

Merging the Results of Approximate Match Operations

Sudipto Guha
U of Pennsylvania
sudipto@central.cis.upenn.edu

Nick Koudas Amit Marathe Divesh Srivastava
AT&T Labs–Research
{koudas,marathe,divesh}@research.att.com

Abstract

Data Cleaning is an important process that has been at the center of research interest in recent years. An important end goal of effective data cleaning is to identify the relational tuple or tuples that are “most related” to a given query tuple. Various techniques have been proposed in the literature for efficiently identifying approximate matches to a query string against a single attribute of a relation. In addition to constructing a ranking (i.e., ordering) of these matches, the techniques often associate, with each match, scores that quantify the extent of the match. Since multiple attributes could exist in the query tuple, issuing approximate match operations for each of them separately will effectively create a number of ranked lists of the relation tuples. Merging these lists to identify a final ranking and scoring, and returning the top-K tuples, is a challenging task.

In this paper, we adapt the well-known footrule distance (for merging ranked lists) to effectively deal with scores. We study efficient algorithms to merge rankings, and produce the top-K tuples, in a declarative way. Since techniques for approximately matching a query string against a single attribute in a relation are typically best deployed in a database, we introduce and describe two novel algorithms for this problem and we provide SQL specifications for them. Our experimental case study, using real application data along with a realization of our proposed techniques on a commercial data base system, highlights the benefits of the proposed algorithms and attests to the overall effectiveness and practicality of our approach.

1 Introduction

The efficiency of every information processing infrastructure is greatly affected by the quality of the data residing in its databases (see, e.g., [9]). Poor data quality is a result of a variety of reasons, including data entry errors, poor integrity constraints or lack of standards for recording values in database fields (e.g., addresses). Data of poor quality could instigate a multitude of business problems,

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

**Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004**

for example inefficient customer relationship management (inability to retrieve customer information during a service call), billing errors (sending a bill to a wrong address) and distribution delays (erroneous delivery of goods) impacting the overall effectiveness of a business. Recognizing the importance of this problem, a variety of commercial products (see, e.g., [1]) and research prototypes (see, e.g., [21]) target the space of data cleaning, offering an array of techniques to identify and correct data quality problems. At a high level, data cleaning solutions can be classified into two broad categories: (a) those that operate on top of an RDBMS using an SQL interface to express and realize data cleaning tasks [16, 17, 18, 7] and (b) those that extract the relevant data out of a database and operate on them using proprietary techniques and interfaces [1].

The majority of cleaning techniques focus on the identification of problems on attribute values of string type (e.g., customer names, addresses, product names, etc.) which abound in customer related databases. These include techniques for indexed retrieval of strings based on notions of approximate string match [7], correlating string attributes using string similarity predicates (e.g., cosine similarity, edit distance and variants thereof) [17, 18, 6, 5] and deploying algorithms and/or rule engines for automatically correcting/transforming strings into canonical forms [1, 3, 22]. These techniques will successfully match a query string (or a collection of strings) approximately (for suitably defined notions of approximate match) against the values of an attribute in a relation R . For a given query string, such techniques can tag each matching attribute value with a score quantifying the degree of similarity (closeness) of the query string to the attribute value string. As a result, they can effectively return a scored ranking of the attribute values (and hence the tuples) in R with respect to the query string. It is then up to the user, to observe the highest scoring attribute values and identify the one(s) that should be declared as good approximate matches.

Although such techniques are effective in identifying approximate matches between a string and the values of a relational attribute, it is often the case that *multiple* attributes should be involved in the approximate match operation. Consider for example a relation $R(\text{custname}, \text{address}, \text{location})$, recording the name of a customer, the customer address, and the geographical coordinates (latitude, longitude) of the customer address. Given a tuple of query strings, Q (possibly obtained from a different database), we wish to obtain approximate matches

Relation R			
tuple id	$custname$	$address$	$location$
t_1	John Smith	800 Mountain Av Springfield	5,5
t_2	Josh Smith	100 Mount Av Springfield	8,8
t_3	Nicolas Smith	800 Spring Av Union	11,11
t_4	Joseph Smith	555 Mt Road Springfield	9,9
t_5	Jack Smith	100 Springhill Lake Park	6,6

