ED-JOIN: AN EFFICIENT ALGORITHM FOR SIMILARITY JOINS WITH EDIT DISTANCE CONSTRAINTS

By
THAI NGOC THUY HUONG – U065131H
Department of Computer Science
School of Computing
National University of Singapore

2008/2009
ED-JOIN: AN EFFICIENT ALGORITHM FOR SIMILARITY JOINS WITH EDIT DISTANCE CONSTRAINTS

By
THAI NGOC THUY HUONG – U065131H
Department of Computer Science
School of Computing
National University of Singapore

2008/2009

Project No:
Advisor: Associate Professor CHAN, CHEE YONG

Deliverables:
Report: 1 Volume
**ABSTRACT**

Similarity join is a fundamental operation in many application areas, such as data integration and cleaning, bioinformatics, and pattern recognition. In this project, we implement an efficient algorithm for similarity join with edit distance constraints. Current approaches are mainly that the edit distance constraint is converted to a weaker constraint on number of matching q-grams between pair of strings. In our project, we exploit a novel perspective of investigating mismatching q-gram. We derive two new edit distance lower bounds by analyzing the locations and contents of mismatching q-grams. A new algorithm, Ed-Join, is proposed that exploits the new mismatch-based filtering methods; it achieves substantial reduction of the candidate sizes and hence saves computation time.

**Keywords:**
Similarity join, location-based filtering, content-based filtering, mismatching q-gram

**Implementation Software and Hardware**
Java JDK 6, Eclipse 3.2, Notepad++
ACKNOWLEDGEMENT

We want to show our deep gratitude to Ass Prof. CHAN, Chee Yong for his guidance and valuable advices.
TABLE OF CONTENTS

Title i
Abstract ii
Acknowledgement iii
List of Figures iv

1 Introduction 1
  1.1 Introduction 1
  1.2 Project Objectives 2

2 Preliminaries 3
  2.1 Problem Definition and Background 3
  2.2 q-gram-based Filtering 4
  2.3 Prefix Filtering 4
  2.4 The All-Pairs-Ed Algorithm 5

3 Location-based Mismatch Filtering 6
  3.1 Overview of the Ed-join Algorithm 6
  3.2 Location-based Mismatch Filtering 6
  3.3 Minimum Prefix for Edit Similarity Join 7
  3.4 The Ed-Join Algorithm 8

4 Content-based Mismatch Filtering 8
  4.1 Content-based Mismatch Filtering 8
  4.2 Tightening Lower Bound 11
  4.3 The Verification Algorithm based on Multiple Filters 12
  4.4 The exact Edit Distance Algorithm 15

5 Implementation 15
  5.1 Data Structure 16
  5.2 Important Classes and method 16

6 Evaluate the correctness 17
  6.1 Prefix filtering 17
  6.2 Simple Adjustment 18
6.3 Complicated Adjustment

7 Testing Methodology

7.1 Manual Test

7.2 Auto generated Testing

7.3 Experiment Setup

7.4 The great difference between naïve approach and novel approach in processing

7.5 Effect of q-gram length

7.6 Large edit error thresholds

7.7 Experiments of Complex Modified Ed_Join and Simple Modified Ed_Join

8 Conclusions

8.1 Achievements

8.2 Conclusions

References

Appendix A – Program Listing

Appendix B – How to Use the Program
1 INTRODUCTION

1.1 Introduction
With the wide availability of data sources on the Web and the increasing demands for data integration within enterprises, similarity join has become an essential procedure to provide an effective and efficient way to correlate data together. Similarity join between two sets of objects returns pairs of objects from each set such that similarity values between the pairs are above a given threshold. Due to its importance, similarity join has been studied in many areas, such as data integration and cleaning, bioinformatics, and pattern recognition. Similarity join is also adopted in the industry solutions. For example, Google adopts both approximate and exact similarity join for near duplicate Web page detection [18], query log mining, and collaborative filtering [3], etc.

In the paper, we study similarity joins with edit distance thresholds or edit similarity join [12]. Edit distance measures the minimum number of edit operations (insertion, deletion, and substitution) to transform one string to another. Edit distance has two distinctive advantages over alternative distance or similarity measure: (a) it reflects the ordering of tokens in the string; and (b) it allows non-trivial alignment. These properties make edit distance a good measure in many application domains, e.g., to capture typographical errors for text documents, and to capture similarities for Homologous proteins or genes.

A naïve algorithm for similarity join with edit distance thresholds will evaluate the edit distance for every pair of strings and incur an expensive running cost $O(N^2 \cdot n^2)$, where $N$ is the number of strings and $n$ is the length of strings.

