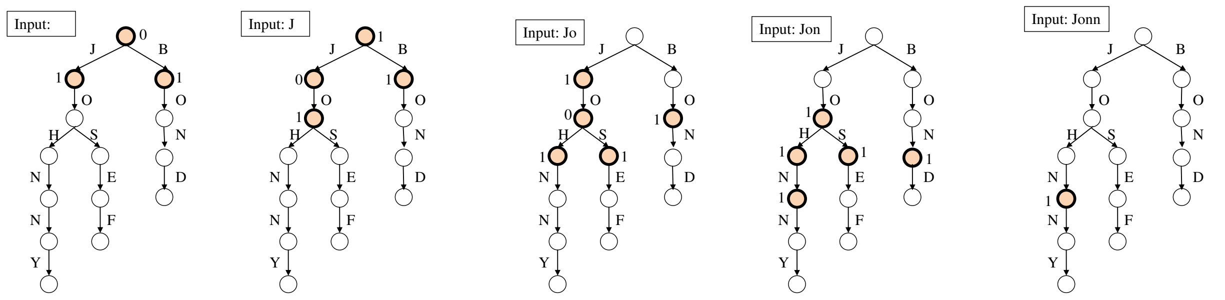

Figure 6: Illustrating Algorithm 5.1

Algorithm 5.2 Algorithm for Buffered k-Extension Strategy

```
Input:  String $s_1$ input incrementally with new
        character being appended at each step, edit threshold $k$ and
        transition length $len$.
 1. Set current threshold = 0
 2. For each character $c$ of the input
 3.   if (length of $s_1$ is $len$)
        RefreshValidNodes($l$)
        Return leaf nodes as in Algorithm 5.1
 4.   else
 5.     Run Algorithm 5.1 with current threshold

Procedure RefreshValidNodes($l$)
 1. Match $s_1$ against pre-computed index
 2. Mark nodes corresponding to output as valid
 3. Set current threshold to $k$
```

5.2 Buffered k-Extension Strategy

Efficiency-wise, one of the problems with the full-extension strategy above is that the number of valid nodes can be large. For instance, when the input string is empty, all nodes that are within distance k of the root of the trie are deemed valid.

We empirically study how the number of valid nodes changes as we progress in the lookup string. We work with an address data set consisting of 100 thousand strings (see Section 6 for more details of the experimental set up). We select lookup strings at random from the same database and compute the average number of valid nodes at any given lookup length, for various edit distance thresholds. The resulting plot is shown in Figure 7. The X-axis shows the length of the lookup string and the Y-axis plots the number of valid nodes on a logarithmic scale. As we can see, the number of valid nodes rises sharply with the edit distance threshold reaching a maximum of close to 25% of the number of strings in the data for $k = 4$, but also drops rather quickly once some initial portion of the string has been processed. This sharp increase in the number of valid nodes leads to a significant increase in execution time.

In addition to the utility argument presented in Section 2, this observation further motivates us to consider the *buffered* autocompletion strategy. In order to use Algorithm 5.1 to support this strategy, we need a way to determine the set of valid nodes at the transition length. While we could accomplish this by performing the traversal in Algorithm 5.1, this would perform poorly as explained above.

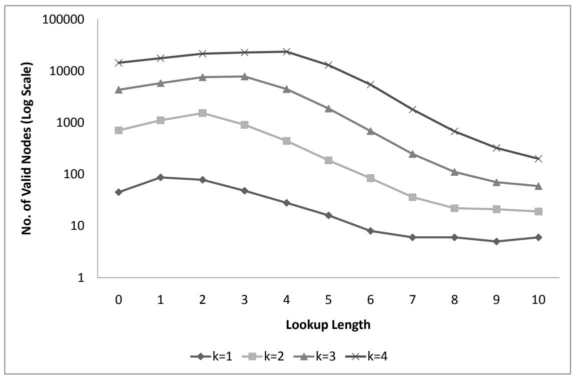

Figure 7: Valid Nodes (log scale) Vs. Lookup Lengths and Edit Distance Thresholds

We therefore maintain a separate index and invoke an offline edit distance matching algorithm at the transition length. The fact that the transition length is small can be used to *pre-compute* all edit distance matches. We reduce the alphabet size by hashing all characters to a small number of bits. Note that in this process, the edit distance between two strings can only decrease. In this hashed space, the number of strings of a small length is not very large. For instance, if we hash all characters to 4 bits and consider a transition length of 5, the number of possible hashed strings is 1 million. For each of these, we pre-compute all distance k neighbors from T. Note that at lookup time, we need to verify the strings returned from pre-computation to check whether they are within edit distance k in the original alphabet space (this can be achieved via the dynamic programming algorithm described in Section 3.3.2).

