Efficient K-Nearest Neighbor Join Algorithms for High Dimensional Sparse Data

Jijie Wang, Lei Lin, Ting Huang, Jingjing Wang and Zengyou He

School of Software, Dalian University of Technology, Liaoning, China, 116600, Jijie.Wang@hotmail.com

Abstract. The K-Nearest Neighbor (KNN) join is an expensive but important operation in many data mining algorithms. Several recent applications need to perform KNN join for high dimensional sparse data. Unfortunately, all existing KNN join algorithms are designed for low dimensional data. To fulfill this void, we investigate the KNN join problem for high dimensional sparse data.

In this paper, we propose three KNN join algorithms: a brute force (BF) algorithm, an inverted index-based (IIB) algorithm and an improved inverted index-based (IIIB) algorithm. Extensive experiments on both synthetic and real-world datasets were conducted to demonstrate the effectiveness of our algorithms for high dimensional sparse data.

1 Introduction

The K-Nearest Neighbor (KNN) join operation associates each data point in one set $R$ with its $k$-nearest neighbors from another set $S$. Such KNN join operation can be used as a primitive building block in many data mining algorithms. Typical examples include $k$-nearest neighbor classification and $k$-means clustering. Therefore, several effective algorithms have been proposed recently [1,2,3,4]. However, all these existing KNN join algorithms were designed for low dimensional data in which the dimensionality is generally less than 100. This deficiency restricts the use of the KNN join in practice since several recent applications in proteomics need to perform such operation for extremely high dimensional sparse data.

Proteomics is a relatively new but rapidly developing concept within life science research [5]. Peptide identification is the key and essential step in proteomics. In this process, the peptides in the sample are identified by searching a protein database according to the measured mass tandem mass spectra (MS/MS) [6].

From the view of computation, such MS/MS-based peptide identification problem can be boiled down to a KNN join problem: Set $R$ is the group of all experimental MS/MS spectra and set $S$ is a group of theoretic spectra derived from all peptides in the database, where each spectrum is a high dimensional sparse vector (the dimensionality is around 10,000). The objective is to find $k$
possible peptides that are associated with every input MS/MS spectrum. Un-
fortunately, all existing peptide identification algorithms have never optimized
their search methods from a KNN join’s perspective, resulting in considerable
running deficiency.

Based on above observations, this paper focuses on the KNN join problem
for high dimensional sparse data. To the best of our knowledge, there are still
no available algorithms that are designed for the same purpose.

In this paper, we first propose a brute force algorithm as the baseline. Then
an inverted index-based algorithm is proposed. This algorithm has significant
performance benefits since it can avoid iterating unnecessary features in every
vector of set $S$. In order to further improve the efficiency of the inverted index-
based algorithm, we add a threshold-based refinement to it. This refinement
can dramatically reduce overhead such as index construction and inverted list
scanning during score accumulation.

Our contributions can be summarized as follows:

- We are first to investigate the high-dimensional KNN join problem for sparse
  vectors, which is rooted from the peptide identification problem in computa-
tional proteomics.
- We propose several novel KNN join algorithms for sparse vectors in the high
dimensional space.

The remainder of our paper is organized as follows: In section 2, we review
some related work in KNN join algorithms. In section 3, we define the KNN join
problem. Our proposed algorithms are presented in Section 4. Section 5 provides
a performance evaluation based on both synthetic and real data. Finally, we
conclude the paper in Section 6.

2 Related work

Several KNN join algorithms have been proposed in the past a few years [1,2,3,4,7].
MuX [4] is essentially an R-tree based method designed to satisfy the conflicting
requirements of reducing both CPU cost and I/O cost. It employs large-sized
pages to optimize I/O time and uses small-sized buckets to partition the data
with finer granularity so that CPU time can be reduced. Gorder [3] is a block
nested loop join method that exploits sorting, join scheduling and distance com-
putation filtering to reduce both I/O and CPU costs. It sorts input datasets
by G-order and applies the scheduled block nested loop join on the G-ordered
data. Yu et al. [1,2] propose an index structure called $iDistance$ to partition the
data. Reference points are selected for each partition, and every point in each
partition is mapped to a single dimensional space based on its similarity score
to the corresponding reference point. Then the one-dimensional KNN search is
performed on the transformed data indexed by a B+ tree. Since all these ex-
isting researches focus on data whose dimensionality is less than 100, none of
them could be used to handle high dimensional sparse data with nearly 10000
features.
Bayardo et al. deal with high dimensional sparse data, but their objective is to find all pairs whose similarity score is above a threshold rather than finding each point with its \( k \) nearest neighbors. Meanwhile, they focus on self similarity search and propose an efficient in-memory approach.