Current state-of-the-art approaches to process edit similarity join are mainly based on a filter-and-verify approach, operating on the q-gram representation of strings [14]. The q-gram representation of a string is the set of substrings obtained by sliding a window of length $q$ over the string. If there are only few edit errors between two strings, the majority of the q-grams in one string will be preserved and we should be able to find them in approximately the same locations in the other string. Q-grams that are preserved are
called matching q-grams. Several filtering conditions regarding the total number and locations of the matching q-grams and the lengths of the strings were developed [14]. These filtering conditions are only necessary conditions for the edit distance threshold, and therefore, string pairs that survive all the filters still need to be verified by the edit distance calculation. Hence, the efficiency of this filtering-based approach critically depends on the pruning power of the filters. A recent progress is the introduction of prefix filtering [30, 12, 3]. When applied to the edit similarity join, it usually reduces the candidate size significantly and hence speeds up the computation.

In our approach, we not only make use of prefix filtering but also take a novel perspective by studying mismatching q-grams. As traditional filtering methods base on the count of matching q-grams, they have high computational cost as they have to access all the q-grams before rejecting a candidate pair. We find that q-grams that cannot be matched also provide valuable information on the similarity of strings. Hence, we derive two novel lower bounds to edit distance by analyzing mismatching q-grams. Two filtering methods, location-based mismatch filtering and content-based mismatch filtering, are developed that usually reduce the size of the candidates substantially compared with previous methods [14, 12, 3]. We also show how the additional filters are complementary to existing ones and that they are all integrated into a new algorithm, Ed-Join, to attain the maximal pruning power.

1.2 Project Objectives

Java programming language is popular today. It is applied in many areas such as Web, Game Development and Industry, etc. In this project, we study thoroughly the two new novel filtering methods, location-based filtering and content-based filtering, of similarity join with edit distance threshold. We will try to implement this new similarity join in java programming language (it is originally intended to be implemented in C++). While implementing and study the suggesting algorithms, we also evaluate the efficient, the correctness of the algorithms and look for possible improvements.
In this approach to similarity join with edit distance constraints, selecting q-gram length and edit distance constraint $\tau$ have great impact on filtering candidate strings. Hence, we also setup experiments to study $q$ and $\tau$.

2 PRELIMINARIES

2.1 Problem Definition and Background:

Let $\sum$ be a finite alphabet of symbols $\alpha_i$ ($1 \leq i \leq |\sum|$). A string $s$ is an ordered array of symbols drawn from $\sum$. The length of string $s$ is denoted as $|s|$. Each string $s$ is also assigned an identifier $s$.id. All input string sets are assumed to be in increasing order of string length. $ed(x, y)$ denotes the edit distance between strings $x$ and $y$, which measures the minimum number of edit operations (insertion, deletion, and substitution) to transform one string to another (and vice versa). It can be computed in $O(n^2)$ time and $O(n)$ space using standard dynamic programming [32].

Given two sets of strings $R$ and $S$, a similarity join with edit distance threshold $\tau$ returns pairs of strings from each set, such that their edit distance is no larger than $\tau$, i.e., \{ <r, s>|$ed(r, s) \leq \tau$, $r \in R$, $s \in S$ \}. A q-gram is a contiguous substring of length $q$; and its starting position in a string is called its position or location. A positional q-gram is a q-gram together with its position. Let $w$ be a q-gram and $df(w)$ be the number of strings containing $w$. The inverse document frequency of $w$, $idf(w)$, is defined as $1/df(w)$.

We extract all the positional q-grams of a string and order them by decreasing order of their idf values and increasing order of their locations. We call the sorted array the q-gram array of the string. Sorting positional q-grams in this order is a good heuristic to speeding up similarity joins later.

A string $s$ can generate $l = |s| - q + 1$ q-grams. Given a q-gram array $x$, $str(x)$ denotes its corresponding string. The i-th positional q-gram in $x$ is denoted as $x[i]$; its q-gram and location are denoted as $x[i].$token and $x[i].loc$, respectively. The k-prefix of $x$ is its first k entries, i.e., $x[1..k]$. An inverted index for q-grams is a data structure that maps a q-
gram w to an array Iw containing entries in the form of (id, loc), where id identifies the string that contains w and loc is the starting location of w in the string identified by id. The entries in Iw are sorted in the increasing order of id and loc.

Example 1: Consider the string s = abaabab. Let q = 2, it has l = 6 positional q-grams: (ab, 0), (ba, 1), (aa, 2), (ab, 3), (ba, 4), and (ab, 5). And if idf(aa) > idf(ab) > idf(ba), the q-gram array of s is

| (aa,3) | (ab,1) | (ab,4) | (ab,6) | (ba,2) | (ba,5) |

2.2 Q-gram based Filtering:
Current approaches for edit similarity joins are to relax the edit distance constraints to a weaker constraint on the number of matching q-grams. Two q-grams match if they have the same token and their locations are within the edit distance threshold τ. Specifically, three filters were proposed as follows: if two strings s and t are within edit distance τ, then

**Count Filtering** mandates that s and t must share at least

$$LB_{s,t} = (\max(|s|, |t|) − q + 1) − q \cdot \tau$$

common q-grams.