As can be seen from Figure 7, the number of valid states drops sharply as the transition length increases. By setting an appropriate transition length, we can utilize pre-computation to overcome the problem with short strings. This algorithm is outlined in Figure 5.2. Indeed, we provide a formal basis for this intuition in Section 5.3.

5.3 Analysis

We now formally analyze the trie-based algorithms proposed above. We first analyze them from the point of view of autocompletion, and then from the point of view of offline edit distance matching. Owing to lack of space, all proofs are omitted.

5.3.1 Autocompletion

For the Full k-Extension strategy, if we are required to output all k-extensions at each step, then Algorithm 5.1 in a sense does the minimal amount of work — the number of valid nodes at each point is at most $2k+1$ times the number of strings that are returned in this step. Algorithm 5.1 transitions from one set of valid nodes to the next in time linear in the number of such nodes. Suppose that the eventual lookup string s consists of m characters. Then, we perform autocompletion m times. Let the total output size of these completions across all m steps be $ACOutput(s)$. We also count the running time of Algorithm 5.1 over all m character appends.

Lemma 1. *Algorithm 5.1 runs in time $O(k \cdot ACOutput(s))$.*

Note that this is the worst case. In practice, the number of valid states is likely to be much smaller since the trie compresses common prefixes.

For the *Buffered* strategy, a similar analysis holds after the transition length. At the transition length, since the output is pre-computed in the hashed space, there is additional work in verifying the edit distance condition in the original alphabet space. We therefore need to analyze the probability of false positives produced after hashing.

Let the original alphabet size be σ and the size of the hashed alphabet be γ. We can show the following result assuming a perfect hash function h. Given string s, we refer to its hash image as $h(s)$.

Lemma 2. *Suppose that $ed(s_1, s_2) > c \cdot k$ and that both strings consist of unique characters from the alphabet Σ. Then the probability that $ed(h(s_1), h(s_2)) \leq k$ is at most $(\frac{1}{\gamma})^{\frac{(c-1)k}{6}}$.*

This result can be generalized to the case where each character appears at most a bounded number of times.

5.3.2 Edit Distance Matching

Observe that by construction, both Algorithm 5.1 and Algorithm 5.2 are algorithms for edit distance matching. More precisely:

Property 3. *For a lookup string s, consider the set of active nodes after all characters of s have been consumed that also correspond to full strings in T. These full strings are exactly the set of strings within edit distance k of s.*

We also analyze the running time of this algorithm viewed as an edit distance matching algorithm. The cost of Algorithm 5.1 is proportional to the total number of nodes in the trie that are considered valid in some iteration. At each step, the valid nodes are precisely those that are within edit distnace k of the string typed thus far. Based on this observation, we can show that:

Lemma 3. *Algorithm 5.1 adapted for edit distance matching of a string of length m runs in time $O(m \cdot \min((m+1)^k(\sigma+1)^k, |T|))$.*

The question arises whether the buffered strategy can be shown to be formally better. Assuming that the transition length $b << m$, the length of the entire lookup string, as is the case in practice, the worst case cost stays the same, which raises the question whether the running time is better on average. We perform an analysis for the case of a constant sized alphabet, since this assumption holds once the characters are hashed. (As noted above, the scope for an excessive number of hash collisions is low.) We obtain the following result. Here, $Output(s)$ denotes the output of edit distance matching for string s.

Lemma 4. *Suppose we are performing edit distance matching for a constant edit distance k. Suppose that the strings in T are generated as follows. We fix a skeleton of table T where for each string, its length is fixed. We then generate each string by choosing each character independently uniformly at random from an alphabet of constant size $\gamma > 1$.*

Consider Algorithm 5.2 with transition length $O(\log |T|)$. The lookup time for string s of length m is $O(m + Output(s))$ with high probability.

We note that the above analysis merely merely provides a principled basis for our approach. In particular, we do not necessarily choose the transition length in accordance with the above result.

5.4 Top-l Semantics

As noted in Section 2 we can order the extensions even in the case of exact autocompletion via a static score associated with each string in T. This ordering can be used to return only the top-l extensions. This further helps in keeping the output size small.