In our work, we devote ourselves to solving the KNN join problem with a large collection of sparse vector data in the high-dimensional space. Furthermore, the block nested loop join strategy is introduced to our algorithm since the size of the data set may exceed the available memory.

3 Problem definition

**KNN join:** Given two data sets \( R \) and \( S \), an integer \( k \) and the metric \( \text{sim()} \), the KNN join of \( R \) and \( S \), denoted as \( R \bowtie_{KNN} S \), returns pairs of vectors \((r, s)\) where \( r \in R \) and \( s \in S \), and there are at most \( k-1 \) vectors from \( S \) such that \( \text{sim}(r, s') \geq \text{sim}(r, s) \) (for any \( s' \in S \)).

In essence, the KNN join combines each vector in outer dataset \( R \) with its \( k \)-nearest neighbors in inner dataset \( S \). Both \( R = \{r_1, r_2, ..., r_{|R|}\} \) and \( S = \{s_1, s_2, ..., s_{|S|}\} \) are composed of real-valued vectors of fixed dimensionality \( D \), where \( |R| \) and \( |S| \) represent the number of vectors in set \( R \) and \( S \) respectively.

In this paper, the similarity metric is the dot product. For each vector \( r \) in \( R \) and \( s \) in \( S \), we calculate their similarity score as follows:

\[
\text{dot}(r, s) = \sum_{i=1}^{D} r[i] \cdot s[i] \quad (1)
\]

Due to the sparsity of input vectors, we present a sparse vector \( x \) as a set of pairs \((d, w)\), where \( d \) is the dimension index iterator and \( w = x[d] \), \( w > 0 \) over all \( d = 1...D \). In general, all these pairs are organized in ascending order of the value \( d \). Like [8], we define such pairs as the features of the vector and the size of these features as \( |x| \).

4 Algorithms

In this section, we describe our proposed algorithms. We first describe the block nested loop join in our work. Then a brute force algorithm is introduced as a baseline in Section 4.2. In order to improve the efficiency of brute force algorithm, we propose an inverted index-based algorithm and an improved inverted index-based algorithm in Section 4.3 and 4.4, respectively.

4.1 Block nested loop join

Block nested loop join is designed for effectively utilizing buffer pages to deal with disk resident data. It uses one page as the input buffer for scanning the inner data set \( S \), one page as the output buffer, and uses all remaining pages to
hold blocks of outer $R$. It has been proved that this strategy of caching more pages of the outer set is beneficial for I/O processing, because the inner set can be scanned for fewer times\[^9\]. In our KNN join problem, the sizes of data set $R$ and $S$ tend to exceed the available memory. Therefore we introduce block nested loop strategy into our algorithm. Suppose we allocate $n_r$ and $n_s$ buffer pages for $R$ and $S$. We partition $R$ and $S$ into blocks with the allocated buffer size. Both the blocks of $R$ and $S$ are loaded into memory sequentially, which is efficient in terms of I/O time as it significantly reduces seek overhead.

Algorithm 1 outlines the block nested loop join algorithm. It loads blocks of data set $R$ into memory sequentially (line 1-2). For the in-memory block $B_r$, $pruneScore$ of its vectors is initialized to 0 (line 3). Then blocks of data set $S$ are loaded into memory one by one (line 4-5). With each pair $B_r$ and $B_s$, we join them in memory by calling function $KNN_Join_Algorithm$ (line 6). After computing the similarity score between vectors from $B_r$ and every block in data set $S$, the KNN candidate sets for every vector in $B_r$ will be outputted as the join results (line 7).

**Algorithm 1** Block Nested Loops Join($R, S$)

*Input:* $R, S$ are two sparse vector data sets that have been partitioned into blocks.

*Description:*

1: For each block $B_r \in R$ do
2: ReadBlock($B_r$)
3: InitPruneScore($B_r$)
4: For each block $B_s \in S$ do
5: ReadBlock($B_s$)
6: KNN_Join_Algorithm ($B_r, B_s$)
7: OutputKNN($B_r$)

### 4.2 Brute force algorithm

As a baseline, one might consider a brute force algorithm: simply compute the similarity score between every vector $r$ in $B_r$ and every vector $s$ in $B_s$. If the similarity score is higher than the pruning score of $r$, then $s$ is inserted into $r$’s KNN candidate set and at the same time the pruning score is updated.