**Position Filtering** mandates that s and t must share at least $LB_{s,t}$ matching positional q-grams.

**Length Filtering** mandates that $|s| - |t| \leq \tau$.

2.3 Prefix Filtering:
In q-gram-based Filtering, for candidate pairs that share $LB_{s,t}$ matching q-grams, we cannot discard them unless all their q-grams have been accessed and compared. This is inefficient. Hence, a prefix-based filtering is proposed to quickly filter out candidate pairs that are guaranteed not to meet the $LB_{s,t}$ threshold.

**Lemma 1 (Prefix Filtering).** Let x and y be two q-gram arrays and $ed(str(x), str(y)) \leq \tau$. Then the $(q \cdot \tau + 1)$-prefix of x and the $(q \cdot \tau + 1)$-prefix of y must have at least one matching q-gram.
2.4 The All-Pairs-Ed Algorithm:

All-Pairs-Ed Algorithm is the traditional approach for similarity join. It iterates through each q-gram array \( x \); for each q-gram \( w \) in the \((q \cdot \tau + 1)\)-prefix of \( x \), it probes the inverted index to find other q-gram arrays \( y \) that contain matching q-grams to \( w \). Afterwards, \( x \) and all its candidates will be further checked by the Verify algorithm. Within Verify, count and positional filtering will be applied to every candidate pair first. Only those that pass both filters will be further checked by performing the expensive edit distance calculation.
3 LOCATION-BASED MISMATCH FILTERING

3.1 Overview of the Ed-join Algorithm

Most existing algorithms for edit similarity join are based on the accumulation and counting of matching q-grams which has to access and process all the q-grams of strings that share a common q-gram in their respective prefixes.

In this project, we analyze the mismatching q-gram to speed up the join processing. The Ed-Join algorithm employs two novel filtering techniques obtained from analyzing the locations and the contents of mismatching q-grams. Like All-Pairs-Ed, our Ed-join also starts with the candidate-generation, continues with verification and finally employs the expensive exact edit distance calculation for the remaining candidates.

3.2 Location-based Mismatch Filtering

We design a greedy algorithm that finds the minimum number of operations that destroy all q-grams in a given set of q-grams $Q$ in $O(|Q|\log|Q|)$ time.

\[\text{Algorithm 1: All-Pairs-Ed (R, } \tau)\]

1. $S \leftarrow \emptyset$;
2. $I_i \leftarrow \emptyset$ $(1 \leq i \leq |U|)$;
3. for each $x \in R$ do
   4. $A \leftarrow$ empty map from id to boolean;
   5. $p_x \leftarrow q \cdot \tau + 1$;
   6. for $i = 1$ to $p_x$ do
      7. $w \leftarrow x[i].token$; $loc_x \leftarrow x[i].loc$;
      8. for each $(y, loc_y) \in I_w$ such that $|y| \geq |x| - \tau$ and $A[y]$ has not been initialized do
         9. if $|loc_x - loc_y| \leq \tau$ then
            10. $A[y] \leftarrow$ true; /* found a candidate */
      11. $I_w \leftarrow I_w \cup \{(x, loc_x)\}$; /* index the current prefix */
   12. Verify$(x, A)$;
13. return $S$
The algorithm is shown in Algorithm 2. It greedily selects the next unprocessed mismatch q-gram (Q[i]), makes an substitution edit operation at its last position (i.e., Q[i].loc + q – 1), and then removes all the subsequent q-grams that are destroyed by this substitution. The loop continues until all q-grams are destroyed.

**Algorithm 2: MinEditErrors (Q)**

1. Sort q-grams in Q in increasing order of locations if necessary;
2. cnt ← 0; loc ← 0;
3. for i = 1 to |Q| do
   4. if Q[i].loc > loc then
      5. cnt ← cnt + 1;
      6. loc ← Q[i].loc + q - 1;
4. return cnt

**PROPOSITION 1:** Algorithm 2 correctly solves the minimum edit error problem

**PROPOSITION 2:** (MONOTONICITY)

\[ \text{Min-err}(Q) \leq \text{Min-err}(Q'), \quad \forall Q \subseteq Q'. \]

**PROPOSITION 3:** Upper bound of \(|Q|/q \leq \text{min-err}(Q) \leq |Q|\)

**Lemma 2.** Let x and y be two q-gram arrays, and let Q be the set of mismatching q-grams from x to y, then \(\text{ed}(\text{str}(x), \text{str}(y)) \geq \text{min-err}(Qp), \quad \forall Qp \subseteq Q\).

Lemma 2 gives a new edit distance lower bound based on any subset of a string’s mismatching q-grams.