In the presence of string edits, we similarly have the option of returning only the top-l extensions sorted by a ranking function $Score$ which combines the edit distance with the static score of a string in a monotonic fashion (as described in Section 3.2). Finding all extensions and then sorting them by their score can be inefficient since the number of extensions can be large whereas we only want to return the top-l.

We address this by pre-computing the top-l completions by static score for each node in the trie. For exact autocompletion, we use this to simply read off the top-l extensions from the current node in the trie.

When allowing for edits, we have multiple valid nodes in the trie. Therefore we merge the sorted lists corresponding to each valid node to obtain the overall top-l. Since the overall score is monotonic in the static score and the edit distance, we can invoke any of the previously published algorithms such as the Threshold Algorithm [6] that perform early termination when computing the top-l results.

We can extend this idea to the case of the q-gram based algorithms by viewing the q-gram technique as a way of obtaining the valid nodes in the trie at each step.

6. EXPERIMENTS

In this Section, we discuss our empirical study of error-tolerant autocompletion. The goals of our study are:

1. To measure the effectiveness of incorporating error tolerance in autocompletion by studying the number of key strokes saved when typing.

2. To compare the performance of the *Full* and *Buffered* strategies for error-tolerant autocompletion.

3. To compare the performance of trie-based autocompletion with the q-gram based algorithms.

4. To compare the performance of online autocompletion against offline edit distance matching algorithms.

5. To study the impact of data size on autocompletion performance.

We first discuss some details of our implementation and the experimental setting.

6.1 Experimental Setting

6.1.1 Implementation Details

Our prototype is implemented in managed code (C#). All of our implementation is in-memory. The trie is compressed in order to reduce its memory footprint. Specifically, we identify "chains" in the trie where no node has more than one child. These chains are compressed by creating only one node per chain that represents the entire chain. This technique yields significant compression since the trie of data sets we encounter in practice have a large number of chains. We store a separate table of the top-l completions at each node in a separate (in-memory) table. As noted in Section 5.1, the set of active nodes is maintained using a queue and a hash table. For the buffered autocompletion strategy, we consider various transition lengths up to 7 characters setting the hashed alphabet size appropriately so that the total number of strings for which we perform pre-computation is at most 2 million. We pick the best transition length. We refer to the trie-based algorithms as *Trie* in this section.

For the q-gram based algorithm, we study three of the state-of-the-art signature schemes — Locality Sensitive Hashing (LSH) [7], PartEnum (PE) [1] and the improved Prefix Filtering (PF) technique proposed recently [21]. LSH is an *approximate* technique that can miss some results with a tunable probability. We configure LSH so that this failure probability is 5%. In contrast, both PE and PF are *accurate* — they are guaranteed not to miss any results. For each of these signature schemes, we do not perform any set-based post-filtering once the signatures are looked up. This is because we require an edit distance check that incorporates prefixes and doing a set-based verification for all prefixes would be wasteful. Our implementation of the q-gram based indexes is also in-memory. Each of these signature schemes is parameterized. We manually choose these settings to obtain the best performance. We report the best results over all accurate and approximate signature schemes. We refer to these results as *QGram Accurate* and *QGram Approx.* respectively. Since accurate schemes are also approximate, the results for *QGram Approx.* can never be worse than those for *QGram Accurate.*

6.1.2 Data

We report results over two real-world data sets. The first is a collection of author names from DBLP [5] (*DBLP*). This data set has 550000 names and the average string length is 14. The second data set we use is a collection of proprietary addresses (*Address*) that has a total of 1.7 million addresses of average length 36. We use a subset of 500k addresses for most experiments.

Our experiments are run on a workstation running Microsoft Windows Server 2003 with a 3.6GHz Intel Xeon processor and 8GB of main memory. Figure 8 reports the memory consumption of various techniques in MB. For the *Trie* algorithms, the memory consumed counts the space for pre-computation. For the QGram techniques, we report the minimum memory consumed among all relevant signature schemes (for their best performing parameter choices). We find that the memory consumption of *Trie* is significantly better than either of the q-gram approaches. The space consumption of the online q-gram approaches is significantly higher because the number of signatures generated is much larger — we generate signatures not only for a string but also many of its prefixes. Thus, we find that the space consumption is worse when the strings are longer as is the case with addresses. As a point of comparison, we also report the memory consumption of offline techniques. We find that the memory consumption of *Trie* is comparable to the space consumed by the offline indexes for the *Address* data and substantially better for the *DBLP* data set.