Query		
$Q_{custname}$	$Q_{address}$	$Q_{location}$
John Smith	100 Mount Rd Springfield	5.1,5.1

Scored Rankings		
$custname$	$address$	$location$
t_1 (1.0)	t_2 (0.95)	t_1 (0.95)
t_2 (0.8)	t_1 (0.8)	t_5 (0.9)
t_5 (0.7)	t_4 (0.75)	t_2 (0.7)
t_4 (0.6)	t_3 (0.3)	t_4 (0.6)
t_3 (0.4)	t_5 (0.1)	t_3 (0.3)

Figure 1: Example

with tuples of R . Let $Q_{custname}, Q_{address}, Q_{location}$ be the values specified in Q . For each of these values assume there is an agreed upon methodology to generate approximate matches with tuples of R . The specific methodology used for each attribute is orthogonal to our discussion; any of the known techniques could be applied. For example, for $Q_{custname}$ one could utilize the technique of [17] deploying string edit distance to generate approximate matches with attribute values from attribute $custname$ of R . This effectively creates an ordering (ranking) of relation R based on non-decreasing values of the edit distance (possibly thresholded) between $Q_{custname}$ and $R(custname)$. For $Q_{address}$ one could decide to utilize the techniques of [18, 7] and generate a list of approximate matches using the $address$ attribute of R . A ranking of R will be obtained in this case as well, based on tf-idf and cosine similarity of $Q_{address}$ and $R(address)$. Finally, for $Q_{location}$ we could choose spatial closeness (in terms of some spatial norm, e.g., Euclidean distance between $Q_{location}$ and $R(location)$) and generate a corresponding list of matches (and, hence, ranking of R). Figure 1 presents an example of such (scored) rankings. Notice that the resulting rankings and scores heavily depend on the specific methodology applied to derive them.

Although in principle one could use the information conveyed by only one ranking to form an answer, this is not necessarily a good choice. For example, tuple t_5 ranks second (with a score of 0.9) in the ranking on $location$, but ranks poorly on the other query attributes. Evidently, a more conclusive answer could be obtained by merging the positions and/or scores of the tuples in all rankings. This can be accomplished with the use of *merging functions* operating on the tuples of the various rankings. For purposes of exposition, consider a merging function that sums the scores of the tuples in the various rankings. In this case notice that tuple t_1 achieves a better value (2.75) in that sum than any other tuple in R . Intuitively, this is a better approach, as it is obtained by examination of all the query attributes and the results of all the induced rankings. It is evident that, in the presence of different rankings, principled techniques are required to derive a final ranking by taking into account information conveyed by all rankings. Such

principled techniques can be realized as merging functions operating on the tuples of rankings. The derivation of a final ranking can then be cast as an optimization problem over the values of such merging functions.

Given that a variety of approximate match techniques are fully expressible in a declarative way, and best deployed in a database, rankings can be easily generated by directly operating on R using SQL. Thus, they can be readily materialized as relations as an input to the merging. The result of the merging itself may be used as an input to other queries/operations in the database. As a result, it is desirable to express methodologies that merge the information conveyed by the rankings in a declarative way as well.

In this paper, we study efficient techniques to merge rankings produced as a result of approximate match operations, and their declarative expression. In particular, we consider the top- k selection problem, and propose efficient algorithms for this problem. This is defined as the task of deriving an optimal final ranking (for suitably defined notions of optimality) of user-specified length k when individual rankings to be merged are generated with respect to a single query tuple. We make the following contributions:

- We study the top- k selection problem in the context of merging results of approximate match operations and show how it can be modeled as an instance of the minimum cost perfect matching problem.
- We show how the preferred solution to the minimum cost perfect matching problem, namely the *Hungarian Algorithm (HA)*, can be adapted in the context of the top- k selection problem.
- We propose a modification of *HA* tailored to the top- k selection problem, called *MHA*, analytically show its superiority over *HA*, and provide its full SQL specification.
- We propose a new algorithm, called *SSP*, specifically for the top- k selection problem, and fully specify the algorithm in SQL as well.
- We present a thorough performance evaluation of all the applicable algorithms using real data sets on a commercial database system, identifying their relative performance and applicability.