### 3.3 Minimum Prefix for Edit Similarity Join:

In lemma 1, prefix filtering requires each string to generate a fixed-length q·τ + 1 prefix and consider as candidates other strings whose prefix matches one of the q-grams in the prefix. This means that a string in a candidate pair can have up to q · τ mismatching q-grams in the prefixes. It is highly likely that the minimum edit errors of those mismatching q-grams already exceed τ and the candidate pairs are in fact disqualified with respect to the edit distance constraint due to Lemma 2.

Based on the monotonicity of the minimum edit errors (Proposition 2), we can further strengthen the filtering condition by reducing the prefixes to the minimum prefixes.
Intuitively, the minimum prefix is the shortest prefix of the q-gram array x such that if all
the q-grams in the minimum prefix are mismatched, it will incur at least $\tau + 1$ edit errors.
We call the length of such minimum prefix minimum prefix length.

If we let $u = q \cdot \tau + 1$, then the cost of a naïve algorithm for calculation minimum prefix
length will be $O(u^2)$. In algorithm 3 CalcPrefixLen, we make use of algorithm 2 and
binary search to reduce the cost to $O(u \log^2 u)$.

```
Algorithm 3: CalcPrefixLen (x)
1 left ← $\tau + 1$; right ← $q \cdot \tau + 1$;
2 while left < right do
3     mid ← (left + right)/2;
4     err ← MinEditErrors(x[1..mid]);
5     if err $\leq \tau$ then
6         left ← mid + 1;
7     else
8         right ← mid;
9 return left
```

**Lemma 3 (Location-based Mismatch Filtering).** Let the minimum prefix length for q-
gram arrays x and y be $l_x$ and $l_y$, respectively. If $ed(str(x), str(y)) \leq \tau$, x’s $l_x$-prefix and
y’s $l_y$-prefix must have at least one matching q-gram.

### 3.4 The Ed-Join Algorithm
This algorithm is similar to the All-Pairs-Ed algorithm with two important modifications:

1. A shorter minimum prefix length is used instead of the standard prefix length required by prefix filtering. We note that this optimization is critical in reducing candidate size, a measure that is highly correlated to the overall join performance, in addition to other obvious benefits (e.g., smaller inverted index).

2. A new implementation of the Verify algorithm that exploits another novel mismatch-based filtering. We will show the details in Section 4. The new filtering will effectively reduce the final number of candidate pairs to be verified by the expensive edit distance function.

4 CONTENT-BASED MISMATCH FILTERING

4.1 Content-based Mismatch Filtering

We define two types of edit errors. First, non-clustered edit errors are a set of edit errors such that no two of them are within distance $q$. The location-based mismatch filtering we have is well suited for detecting this type of errors. However, one weakness in location-
based mismatch filtering is that we assume the set of observed mismatching q-grams were caused by the minimum number of edit operations. It is possible that several edit errors actually occur within the same mismatching q-gram. We call this second type of errors clustered edit errors. And clustered edit errors occur fairly frequently in real datasets. For example, it is often the case that an entire word is deleted/inserted/substituted when editing text documents. In protein sequence alignment, it is well-known that a gap, i.e., several contiguous insertions or deletions of amino acid residues, is common and needs special treatment as it is likely to be due to one evolutionary event. Hence, we design the content-based mismatch filtering to detect clustered edit errors.

For content-based mismatch filtering, our idea is to select a probing window and look into the contents of both strings within the probing window; the content difference in the probing windows, when measured by an appropriate distance measure, will lower bound the edit distance of the pair.

A probing window \( w \) is an interval \([w.s \ldots w.e]\). The content of \( w \) on a string \( s \) is the substring between location \( w.s \) and \( w.e \), i.e., \( s[w.s \ldots w.e] \). Given a (sub-) string \( t \), its frequency histogram \( H_t \) is a vector of size \( |\Sigma| \), where \( H_t[i] \) records the number of occurrences of a symbol \( \alpha_i \) in \( t \).

We also defined the L1 distance between two n-dimensional vector \( u \) and \( v \) to be

\[
\sum_{1 \leq i \leq n} |u[i]-v[i]|
\]

**Lemma 4 (Content-based Mismatch Filtering).** If the edit distance between the two strings is within \( \tau \), there does not exist a probing window such that the L1 distance between the frequency histograms of two strings within the probing window is larger than \( 2\tau \).
4.2 Tightening the Lower Bound:

The above content-based mismatch filtering cannot factor into edit errors made on the right side of the probing window. We overcome this by collecting all the mismatching q-grams of one string in the candidate pair, and compute the minimum number of edit errors that destroy all the mismatching q-grams on the right side of the probing windows. We call this task the SumRightErrs, whose algorithm is similar to algorithm 2 but works in the reverse direction on the strings.