	Trie	QGram (Approx.)	QGram (Acc.)	QGram offline (Approx.)	QGram offline (Acc.)
DBLP	121	777	777	248	248
Address	173	2378	2608	162	356

Figure 8: Memory Consumption (MB)

6.2 Effectiveness of Error Tolerance in Autocompletion

In this experiment, we empirically study how critical error tolerance is when performing autocompletion. Our methodology is as follows. We fix a data set and a set of pairs of strings obtained by performing a separate "fuzzy" self-join on the data set at a small distance threshold. This represents pairs that that are different representations of the same underlying (author or address) entity.

For each of these pairs (s_1, s_2), we fix s_1 as the lookup string and measure the total number of key strokes needed to find s_2. In the absence of autocompletion, this number is the number of characters in s_1. In the presence of autocompletion, we count the number of characters of s_1 need to be entered before s_2 appears in the top 10 results of autocompletion, and the number of key strokes needed to navigate the top 10 list before finding s_2. The difference between the above yields the number of key strokes *saved* due to autocompletion. We measure this both for exact autocompletion and error-tolerant autocompletion for various settings of the autocompletion edit distance threshold, denoted k.

Data Set	k=0	k=1	k=2
DBLP	2.84	3.53	4.39
Address	11.65	16.24	23.98

Figure 9: Key Strokes Saved Per Lookup String

Figure 9 reports the results of our study. We report the average number of key strokes saved per lookup string for various values of error threshold k. On the *DBLP* data set, we observe that in contrast with exact autocompletion ($k = 0$), 54.58% additional key strokes are saved when autocompletion is performed with threshold 2 for the *DBLP* data set. For the *Address* data set that consists of larger strings, more than 100% additional key strokes are saved. Even for $k = 1$, we find that close to 24% additional keystrokes are saved for *DBLP* and the corresponding number for *Address* is close to 39%. This indicates that tolerating errors when performing autocompletion significantly reduces the user's typing.

6.3 Autocompletion Strategy

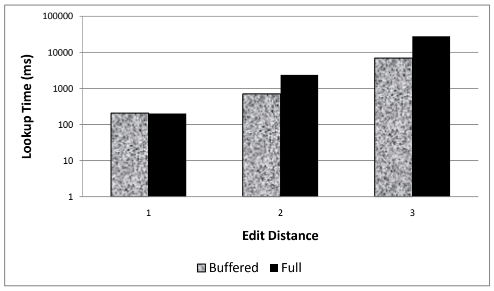

Figure 10: Full Vs Buffered (log scale): DBLP

We first compare the *Full* and *Buffered* strategies. The transition length used is 7 with all characters hashed to 3 bits. In order to make the comparison meaningful, we use the autocompletion algorithms to perform edit distance matching. We perform the comparison only for the Trie approach — the results are similar even for q-gram based approaches. Figure 10 reports the result of our study over the DBLP data set (the results look similar for the *Address* data). The X-axis shows the edit distance threshold used and the Y-axis reports the execution time for 1000 lookups in milli-seconds, on a log scale. We find that the full strategy is at least 5 times slower than the buffered strategy for $k = 2$ and $k = 3$.

Recall from Section 5.2 that the primary reason for this disparity is that the number of valid nodes in the initial stages of autocompletion is extremely large for the *Full* strategy. This number sharply drops once a few characters of the input have been processed. Thus, the initial overhead incurred by the *Full* strategy is saved. Henceforth in this section, we do not consider the *Full* strategy.

6.4 Efficiency: Trie Vs NGram

The rest of this section studies the overall efficiency of our autocompletion. We first compare the *Trie* and q-gram based autocompletion algorithms for the same (buffered) strategy. Figure 11 shows the comparison over both the data sets. We vary the edit threshold plotted on the X-axis and compute the lookup time aggregated over 1000 lookups, plotted on the Y-axis on a log scale. We find that the trie-based Algorithm 5.2 significantly out-performs the ngram based algorithm, irrespective of the signature scheme chosen often by an order of magnitude. The primary reason for this is that Algorithm 5.2 pretty much navigates at each step from the previous answer set to the current answer set. Instead, the q-gram approach needs to access the rid-list corresponding to several q-grams. Over the course of a long string such as an address string, this can be quite expensive. Since the trie-based Algorithm 5.2 for the *Buffered* strategy is the most efficient, we only report for this algorithm in the rest of this section.