Algorithm 2 shows the outline of our brute force (BF) algorithm for KNN join problem. In BF algorithm, the most important part is the function $dot(r, s)$. This function could be implemented using a fast algorithm (shown in line 8-23) and its complexity is:

$$ C_1 = |r| + |s|, $$

where $|r|$ refers to the number of features in vector $r$ and $|s|$ refers to the number of features in vector $s$. Then, we calculate the complexity of BF algorithm as...
follows:

\[ C_2 = \sum_{i=1}^{\|B_r\|} \sum_{j=1}^{\|B_s\|} (|r_i| + |s_j|), \quad (3) \]

where \(|B_r|\) and \(|B_s|\) are the number of vectors in \(B_r\) and \(B_s\) respectively.

**Algorithm 2 Brute Force (BF) Algorithm**

1: KNNJoinAlgorithm_BF(B_r, B_s):
2: For each vector \(r \in B_r\) do
3: For each vector \(s \in B_s\) do
4: \(v = \text{dot}(r, s)\)
5: If \((v > \text{pruneScore}(r))\) then
6: Insert \(s\) into the KNN candidate set of \(r\)
7: Update \(\text{pruneScore}(r)\)

*\(\text{pruneScore}(r)\) represents the similarity score between vector \(r\) and its \(k\)th nearest neighbor*

8: \(\text{dot}(r, s)\):
9: ret=0
10: iterator_r = iterator that traverses through all the features in \(r\)
11: iterator_s = iterator that traverses through all the features in \(s\)
12: While (iterator_r.hasNext() and iterator_s.hasNext()) do
13: feature_r = iterator_r.currentValue()
14: feature_s = iterator_s.currentValue()
15: If (feature_r.d == feature_s.d) then
16: ret = ret + feature_r.w · feature_s.w
17: iterator_r.next()
18: iterator_s.next()
19: Else If (feature_r.d > feature_s.d) then
20: iterator_s.next()
21: Else
22: iterator_r.next()
23: Return ret

**4.3 Inverted index-based algorithm**

In our work, we deal with a large collection of sparse vectors in the high dimensional space. Hence, the efficiency of BF algorithm is far from satisfactory. One reason for BF’s low efficiency is due to its traversal over a great number of unnecessary features in vector \(s\) during the calculation of \(\text{dot}(r, s)\). However, these features have no contributions to the similarity score of \(r\) and \(s\). This problem will still exist even we use other methods, such as hashing and indexing, to facilitate the implementation of \(\text{dot}(r, s)\). Considering this fact, we introduce inverted list into our algorithm.
Inverted list is a set of lists \( \{ I_1, I_2, ..., I_D \} \) (one for each dimension). Each list \( I_d \) consists of a set of pairs \((x, w)\), where \( x \in S, w=x[d] \) and \( w \) is non-zero.

Algorithm 3 shows our inverted index-based algorithm: IIB. First, it calls function Create_Inverted_List_IIB to create inverted list (line 2). Then it traverses every vector \( r \) in \( B_r \) and calls function Find_Matches_IIB to find \( k \)-nearest neighbors of \( r \) (line 3-4).

Line 5-8 outlines Create_Inverted_List_IIB function. This function first visits every vector \( s \) in \( B_s \) (line 6). And for every feature in \( s \), it inserts the pair \((s, s[d])\) into \( I_d \)(line 7-8).

Find_Matches_IIB is shown in line 9-17. First, it constructs a map \( A \) to connect vector id and similarity value (line 10). And then for every feature \((d, r[d])\) in \( r \), it traverses corresponding \( I_d \) and add \( r[d] \cdot s[d] \) to \( A[s] \) for every existing pair \((s, s[d])\) in \( I_d \) (line 11-13). After all elements in \( A \) have been visited, the \( k \)-nearest neighbors of vector \( r \) are found. Note that when the KNN candidate set of \( r \) is updated, pruneScore(r) should also be updated.

Now, we compute the complexity of IIB algorithm:

\[
C_3 = \sum_{i=1}^{\lvert B_r \rvert} \lvert s_i \rvert + \sum_{i=1}^{\lvert B_s \rvert} \sum_{j=1}^{\lvert r[d] \rvert} \lvert I_{r[j],d} \rvert, \tag{4}
\]

where \( r \) refers to the \( i \)th vector in \( B_r \) and \( \lvert I_{r[j],d} \rvert \) refers to the number of pairs \((s, w)\) in \( I_{r[j],d} \).