Example 2: Consider the same string in example 1. Assume the following q-grams are not matched against a candidate y:

| (ab, 1)  | (ba, 5)  | (ab, 6)  |

If we run Algorithm 2 in the reverse direction, we can compute the suffix sum array on the left hand side of the figure below:

<table>
<thead>
<tr>
<th>loc</th>
<th>SumRightErrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ \text{SUFFIX SUM ARRAY} \]

\[ \Rightarrow \]

<table>
<thead>
<tr>
<th>loc</th>
<th>SumRightErrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ \text{CONDENSED SUFFIX SUM ARRAY} \]

From the lemma 4 and SumRightErrs, we implements the tightened content-based mismatch filtering as the following:
The heuristic to select the probing window is to use the set of contiguous mismatching q-grams.

4.3 The Verification Algorithm based on Multiple Filters:

The verify algorithm used in Ed-Join is design as following:

```
Algorithm 5: ContentFilter(s, t, Q)

Input: Q is an array of mismatching q-grams sorted in increasing order of locations; String x and y form a candidate pair.
Output: A lower bound of the ed(s, t)

1. j ← 1; i ← 2;
2. while i ≤ |Q| do
3.   if Q[i].loc - Q[i - 1].loc > 1 then
4.     ε = L₁Distance(s, t, Q[j].loc, Q[i - 1].loc + q - 1)
5.     + SumRightErrs(Q[i - 1].loc + q);
6.     if ε > 2r then
7.       return 2r + 1 /* early termination here */
8.     j ← i;
9.     i ← i + 1;

return L₁Distance(s, t, Q[j].loc, Q[i - 1].loc + q - 1)
+ SumRightErrs(Q[i - 1].loc + q)
```
It applies three filtering methods in turn before running the final edit distance calculation: count and position filtering (Lines 2–3), location-based mismatch filtering (Lines 4–6), and content-based mismatch filtering (Lines 7–9).

The verify algorithm call a CompareQGrams(x,y) method. This method will calculate the number of mismatching q-gram from x to y. The following is the algorithm of the method:
Algorithm 8: CompareQGrams(x, y)

Input: x and y are two sorted q-gram arrays.
Output: Q is the set of loosely mismatching q-grams from x to y; $\epsilon$ is the number of strictly mismatching q-grams from x to y

1. $i \leftarrow 1;\quad j \leftarrow 1;\quad \epsilon \leftarrow 0;\quad Q \leftarrow \emptyset;
2. while $i \leq |x|$ and $j \leq |y|$ do
3.     if $x[i].token = y[j].token$ then
4.         if $|x[i].loc - y[j].loc| \leq \tau$ then
5.             $i \leftarrow i + 1;\quad j \leftarrow j + 1;
6.         else
7.             if $x[i].loc < y[j].loc$ then
8.                 if $x[i].token \neq x[i - 1].token$ or $x[i].token \neq y[j - 1].token$ or $|x[i].loc - y[j - 1].loc| > \tau$ then
9.                     $Q \leftarrow Q \cup x[i];$
10.                    $\epsilon \leftarrow \epsilon + 1;\quad i \leftarrow i + 1;$
11.                    $j \leftarrow j + 1;$
12.                 else
13.                     $i \leftarrow i + 1;$
14.             else
15.                 if $x[i].token < y[j].token$ then
16.                     if $x[i].token \neq x[i - 1].token$ or $x[i].token \neq y[j - 1].token$ or $|x[i].loc - y[j - 1].loc| > \tau$ then
17.                         $Q \leftarrow Q \cup x[i];$
18.                         $\epsilon \leftarrow \epsilon + 1;\quad i \leftarrow i + 1;$
19.                         $j \leftarrow j + 1;$
20.                     else
21.                     $j \leftarrow j + 1;$
22.     else
23.     if $x[i].token \neq x[i - 1].token$ or $x[i].token \neq y[j - 1].token$ or $|x[i].loc - y[j - 1].loc| > \tau$ then
24.         $Q \leftarrow Q \cup x[i];$
25.         $\epsilon \leftarrow \epsilon + 1;\quad i \leftarrow i + 1;$
26.     end if
27. end if
28. return $(Q, \epsilon)$
4.4 The exact Edit Distance Algorithm:

The following is the dynamic algorithm for calculating exact edit distance between two strings. Its complexity is O(n^2) where n is the length of the string.

---

Algorithm 9: EditDistance(String x, String y)

---

Input: 2 strings x, y
Output: the exact edit distance between x and y

1 x \rightarrow \text{char[]} \text{s size} m
2 y \rightarrow \text{char[]} \text{t size} n
// d is a table with m+1 rows and n+1 columns
3 \text{declare int} d[0..m, 0..n]
4 \text{for} i \text{ from} 0 \text{ to} m
   d[i, 0] := i
5 \text{for} j \text{ from} 0 \text{ to} n
   d[0, j] := j
6 \text{for} i \text{ from} 1 \text{ to} m
   \text{for} j \text{ from} 1 \text{ to} n \{ 
      \text{if} s[i] = t[j] \text{ then} \text{cost} := 0
      \text{else} \text{cost} := 1
      d[i, j] := \text{minimum}(d[i-1, j] + 1, // deletion
                                 d[i, j-1] + 1, // insertion
                                 d[i-1, j-1] + \text{cost} // substitution)
   \}
7 \text{return} d[m, n]

---

5 IMPLEMENTATION

5.1 Data Structure

We have a Q_gram object which represents the q-gram probing window. It has the token and token’s location.