6.5 Per-Character Response Time

Recall that in order to look instantaneous to the user, the response time must be less than 100ms. In a client-server setting, this 100ms bound includes not only the autocompletion time, but other overheads such as the communication overheads. Thus, it is desirable to make the autocompletion cost itself minimal. In this section, we report the average per-character execution time of the Trie-based Algorithm 5.2 for the *Buffered* strategy. We find the per-character response times to be 0.39ms for the *DBLP* data set and 0.32ms for the *Address* data set even when the edit distance threshold is 3.

6.6 Online Vs Offline

We next compare the performance of our algorithm to offline edit distance matching algorithms. As discussed in Section 2.3, we expect the offline algorithms to perform better and our goal is to measure the difference in performance and contrast this difference with the case of exact matching. Accordingly, we also measure the performance of trie-based exact autocompletion by setting the edit distance threshold to 0. We use hashing as our method of performing offline exact matching.

Figure 12 plots the results. The figure on the left reports the results for the *Address* data and the one on the right, for the *DBLP* data. The X-axis shows the edit distance threshold and the Y-axis plots the execution time (in milli-seconds) for 1000 lookups on a log scale. For this experiment, in addition to the q-gram based offline matching algorithms, we also consider previously proposed offline trie-based pattern matching algorithms [18]. As noted above, we only report the best running times over all offline algorithms we consider.

For the *Address* data, as expected, the offline algorithms out-perform the online algorithm described in this paper. The performance gap is close to an order of magnitude. However, this gap is worse for exact matching where the online approach is more than two orders of magnitude more expensive.

For the *DBLP* data, the *Trie* algorithm is much more competitive with the offline algorithms. When compared to the accurate offline algorithms, for $k = 1$, the online approach is about twice as worse, whereas for $k = 2, 3$, the gaps drop to 25% and 12% respectively. Only the *QGram Approx.* technique performs substantially better for $k = 3$, but this comes with the tradeoff that 5% of the results can be missed. The main reason for this smaller gap in the DBLP data is that the strings are much shorter compared to addresses. For the q-gram based approaches, the set-similarity threshold for the underlying set-similarity match is much lower for the same value of k, leading to a larger execution time for all of the signature schemes. In contrast, the gap between exact offline matching and exact autocompletion remains at more than two orders of magnitude for the *DBLP* data.

We recall that our goal as discussed in Section 2.3 is that the online vs. offline performance gap be comparable to the case of exact matching. We can see from the above discussion that we meet this goal on the data sets we study.

6.7 Overhead of Error Tolerance

We can use the results in Figure 12 to also study the overhead of error tolerance in autocompletion. As expected, the greater the magnitude of error we wish to tolerate, the