Compared with \( C_2 \), \( C_3 \) dramatically drops because IIB algorithm avoids iterating many unnecessary features in \( s \) for the corresponding vector \( r \).

Algorithm 3 Inverted Index-based (IIB) Algorithm

1: \( \text{KNN}_\text{Join}_\text{Algorithm IIB}(B_r, B_s) \):
2: Create_Inverted_List_IIB(B_s):
3: For each vector \( r \in B_r \) do
4: \hspace{1em} Find_Matches_IIB(r)
5: Create_Inverted_List_IIB(B_s):
6: For each vector \( s \in B_s \) do
7: \hspace{1em} For each feature \((d, w) \in s\) do
8: \hspace{2em} Insert pair \((s, w)\) into list \( I_d \)
9: Find_Matches_IIB(r):
10: \( A = \) empty map from vector to similarity score
11: For each feature \((d, r[d]) \in r\) do
12: \hspace{1em} For each pair \((s, s[d]) \in I_d\) do
13: \hspace{2em} \( A[s] = A[s] + r[d] \cdot s[d] \)
14: For each \( s \) with non-zero score \( \in A \) do
15: \hspace{1em} If \( (A[s] > \text{pruneScore(r)}) \) then
16: \hspace{2em} Insert \( s \) into the KNN candidate set of \( r \)
17: \hspace{1em} Update pruneScore(r)
4.4 Improved inverted index-based algorithm

Many researches on distance join utilize the similarity score threshold to determine which candidate pair should be added into the result set. Roberto et al. [8] go a step further to exploit such threshold to reduce the amount of information indexed in the inverted lists. Unlike distance join, KNN join does not have a pre-determined threshold and hence we cannot directly utilize existing algorithms on similarity join problem. Thanks to the specific property of block nested loop join, we could utilize the computation results from previous loops to obtain a threshold that could be used in forthcoming loops. That is, we define the minimum similarity score in current block $B_r$ as $MinPruneScore = \min_{r \in B_r} pruneScore(r)$. Note that when we finish joining block $B_r$ and previous blocks from $S$, $pruneScore(r)$ is updated and so does $MinPruneScore$. Hence, $MinPruneScore$, as a threshold derived from previous computation, could be used to help compute the similarity scores between block $B_r$ and current block $B_s$ in $S$.

This threshold-based refinement is added to our improved inverted index-based algorithm, as shown in Algorithm 4. The loop in Create_Inverted_List_IIB now iterates from the most frequent feature to the least one (line 6-10), and avoids inserting any pair $(s, s[d])$ into the $I_d$ until a condition is met (line 11-13). The rationale is that we only need to include potential KNN candidates of $r$ into the inverted lists. The frequency-based ordering aims at minimizing the length of the inverted lists, which has also been utilized in [8].

In order to preserve memory and speed up the computation, we remove the feature $(d, w)$ from $s$ after it is inserted into $I_d$ (line 14). Because only partial features are indexed in the inverted lists, Find_Matches_IIB (line 17-19) does not finish the score accumulation task. Therefore, we need to continue the score calculation using non-indexed features (line 21). After doing these, we can get an exact similarity score between two vectors.

**Theorem 1.** If there exist two vectors $r$ and $s$ such that $\text{dot}(r, s) > pruneScore(r)$, then $s$ will become one of the KNN candidates of $r$.

**Proof.** We use $s'$ to denote the vector with unindexed features of $s$ and $s''$ to denote the vector with indexed features of $s$. In Create_Inverted_List_IIB, when a trivial upper bound $t$ exceeds $MinPruneScore$, it will begin to index the remaining features. Thus for any vector $r$ and indexed vector $s''$, there exists $\text{dot}(r, s') < MinPruneScore$. Since $pruneScore(r) \geq MinPruneScore$ and $\text{dot}(r, s) > pruneScore(r)$, we can deduce that $\text{dot}(r, s) > MinPruneScore$. Note that $\text{dot}(r, s) = \text{dot}(r, s') + \text{dot}(r, s'')$, hence for any vector $r$ and unindexed vector $s'$ meeting the condition $\text{dot}(r, s') < MinPruneScore$, we have $\text{dot}(r, s'') > 0$. Therefore, we have at least one indexed feature of $s$ in common with one feature of $r$, and the similarity score of $r$ and $s$ will be computed completely in line 21. After computing the similarity score and getting $\text{dot}(r, s) > pruneScore(r)$, $s$ will become one of the KNN candidates of $r$. □
Algorithm 4 Improved Inverted Index-based (IIIB) Algorithm