We have a Q_gramArray object. It has an ArrayList of Q-gram object, id and string. This object stores id of the string. It can get the input string and then build a q-gram array sorted with decreasing order of idf and increasing order of location of q-grams.
We have an IEElement object which is an element of the inverted list. Each element of the inverted list have the location of q-gram w and the string that contain w. The important object is the SimilarityJoins object. It makes use of the above data structures and is built important methods for Ed-Join and Ed-Join itself.

5.2 Important Classes and Methods

Class Q_gram

Constructor   public Q_gram(String q, int location)
Methods:      public String getQ_gram()
              public int getLoc()

Class Q_gramArray

Constructor    public Q_gramArray(String st)
              public Q_gramArray(String st, int id)
              Q_gramArray(ArrayList<Q_gram> qarray)
Methods:       public ArrayList<Q_gram> getQ_gramArray()
              public String getString()
              public int getId()
              arrange() // sort the q-gram array in decreasing of idf and increasing
              of location of q-grams.

Class IEElement

Constructor    public IEElement(Q_gramArray Q, int loc)
Methods:       public Q_gramArray getQ_gramArray()
              public int getLoc()

Class Pair

Constructor    public Pair(Q_gramArray q_gramArr, int mis)
Methods:       public Q_gramArray getQ_gramArray()
              public int getMismatchNo()

Class SimilarityJoins

Constructor    public SimilarityJoins()
Methods:       public void putSymbolSet()
              public int MinEditErrors(Q_gramArray Q)
public int CalcPrefixLen(Q_gramArray x)
public int SumRightErrs(int pos, Q_gramArray Q)
public int ContentFilter(String s, String t, Q_gramArray Q)
public int L1Distance(String s, String t, int lo, int hi)
public int[] frequencyHistogram(String s, int start, int end)
public void Ed_Join()
public void Verify(Q_gramArray x, hashtable<Q_gramArray,Boolean> A)
public Pair CompareQGrams(Q_gramArray x, Q_gramArray y)
public int EditDistance(String x, String y)

Class SelfTest
Class AutoTest

6 EVALUATE THE CORRECTNESS OF SUGGESTED ALGORITHM:

6.1 The prefix filtering:
In the new prefix filtering method, the authors suggest using location-based mismatch filtering (the MinEditErrors method) to eliminate non-candidates. Although the ideas of location-based mismatch filtering are correct and novel, its application into CalPrefixLen and Ed_Join is not correct.

In location-based filtering, the parameter Q is the set of mismatch q-grams between two string x and y.

Algorithm 2: MinEditErrors (Q)

1. Sort q-grams in Q in increasing order of locations if necessary;
2. cnt ← 0; loc ← 0;
3. for i = 1 to |Q| do
4.     if Q[i].loc > loc then
5.         cnt ← cnt + 1;
6.         loc ← Q[i].loc + q - 1;
7. return cnt
However, the CalcPrefixlen method calls MinEditErrors with parameter which is the
prefix of q-gram array x:

And the application of CalcPrefixLen in Ed_Join also uses the prefix of q-gram array x
for parameter of MinEditErrors:

6.2 Simple adjustment:
We have used the two filtering methods in Verify step. To simply correct the prefix
filtering, we can just used the normal prefix filtering, which is $q \cdot \tau + 1$
6.3 Complicated adjustment:

To correctly make use of location-based mismatch filtering in prefix filtering, we need to adjust both the Ed_Join and CalcPrefixLen methods. First, we need a new method that calculate mismatching q-gram between 2 strings x, y from location 0 to hi. We call this method Mismatch(s, t, hi)

Then we adjust CalcPrefixLen as following:

We make quite a lot adjustment for Ed_Join:
With the adjustment we make, we save a lot of space in main memory. With the old Ed_Join algorithm, we need to use the hash table of inverted list I to store all different tokens as keys and an array list of strings that contain this token for each key. The cost of hash table is about n times of the string list R. With the adjustment, the space cost is just 1 list R. This brings great advantage when the data set is large.

7 TESTING METHODOLOGY

7.1 Manual Test
We build two classes for testing. The first test class, SelfTest.java, is for manual testing. We set the input in file self_input.txt. And the result will appear on console window and be stored in file self_result.txt. This class is mainly for testing the correct running and debugging of the program.