In section 2, we introduce background material, and formally define the problem of interest on ranked lists. Section 3 presents our proposed techniques, including complete SQL specification of our proposed solutions. Section 4 details a performance evaluation of our proposals. In section 5, we discuss the enhancements of our basic techniques to deal with scores. In section 6 we review work related to the problems of interest in this paper. Finally, section 7 concludes the paper by discussing additional problems of interest in this important area.

2 Background and Definitions

Let R be a relation of cardinality n consisting of m attributes and let t_1, \dots, t_n denote the tuples of R . A ranking of R is an ordered list $\tau = \{x_1 \preceq x_2 \preceq \dots \preceq x_n\}$, where

$x_i \in R$ and \preceq is some ordering relation on R . Such a ranking can be obtained by applying a suitable approximate match operation between a query value and the values of an attribute of R . Without loss of generality, we assume that rankings of R are complete in the sense that every $t_i \in R$ belongs to the ranking. If this is not the case (for example if the approximate match predicate is thresholded) and the ranking contains $S \subset R$ tuples, we obtain a complete ranking by padding the partial ranking at the end (with the same rank value) with all the “missing” $R \setminus S$ tuples. Semantically, this declares that we are indifferent about the relative ordering of the missing tuples and are all equivalently placed at the end of the ranking obtained. For a ranking τ , we use the notation $\tau(t_i)$ to refer to the position of tuple t_i in the ranking (a highly ranked tuple has a low numbered position). At times an approximate match operation used to generate a ranking provides an approximate match score (see, e.g., [18, 17, 7]).

Given a query tuple t_q with m values, let τ_1, \dots, τ_m denote the resulting rankings of R after applying approximate match operations on corresponding values of t_q and attributes of R . We will deploy *merging functions* to synthesize new rankings of R out of τ_1, \dots, τ_m . A variety of such functions have been proposed in the literature [10, 8, 20]. The bulk of our discussion equally applies to metric merging functions [13]. To ease presentation, we adopt for the bulk of the paper one instance of such a function, namely the *footrule distance*. We stress however that our methodologies are orthogonal to the choice of a metric merging function.

Definition 1 (Footrule Distance) *Let σ and τ be two rankings of R . The footrule distance between the two rankings is defined as $F(\sigma, \tau) = \sum_{i=1}^n |\sigma(t_i) - \tau(t_i)|$. ■*

Thus, the footrule distance is the sum of the absolute differences of the positions of tuples of R in the two rankings. Notice that $F(\sigma, \tau)$ can be divided by $\frac{n^2}{2}$ if n is even or $\frac{n^2-1}{2}$ if n is odd (this maximum value is attained when rankings are ordered in opposite ways) and a normalized value between 0 and 1 can be obtained. It is evident that $F(\sigma, \tau)$ can be defined over the approximate match scores of the corresponding rankings as well. For simplicity of exposition, we adopt definition 1 for the bulk of the paper. Such a measure generalizes in a natural way to account for multiple rankings. The footrule distance between rankings σ and τ_1, \dots, τ_m is defined as

$$F(\sigma, \tau_1, \dots, \tau_m) = \sum_{i=1}^m F(\sigma, \tau_i)$$

To a large extent, any interpretation of results obtained will be in terms of this distance measure. While such a measure seems natural we will not delve into the discussion of its goodness. We refer the reader to the vast bibliography [10, 20, 8] discussing and analyzing the relative goodness and applicability of the various distance measures.