1: KNN Join Algorithm IIIB($B_r, B_s$):
2: Create Inverted List IIIB($B_s$):
3: For each vector $r \in B_r$ do
4: Find_Matches IIIB($r$)
5: Create Inverted List IIIB($B_s$):
6: Reorder the dimension 1...$D$ such that dimensions with the more non-zero entries in $B_r$ appear first
7: Denote the maximum value $x[d]$ over all $x \in B_r$ as $\text{maxWeight}_d(B_r)$.
8: For each vector $s \in B_s$ do
9: $t = 0$;
10: For each feature $(d, w) \in s$ do
11: $t = t + \text{maxWeight}_d(B_r) \cdot w$
12: If ($t > \text{MinPruneScore}$) then
13: Insert pair $(s, w)$ into $I_d$
14: Remove feature $(d, w)$ from $s$
15: Find_Matches IIIB($r$):
16: $A = \text{empty map from vector to similarity score}$
17: For each feature $(d, r[d]) \in r$ do
18: For each pair $(s, s[d]) \in I_d$ do
19: $A[s] = A[s] + r[d] \cdot s[d]$
20: For each $s$ with non-zero score $\in A$ do
21: $A[s] = A[s] + \text{dot}(r, s)$
22: If ($A[s] > \text{pruneScore}(r)$) then
23: Insert $s$ into the KNN candidate set of $r$
24: Update pruneScore($r$)

5 Experimental Results

A number of experiments were conducted to evaluate the performance of our KNN join algorithms. The data sets used in the evaluation consist of both synthetic and real-world data sets. Our synthetic data sets consist of 10,000 to 100,000 random sparse vectors with 10,000 dimensions. Our real data is the MS/MS data obtained from spectra of the Yeast and Worm (http://noble.gs.washington.edu/proj/percolator/).

In the Yeast and Worm datasets, each spectrum acts as the sparse vector in our experiments, where Yeast consists of 35,236 vectors and Worm consists of 207,804 vectors. In each spectrum, we treat each peak as the feature of vector. In the pre-processing step, the value of $m/z$ multiplies 10 to serve as dimension index and peak’s intensity is directly used as the value of corresponding dimension. In our experiments, we use the Yeast as dataset $R$ and the Worm as dataset $S$.

The experiments were performed on a 2.4 GHz machine with 2G RAM and a 7200 RPM SATA-IDE hard disk. The default settings of our experiments are summarized in Table 1.
We implement all three KNN join algorithms: BF, IIB and IIIB and compare their performance from different perspectives. Since those three algorithms utilize the same block nested loop strategy to handle the disk resident data, we only choose the I/O time of BF to show our algorithms' I/O performance. Furthermore, the performances of three algorithms in terms of the CPU time are presented.

| Parameter                  | Default Setting                      |
|----------------------------|--------------------------------------|
| Number of nearest neighbors| 5                                    |
| Buffer size                | Around 50% of total size of $R$ and $S$ |
| Size of $R$ data in buffer | Around 80% of buffer                 |
| Buffer page size           | 8192                                  |

### Table 1. Default parameter values

5.1 **Evaluation using synthetic datasets**

In this set of experiments, we compare the performance of BF, IIB and IIIB algorithms using the synthetic dataset.

**Effect of data size** We first study the effect of varying data size on these three algorithms. Fig. 1 shows the results for KNN join on 10,000-dimensional synthetic datasets of size varying from 10,000 to 50,000. From the results, we observe that with the increase of data size, the cost of BF increases dramatically, while IIIB and IIB perform more stable and better than BF. That is because IIB could avoid iterating unnecessary features in data set $S$, and the threshold-based refinement in IIB could prune more vectors.

![Fig. 1. Effect of data size (10000-dimensional synthetic datasets)](image-url)
Effect of relative size of data sets In this set of experiments, we joined two datasets of different sizes and studied the effect of the relative size on the performance of the join algorithm. To study such an effect, we fixed the size of \( R \) at 10,000 vectors and varied the size of \( S \) from 1,000 to 100,000 so that the relative size of \( R:S \) is changed from 10:1 to 1:10. Fig.2 shows the results.