Sample input
The input file contains strings along with their ids. In this project, for simplicity, we only create $\Sigma$ with capital alphabet character. The program will automatically convert small characters to capital characters. According to the algorithm, the minimum length of each string must be equal or greater than $q^\tau(\tau+1)$. The program will automatically sort the strings in order of their length. The sorted strings are stored in file SelfInputArranged.txt.

Sample output with $\tau = 1$

```
(3.0,1.0)
(2.0,1.0)
(2.0,3.0)
(4.0,3.0)
```

The result which is pairs of strings’ id that are similar will be both printed to console window and stored in the file self_result.txt for further references.

7.2 Auto Generated Testing
The second class, AutoTest.java, will automatically generate thousands of strings for testing. The result will appear on console window and be stored in file auto_result.txt. It is designed for running experiments such as experiment with various q-gram lengths to study the effect of q-gram to the program, etc.

EXPERIMENTS:

7.3 Experiment Setup

We randomly generate data for the experiments as following:

| N    | Average string length | |∑| |
|------|-----------------------|---|
| 1 000| 14.712                | 26 |
| 5 000| 14.5598               | 26 |
| 10 000| 14.4847              | 26 |
| 25 000| 14.51212            | 26 |

Our |∑| is a set of Capital alphabet characters. Each set of strings are already sorted in their increasing order of their length and stored in files.

7.4 The great effect of filtering compare to simple similarity join:

We respectively process the datasets with simple similarity join (no filtering) and our new approach to similarity join (location-based filtering and content-based filtering) and compute the running time of each approach. In this experiment, we use q = 2 and edit distance threshold equals to 1. We get the following statistic:

<table>
<thead>
<tr>
<th></th>
<th>1000</th>
<th>5000</th>
<th>10 000</th>
<th>25 000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple approach</td>
<td>3156</td>
<td>290391</td>
<td>1150343</td>
<td>1802563</td>
</tr>
<tr>
<td>Our approach</td>
<td>500</td>
<td>11187</td>
<td>46375</td>
<td>268406</td>
</tr>
</tbody>
</table>

Table of running time of simple and new approach
From the table of experiment result, we find that when the number of strings (N) increases, the difference between the naïve approach and our new approach become greater, especially when there are not many similarity pairs in the string set. It happens because in our new approach, by the two novel filtering, most of the strings are eliminated before they reach the verify step which calculate the exact edit distance. Hence, filtering effect are extremely great with large data set in real life although in theory, both have the same complexity $O(N^2 \cdot n^2)$.

![chart of running time between simple and new approach](chart.png)

The chart of running time between simple approach and our approach

7.5 Effect of q-gram length

We set q-gram length to different values. Q-gram is the probing window along the string. Hence, we expected that difference value of q will have difference effects on the efficient of the processing. Therefore, choosing a suitable value of probing window is important for dataset.
In this experiment, we set $\tau = 1$, length of each string is fixed to be 26 for convenience and data set length (N) is 1000 and 5000 respectively.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>8</th>
<th>11</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>2750</td>
<td>750</td>
<td>560</td>
<td>484</td>
<td>688</td>
<td>692</td>
</tr>
<tr>
<td>5000</td>
<td>13000</td>
<td>3360</td>
<td>2265</td>
<td>2203</td>
<td>2407</td>
<td>2497</td>
</tr>
</tbody>
</table>

Table of running time of Ed_Join with varying value of probing window q

Based on the result of the experiment, we can find that the good value of q that reduces the running time a lot is around 8. With small value of q such as 1, 2, the probing window
length is small. Hence, large number of candidate pairs is generated. However, when q become bigger, the minimum prefix is also become bigger and then causes more the inverted list to be probed and candidate generated.

In general, for a specific data set, choosing a suitable value of q is quite important and it can help speed up the processing time a lot, especially, when the data set is large. We can try to run some small experiments with small number of strings from data set and from the result; we choose suitable value of q according to properties of data set.

7.6 Large edit distance threshold
In this experiment, we set the edit distance threshold to varying values with q=2, length of each string is 26 and dataset lengths are 1000 and 5000 respectively.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>750</td>
<td>1157</td>
<td>1875</td>
<td>2687</td>
<td>3843</td>
</tr>
<tr>
<td>5000</td>
<td>3360</td>
<td>5422</td>
<td>8422</td>
<td>12437</td>
<td>17468</td>
</tr>
</tbody>
</table>

Table of running time with varying value of edit distance threshold $\tau$

Chart of running time with varying value of edit distance threshold $\tau$
The results of the experiment are reasonable. When the edit distance threshold $\tau$ increase, it will create more candidate pairs. Hence the processing time will increase linearly.