We are now ready to define formally the main problem of interest in this paper:

Problem 1 (top- k selection problem) *Given a query tuple t_q specifying m attribute values, let $\tau_i, 1 \leq i \leq m$ be the rankings obtained as a result of approximate match operations on R . The top- k selection problem aims to efficiently identify the first k (for a user specified parameter k) tuples of the ranking σ of R that minimizes the footrule distance $F(\sigma, \tau_1, \dots, \tau_m)$. ■*

The first k tuples of the ranking σ of R are derived as a result of the minimization of $F(\sigma, \tau_1, \dots, \tau_m)$. Each tuple t_i in σ is tagged with a cost, referred to as the *ranking cost* as the result of this minimization. A tuple t_i at position $j, 1 \leq j \leq k$, in σ has a ranking cost $\sigma^r(t_i, j) = \sum_{l=1}^m |\tau_l(t_i) - j|$. The answer to the top- k selection problem is a set (of cardinality k) of triples $(t_i, j, \sigma^r(t_i, j))$, where j is the rank (position) of t_i in the result σ , such that among all possible mappings of the tuples t_i to positions $j', 1 \leq j' \leq k, \sum_{j'=1}^k \sigma^r(t_i, j')$ is minimized.

The bulk of this paper deals with merging ranked lists. In section 5, we discuss how our techniques can be extended to effectively deal with approximate match scores.

3 Top- k Selections

In this section we develop our proposed solutions to the top- k selection problem. We first show that the problem can be modeled as a modification of a well-studied problem, namely the *minimum cost perfect matching* [2] problem and show how such an instance can be obtained given our setting. We start from an existing technique to solve the minimum cost perfect matching problem, namely the *Hungarian algorithm (HA)*, and show how this algorithm can be adapted to obtain a solution to our top- k problem. We then propose a *Modified Hungarian Algorithm (MHA)* that takes advantage of the special structure of the top- k selection problem, resulting in improved performance. We also design a new solution for the top- k selection problem, the *Successive Shortest Paths algorithm (SSP)*, and provide its declarative specification. In section 4, we present a comparative evaluation of the three algorithms.

Given an edge-labeled bipartite graph $G = (N_1 \cup N_2, E)$, let $c_{ij}, i \in N_1, j \in N_2$ be the edge cost associated with each edge in E . A minimum cost perfect matching in G is the set of edges $E' \subset E$ of minimum cumulative cost, such that each $i \in N_1$ is incident to exactly one $j \in N_2$ in E' and vice versa. The solution to the minimum cost perfect matching can be reached by solving a linear program minimizing the overall sum of edge costs [2].

Given rankings $\tau_i, 1 \leq i \leq m$, provided as a result of approximate match queries on R , we construct a bipartite graph G as follows. Assume that each ranking τ_i is a complete ranking of R (thus contains n tuples). We instantiate n nodes corresponding to each tuple t_1, \dots, t_n of R and n nodes corresponding to the positions 1 through n of the target ranking σ . For each of the n^2 pairs of tuples t_i and positions j with $1 \leq i, j \leq n$ we introduce the edge (t_i, j) with a ranking cost $\sigma^r(t_i, j) = \sum_{l=1}^m |\tau_l(t_i) - j|$. This will instantiate a full bipartite graph G . A solution of the minimum cost perfect matching problem on G would produce a minimum cost assignment of each of the n tuples to exactly

$$\begin{pmatrix} & 1 & 2 & 3 \\ t_1 & 5 & 3 & 20 \\ t_2 & 6 & 2 & 3 \\ t_3 & 1 & 3 & 20 \end{pmatrix}$$

Figure 2: Example matrix for graph G

one of the n positions, such that no two tuples are assigned to the same position.

The top- k selection problem is a modification of the minimum cost perfect matching problem, in which we restrict our interest in identifying only the k tuples of R matching the first k positions of σ having minimum sum of ranking costs. Notice that a solution to the minimum cost perfect matching problem on G , restricted to the first k positions, does not necessarily yield a solution to the top- k selection problem; the cumulative cost might be sub-optimal. To illustrate this point consider the following example. Let G consist of three tuple vertex nodes and three position nodes. The matrix representation of G with each entry expressing the cost associated with the corresponding edge is shown in figure 2. A solution to the top-2 problem for this graph is $\{t_3, t_2\}$ (with a cost of 3) but the minimum cost perfect matching solution is $\{t_3, t_1, t_2\}$ (with a cost of 4, restricted to the first two positions).