From the results, we can observe that the costs of BF, IIB and IIIB increase in proportion to the increase of the size of data set and are not heavily affected by the relative size. Meanwhile, IIIB is the most efficient algorithm compared with BF and IIB.

![Graph showing I/O Time and CPU Time vs. Size of Dataset S(x1000)](image)

**Fig. 2.** Effect of relative size of data sets (10000-dimensional synthetic datasets)

5.2 Evaluation using real datasets

In this set of experiments, we study the performance of BF, IIB and IIIB algorithms using the real-world dataset.

Effect of \( k \) In this experiment, we study the effect of \( k \). Fig.3 shows the performance results of three KNN join algorithms when \( k \) is varied from 5 to 20 on the Yeast&Worm datasets.

From the results, we notice that with the increase of the number of nearest neighbors, the I/O time almost remains unchanged. From the CPU time's perspective, all these three algorithms increase moderately because their pruning strategies do not rely on \( k \). Thus the increase of \( k \) just leads to generating more candidates for each vector, which will cost a bit more running time. On average, IIIB is about 16% better than IIB for the Yeast&Worm datasets, and both IIIB and IIB outperform BF with the speed-up factor of around 10.

Effect of buffer size In dealing with large datasets, the KNN join algorithm must be efficient in utilizing the limited buffer size. In this experiment, we study the behavior of the join methods with respect to buffer size.
The study is performed on the Yeast&Worm datasets and we decrease the buffer size from around 50% of the total dataset size to around 10% of the total dataset size. In Fig. 4, we compare the performance of BF, IIB and IIIB. From I/O time’s perspective, with the decrease of the buffer size, the I/O time increases unavoidably because small buffer size will cost more I/O access. From CPU time’s perspective, IIB still performs better than IIB and the gap is more evident with the decrease of the buffer size. The reason is that smaller buffer size will bring a more powerful and accurate threshold-based refinement. Meanwhile, IIIB and IIB still have a great advantage over BF.

6 Conclusions

K-nearest neighbor join is the basis of many applications, including some recent applications in proteomics. These applications need to perform KNN join for extremely high dimensional sparse data. In this paper, we have proposed three algorithms to efficiently solve this problem. BF is a brute force algorithm which acts as the baseline. IIB utilizes the inverted lists and has a great advantage against BF. IIIB, which is based on IIB, uses the threshold-based refinement to
solve the KNN join problem more efficiently. We did our performance study on both synthetic and real-world datasets. The results confirm that IIIB and IIB are scalable with respect to the number of nearest neighbors, buffer size, data size and relative size of dataset.

In the future work, we will focus on how to improve the efficiency of IIIB algorithm by finding more powerful refinement strategy. Meanwhile, we will use the proposed algorithms as the basis to implement a more efficient protein search engine.

References

1. Yu, C., Ooi, B., Tan, K., Jagadish, H.: Indexing the distance: An efficient method to kNN processing. In: Proceedings of the 27th International Conference on Very Large Data Bases. (2001) 421–430
2. Yu, C., Cui, B., Wang, S., Su, J.: Efficient index-based KNN join processing for high-dimensional data. Information and Software Technology 49(4) (2007) 332–344
3. Xia, C., Hu, H., Ooi, B.C., Hu, J.: Gorder: an efficient method for kNN join processing. In: Proceedings of the 30th International Conference on Very Large Data Bases. (2004) 756–767
4. Böhm, C., Krebs, F.: The k-nearest neighbour join: Turbo charging the KDD process. Knowledge and Information Systems 6(6) (2004) 728–749
5. Aebersold, R., Mann, M.: Mass spectrometry-based proteomics. Nature 422(6928) (2003) 198–207
6. McHugh, L., Arthur, J.: Computational methods for protein identification from mass spectrometry data. PLoS Computational Biology 4(2) (2008)
7. Böhm, C., Kriegel, H.: A cost model and index architecture for the similarity join. In: Proceedings of the 17th IEEE International Conference on Data Engineering. (2001) 411–420
8. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In: Proceedings of the 16th International Conference on World Wide Web. (2007) 131–140
9. Ramakrishnan, R., Gehrke, J.: Database management systems. McGraw-Hill, Inc. New York, NY, USA (1999)
10. Sarawagi, S., Kirpal, A.: Efficient set joins on similarity predicates. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data. (2004) 743–754
11. Turtle, H., Flood, J.: Query evaluation: strategies and optimizations. Information Processing & Management 31(6) (1995) 831–850