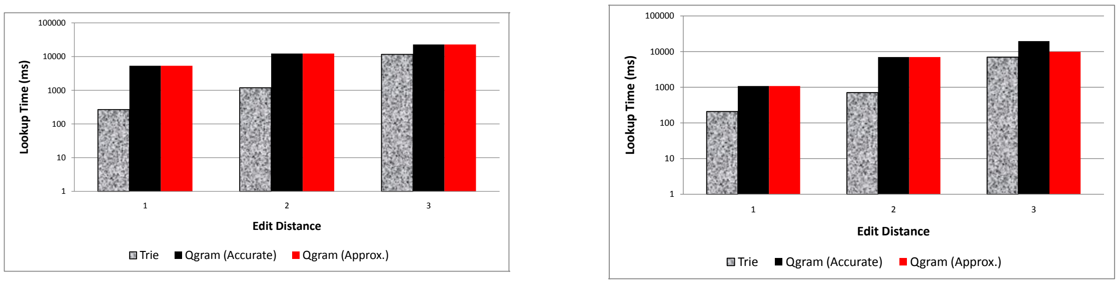

Figure 11: Efficiency of Trie Vs NGram: (a)Address, (b)DBLP

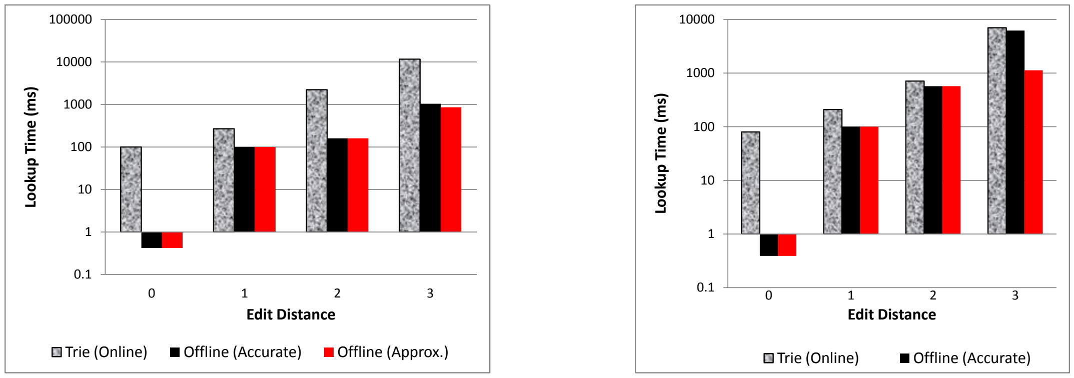

Figure 12: Autocompletion: Online Vs Offline (Log Scale): (a)Address, (b)DBLP

greater is the price we pay. Performing autocompletion with $k = 1$ is a factor of 2 times worse than exact autocompletion ($k = 0$), whereas the gap between $k = 3$ and $k = 0$ (exact autocompletion) is close to two orders of magnitude for both the data sets. This is not very surprising. In the offline world, there is an even worse price to be paid for error tolerance — observe that the gap between edit distance 3 and exact matching is close to four orders of magnitude.

6.8 Effect of Data Size

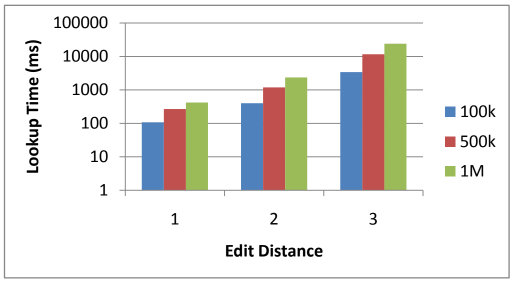

Figure 13: Increasing Data Size: Address

We next study the effect of data size by choosing subsets of the address data of size 100k, 500k and 1 Million records. Figure 13 shows the results. For each data set, we compute the running time for edit thresholds 1 to 3. The X-axis shows varies the edit distance threshold. For each value of the threshold, we report the lookup time for the different subsets of the address data. Overall, we see that the execution time increases linearly with the data size. This is consistent with the analysis captured in Lemma 3 and Lemma 4.

7. RELATED WORK

Autocompletion is a widely used mechanism to find information quickly. It has been adopted in a variety of applications including program editors such as Visual Studio, command shells such as the Unix Shell, search engines such as Google, Live and Yahoo, and desktop search. More recently, it is also gaining popularity for mobile devices such as cell phones. These applications often work over a predefined dictionary of strings over which autocompletion is performed. As noted in Section 1, despite the availability of variants of error-tolerant autocompletion in a limited number of these products, little is known about their underlying algorithms.

There is also a vast body of work on predictive autocompletion [4, 8, 16] where the idea is to use information retrieval techniques, language models and learning to suggest potential completions. While error tolerance is also applicable in these approaches, our paper focuses on the case where autocompletion is performed on the basis of matches in a pre-specified table.

There has been recent work in the research community on autocompletion. Bast and Weber [2] propose an autocompletion method where we have an underlying document corpus and the goal is to return word-level completions such that the resulting multi-word query has non-empty results. Nandi and Jagadish [16] propose a phrase prediction algorithm as a part of the MUSIQ project [11], targeting scenarios such as email composition and introduce the concept of a significant phrase for this purpose. To the best of our knowledge, none of the above bodies of work on autocompletion focus on systematically studying error-tolerant autocompletion in a database lookup scenario.

A functionality that is closely related to autocompletion and is also widely available is *autocorrection* where at the end of a string typed in, the system suggests corrections by matching it against a dictionary. This functionality is present in text editors such as Microsoft Word and is also offered recently by search engines. The key difference from autocompletion is that autocorrection and spelling suggestions are offered at the end of the input string, presumably through an offline error-tolerant match against a pre-defined dictionary. In contrast, autocompletion is online.