3.1 The Hungarian Algorithm (HA)

Let S be the $n \times n$ matrix, corresponding to the matrix representation of the bipartite graph G , with elements $\sigma^r(t_i, j), 1 \leq i, j \leq n$. The Hungarian algorithm solves the minimum cost perfect matching problem by providing a solution to the corresponding linear problem established on S . We start by briefly reviewing this method below; we refer the interested reader to the bibliography on the subject for further details [2]. We then show how such an algorithm can provide a solution to the top- k selection problem as well.

We refer to a row or column of a matrix as a *line*. A set of elements of a matrix is independent if no two of the elements lie on the same line. If a letter c is written next to a row or column of a matrix, we say that the corresponding line is *covered*.

Theorem 1 (Konig's Theorem) *For a square matrix A , the maximum number of independent zeros of A is equal to the minimum number of lines required to cover all the zeros in A . ■*

This theorem forms the basis for the Hungarian method. The algorithm operates as follows:

1. Subtract the smallest element in each column of S from every element in the column yielding matrix S_1 .
2. Subtract the smallest element in each row of S_1 from each element in the row yielding matrix A . A contains at least one zero in every row and column and it is said to be in *standard form*.
3. Find k_m the maximum number of independent zeros of A . Using Konig's theorem, this is the minimum number of lines required to cover all the zeros in A .

If $k_m = n$ then there are n independent zeros in A and the row and column position of these zero elements provide a solution to the minimum cost perfect matching problem.

4. If $k_m < n$, let N denote the matrix of non covered elements of A and let h be the smallest element in N . Add h to each twice covered element of A and subtract h from each non covered element of A . Let the resulting matrix be A^* . Repeat step 3 using matrix A^* instead of A .

This procedure is guaranteed to terminate after a finite number of steps, due to the reduction at each step of the sum of the elements in the resulting matrices (A, A^* , etc). It remains to specify the procedure to identify the maximum number of independent zeros k_m .

Identifying maximum number of independent zeros:

The procedure starts by searching each column of A until a column with no entry marked with the special marker 0^* is found (if every entry contains a 0^* , then $k_m = n$ and the problem is solved). This column is referred to as *pivotal* and it is searched for all its zeros. The rows in which these zeros appear are searched for 0^* in turn until a row containing no 0^* is found. The zero in this row and the pivotal column is marked with the special marker 0^* . If each 0 in the pivotal column has a 0^* in its row, then these row numbers are listed in any order: i_1, \dots, i_t . Further terms are added as follows: Consider the 0^* in row i_1 and all zeros in its column; add to the sequence the row numbers of these zeros in any order (avoiding duplication). Continue with the 0^* 's in rows i_2, \dots, i_t as well as with the terms of the sequence after i_t . There are two possibilities, either a row i_s is reached which does not contain a 0^* or every row whose number is after i_t contains a 0^* . In the former case a transfer of a 0^* happens as follows: row number i_s was added to the sequence because row i_s contained a 0 in the column of a 0^* belonging to some row i_r . This 0^* in the row i_r is transferred to the zero in row i_s thus staying in the same column. Further transfers may happen until the 0^* is transferred from a zero in a row in the initial sequence i_1, \dots, i_t . Then the zero in this row and in the pivotal column can be marked by a 0^* . In the latter case, let v be the rows in the sequence. These rows contain a 0^* . The v columns containing these 0^* 's along with the pivotal column contains 0 's only in the v rows represented in the sequence. It follows that these v rows along with the remaining $n - v - 1$ columns contain all the zeros in A . Thus the matrix can be replaced by A^* (as dictated in step 4) and the algorithm can proceed to step 3.

Figure 3 presents an example operation of HA on the data obtained from example 1. Due to space limitations, we do not provide the full SQL specification of algorithm HA.