### 7.7 Experiments of Complex Modified Ed_Join and Simple Modified Ed_Join

<table>
<thead>
<tr>
<th></th>
<th>1000</th>
<th>5000</th>
<th>10 000</th>
<th>25 000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple approach</td>
<td>3156</td>
<td>290391</td>
<td>1150343</td>
<td>1802563</td>
</tr>
<tr>
<td>Simple Modified Ed_Join</td>
<td>500</td>
<td>11187</td>
<td>46375</td>
<td>268406</td>
</tr>
<tr>
<td>Complex Modified Ed_Join</td>
<td>8531</td>
<td>194671</td>
<td>566469</td>
<td>1067745</td>
</tr>
</tbody>
</table>

Although the time complexity of Complex Modified Ed_Join is more than Simple Modified Ed_Join, the space complexity is much smaller (the space of the set of strings versus $n$ times of the set of strings with $n$ is string’s length). It is the trade off between time and space. The effect of filtering is more powerful and clearer when the dataset becomes larger.

### 8 CONCLUSIONS

#### 8.1 Achievements

From this project, we explore a novel approach to similarity join with edit distance constraints. The two filtering methods, location-based filtering and content-based filtering, capture very well two types of errors, non-clustered edit errors and clustered edit errors. When the complexity of naïve processing similarity joins is $O(N^2.n^2)$, these two novel filtering methods filter a lot of candidate strings and in real time, they speed up the processing time greatly.

From this project, we also learn to practically implement the similarity join from general instructions and suggesting algorithms in java programming language. Working from theory to real implementation is an interesting exploration.
We also learn to setup experiments, generate data for evaluation and study various aspects of the approach. And from the results of experiments, we try to draw useful value and conclusions and test the evaluation again. Because there is no guarantee that the conclusions are correct, the work is challenging and interesting.

8.2 Conclusions

Similarity join is a fundamental operation and an interesting topic in database research. In this project, we exploit a new approach to similarity join, the edit distance constraint with mismatching q-grams. We develop two new filtering methods by analyzing the locations and contents of mismatching q-grams. The two new filters are combined with existing filters in our Ed-Join algorithm and they show that the processing is speeded up greatly, especially in large database. This is really a great improvement.
REFERENCES

1. Books:
   Database Management System 3ed, R. Ramakrishnan and J. Gehrke

2. Papers:
   Chuan Xiao, Wei Wang, Xuemin Lin. EdJoin: An Efficient Algorithm for Similarity Joins With Edit Distance Constraints,

3. Internet Resources:

http://en.wikipedia.org/wiki/Levenshtein_distance
http://www.cs.colorado.edu/~main/applications/Mergesort.java
APPENDIX A – Program Listing

Class Q_gram
  Constructor   public Q_gram(String q, int location)
  Methods:     public String getQ_gram()
               public int getLoc()

Class Q_gramArray
  Constructor   public Q_gramArray(String st)
               public Q_gramArray(String st, int id)
               Q_gramArray(ArrayList<Q_gram> qarray)
  Methods:     public ArrayList<Q_gram> getQ_gramArray()
               public String getString()
               public int getId()
               arrange()// sort the q-gram array in decreasing of idf and increasing
               of location of q-grams.

Class IEElement
  Constructor   public IEElement(Q_gramArray Q, int loc)
  Methods:     public Q_gramArray getQ_gramArray()
               public int getLoc()

Class Pair
  Constructor   public Pair(Q_gramArray q_gramArr, int mis)
  Methods:     public Q_gramArray getQ_gramArray()
               public int getMismatchNo()

Class SimilarityJoins
  Constructor   public SimilarityJoins()
  Methods:     public void putSymbolSet()
               public int MinEditErrors(Q_gramArray Q)
               public int CalcPrefixLen(Q_gramArray x)
               public int SumRightErrs(int pos, Q_gramArray Q)
               public int ContentFilter(String s, String t, Q_gramArray Q)
               public int L1Distance(String s, String t, int lo, int hi)
               public int[] frequencyHistogram(String s, int start, int end)
public void Ed_Join()

public void Verify(Q_gramArray x, HashTable<Q_gramArray, Boolean> A)

public Pair CompareQGrams(Q_gramArray x, Q_gramArray y)

public int EditDistance(String x, String y)

Class SelfTest

Class AutoTest

Class Experiment

Class GenerateData
APPENDIX B – How to Use the Program

In this project, we use Eclipse Classic 3.2.4 for programming. It can be downloaded at http://www.eclipse.org/downloads/download.php?file=/eclipse/downloads/drops/R-3.4.2-200902111700/eclipse-SDK-3.4.2-win32.zip. We compile all the java class in folder SimilarityJoins.

If we want to do manual testing, we set the input in file self_input.txt and run the class SelfTest.java. The result will appear on console window and be stored in file self_result.txt.

If we want to do automatically testing, we only need to run the class AutoTest.java. The program will automatically generate 5000 strings for testing. The result will appear on console window and be stored in file auto_result.txt.

We can modify the value of edit distance threshold t and length of probing window q in class SimilarityJoins.java.