Another body of work that is closely related to this paper is the prior work on computing edit distance matches of a (small) pattern string P in a (large) piece of text T [10, 13, 17, 18, 19, 20]. The goal is to find all substrings of T that are within a small edit distance of P. These algorithms can be divided into two parts — algorithms where there is no preprocessing of the text T [17], and algorithms where we are allowed to create an index over T to match the pattern string faster [18]. The body of work that closely relates to this paper clearly is the latter class of algorithms. Some of these algorithms proceed over suffix tries/trees which is similar to our technique, but their basic approach is not online in the sense required by autocompletion. That is, they do not consume the pattern string character by character. In contrast, our algorithms process the lookup string incrementally.

Viewed purely as an edit distance matching algorithm the running time of our algorithm is $O(m \cdot \min((m+1)^k(\sigma+1)^k, N))$ where m is the size of the pattern string, k is the edit distance threshold and N is the size of the text string. This is identical to the worst case analysis for the algorithms proposed in [18]. There is recent work that improves on this worst case. Specifically, Maaí and Nowak [13] recently proposed the first linear time edit distance matching algorithm. However, this algorithm is not practical since the size of their data structure can be unbounded in terms of the input table size in the worst case.

Finally, there is also the body of work on edit distance matching using q-grams [1, 3, 9, 21]. These algorithms are also "offline" in the sense they do not consume the lookup string character by character. We incorporate these techniques in our paper for autocompletion (Section 4).

8. CONCLUSIONS

We considered the problem of performing autocompletion when a record is being looked up against a database table. By viewing autocompletion as an "online" lookup, we argued that just as "offline" lookup needs to be error-tolerant, so does autocompletion. We took a first step in this paper in designing an error-tolerant autocompletion based on edit distance as our similarity function. We proposed algorithms for autocompletion using both on q-gram based techniques and trie traversal techniques coupled with pre-computation for short strings. Our empirical study indicated both the utility of error-tolerant autocompletion and the fact that we can perform error-tolerant autocompletion with performance trade offs similar to the case of exact autocompletion.

Autocompletion is used in a wide variety of applications depending on the nature of which, other issues need to be addressed such as performing autocompletion in a client-server setting. This paper however focuses on the algorithmic aspects of error-tolerant autocompletion which are relevant regardless of the specific application. The issue of performing error-tolerant autocompletion for other similarity functions also needs to be addressed. We hope to tackle these questions in future work.

9. REFERENCES

[1] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set similarity joins. In *VLDB*, 2006.

[2] H. Bast and I. Weber. Type Less, Find More: Fast Autocompletion Search with a Succinct Index. In *SIGIR*, 2006.

[3] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, 2006.

[4] J. Darragh, I. Witten, and M. James. The reactive keyboard: a predictive typing aid. *Computer*, 11(23):41–49, 1990.

[5] Dblp. http://dblp.uni-trier.de/.

[6] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[7] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[8] K. Grabski and T. Scheffer. Sentence completion. In *27th International Conference on Research and Developement in Information Retrieval*, 2004.

[9] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, 2001.

[10] T. N. D. Huynh, W. K. Hon, T. W. Lam, and W. K. Sung. Approximate string matching using compressed suffix arrays. *Theoretical Computer Science*, 2006.

[11] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *SIGMOD*, 2007.

[12] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD*, 2006.

[13] M. G. Maaí and J. Nowak. Text indexing with errors. *Journal of Discrete Algorithms*, 5(4):662–681, 2007.

[14] R. Miller. Response time in man-computer conversational transactions. In *Proceedings of the AFIPS Fall Joint Computer Conference*, 1968.

[15] A. Nandi and H. V. Jagadish. Assisted querying using instant-response interfaces. In *SIGMOD Conference*, 2007.

[16] A. Nandi and H. V. Jagadish. Effective phrase prediction. In *VLDB*, 2007.

[17] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[18] G. Navarro, R. A. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Eng. Bull.*, 24(4):19–27, 2001.

[19] G. Navarro, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Faster approximate string matching over compressed text. In *Data Compression Conference*, pages 459–468, 2001.

[20] E. Ukkonen. Approximate string-matching over suffix trees. In *Combinatorial Pattern Matching*, pages 228–242, 1993.

[21] C. Xiao, W. Wang, and X. Lin. Ed-Join: An Efficient Algorithm for Similarity Joins With Edit Distance Constraints. In *VLDB*, 2008.