Our top- k selection problem cannot be solved by a direct application of this procedure, since an answer to the minimum cost perfect matching problem restricted to the first k positions is not necessarily an answer to the top- k selection problem. We observe, however, that it is possible

$$\begin{array}{cc}
\begin{pmatrix} 1 & 2 & 5 & 8 & 11 \\ 3 & 2 & 3 & 6 & 9 \\ 11 & 8 & 5 & 2 & 1 \\ 8 & 5 & 2 & 1 & 4 \\ 7 & 4 & 3 & 4 & 5 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 3 & 7 & 10 \\ 2 & 0 & 1 & 5 & 8 \\ 10 & 6 & 3 & 1 & 0 \\ 7 & 3 & 0 & 0 & 3 \\ 5 & 1 & 0 & 2 & 3 \end{pmatrix} \\
\text{(a) Initial matrix from example 1} & \text{(b) After step 2} \\
\begin{pmatrix} 0^* & 0 & 3 & 7 & 10 & c \\ 2 & 0^* & 1 & 5 & 8 & c \\ 10 & 6 & 3 & 1 & 0 & c \\ 7 & 3 & 0^* & 0 & 3 & c \\ 5 & 1 & 0 & 2 & 3 & \end{pmatrix} & \begin{pmatrix} 0^* & 0 & 3 & 7 & 10 \\ 2 & 0^* & 1 & 5 & 8 \\ 10 & 6 & 3 & 1 & 0^* \\ 7 & 3 & 0 & 0^* & 3 \\ 5 & 1 & 0^* & 2 & 3 \end{pmatrix} \\
\text{(c) Conflict in row 4 columns 4,3} & \text{(d) Final result}
\end{array}$$

Figure 3: (a) Initial Matrix (b) after step 2 (c) Conflict in row 4 and columns 4,3, the covered rows are shown at that point. The 0^* in row 4 column 3 is moved to row 5 column 3. A 0^* can now be placed at row 4 column 4 and the algorithm proceeds by placing the final 0^* at row 3 column 5 (d) final result, t_1, t_2, t_5, t_4, t_3

to modify this procedure in a way that deriving a solution to the top- k selection problem becomes possible.

In the top- k selection problem one is only interested in the first k positions of the ranking. Thus, when realizing matrix A for a supplied k value, we only need to compute the elements $\sigma^r(t_i, j), 1 \leq i \leq n, 1 \leq j \leq k$ of A . Algorithm HA however, requires a square matrix (of size $O(n^2)$) to operate correctly since it solves an $n \times n$ matching problem (otherwise the corresponding program will be under specified).

We observe that we can still utilize algorithm HA for the top- k selection problem by inserting a large positive constant¹ for each of the remaining $n(n - k)$ elements. This serves two purposes: (a) it creates a square A matrix as required by algorithm HA and (b) it guarantees that the solution extracted from the solution of the corresponding minimum cost perfect matching on A is a correct solution to the top- k selection problem. By inserting a large positive constant in the corresponding positions of A we essentially add edges of very high cost in the corresponding bipartite graph. Since such edges are of high cost they can never participate in an assignment concerning the first k positions.

A natural question arises regarding the necessity of those $n(n - k)$ additional elements for the algorithm's correctness. The basic form of execution of algorithm HA for the top- k selection problem requires inserting and maintaining these additional $n(n - k)$ values. To see this consider the following: at any iteration of the algorithm, when searching for the maximum number of independent zeros it is possible that step 4 of the algorithm executes. Let i be the column that instigates the execution of step 4. Notice that during the execution of step 4 the contents of any of the elements of A might change but the contents of column i do not. As a result, the contents of column i (for any i) are always required, since they might be affected in later iterations. Consequently, all columns have to be maintained during the entire execution of the algorithm, bringing the total space requirement to $O(n^2)$.

From a performance standpoint, such an adaptation of

¹Greater than the maximum value among $\sigma^r(t_i, j), 1 \leq i \leq n, 1 \leq j \leq k$

HA for the top- k selection problem only reduces the initial overhead of constructing matrix A . Subsequent operations however, will be on the entire matrix A of n^2 elements. The running time of this procedure is $O(n^3)$ since each of the n columns will be considered and, in the worst case, the algorithm will have to examine $O(n^2)$ elements in A for each column considered.

3.2 Algorithm MHA

In this section, we propose a modification of the basic HA algorithm for the top- k selection problem, which operates in space $O(nk)$, with worst case running time $O(nk^2)$.

One observation regarding the operation of algorithm HA is that since the algorithm provides a solution to a linear program, the optimal solution is not affected by a rearrangement of the columns of matrix A . Thus, consider populating matrix A as follows: insert a large positive constant to positions (i, j) of A with $1 \leq i \leq n, 1 \leq j \leq n - k$ and populate the remaining positions with nk ranking costs $\sigma^r(t_i, j), 1 \leq i \leq n, 1 \leq j \leq k$ at matrix positions $(i, n - k + j)$. Denote the resulting matrix as A' . Algorithm MHA operates similarly to HA on matrix A' . Now consider the operation of HA on this modified matrix. While searching for the maximum independent number of zeros in the first $n - k$ columns the algorithm can place 0^* in positions $(l, l), 1 \leq l \leq n - k$ of A' immediately. The reason is that once A' is in standard form positions $1 \leq i \leq n, 1 \leq j \leq n - k$ of the matrix are all zero. The final solution is now represented by the position of the 0^* 's in columns $n - k + 1 \dots n$. The following invariant holds at any point in the execution of the algorithm:

Invariant 1 All columns from 1 to $n - k$ are identical, differing only in the positions of their 0^* 's. Moreover, for any row containing no 0^* , the corresponding entry for this row and any column $j, 1 \leq j \leq n - k$ is 0. ■

As a result of this invariant, it is evident that it is no longer required to explicitly materialize the entire $n \times (n - k)$ sub matrix of A' with large positive values. It is sufficient to only maintain $O(n)$ values representing the state of all the 0^* elements of the A' matrix in the first $n - k$ columns. This has the potential for improved efficiency and performance as algorithm MHA has to operate on a matrix of size $O(nk)$ as opposed to one of size $O(n^2)$ as HA does. The running time of this procedure is $O(nk^2)$ since only k columns will be considered and, in the worst case, the algorithm will have to examine $O(nk)$ elements in A for each column considered.

Figure 4 presents an example operation of algorithm MHA on the data of figure 1. For purposes of exposition we show the entries of the entire matrix during the execution presented in figure 4. Notice that for this top-2 example, only 15 ($3 * n$) entries are required to be explicitly maintained.

3.2.1 SQL Specification of MHA

Complete SQL specification in procedural SQL of algorithm MHA is provided in figures 5 and 6. Such forms of

$$\begin{array}{cc}
\begin{pmatrix} 1 & 2 & M & M & M \\ 3 & 2 & M & M & M \\ 11 & 8 & M & M & M \\ 8 & 5 & M & M & M \\ 7 & 4 & M & M & M \end{pmatrix} & \begin{pmatrix} M & M & M & 1 & 2 \\ M & M & M & 3 & 2 \\ M & M & M & 11 & 8 \\ M & M & M & 8 & 5 \\ M & M & M & 7 & 4 \end{pmatrix} \\
\text{(a) Initial matrix for top-2} & \text{(b) After rearrangement} \\
\begin{pmatrix} 0^* & 0 & 0 & 0^* & 0 \\ 0 & 0^* & 0 & 2 & 0 \\ 0 & 0 & 0^* & 10 & 6 \\ 0 & 0 & 0 & 7 & 3 \\ 0 & 0 & 0 & 6 & 2 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 & 0^* & 0 \\ 0 & 0 & 0 & 2 & 0^* \\ 0 & 0 & 0^* & 10 & 6 \\ 0^* & 0 & 0 & 7 & 3 \\ 0 & 0^* & 0 & 6 & 2 \end{pmatrix} \\
\text{(c) Conflict in row 1 column 4} & \text{(d) Final result.}
\end{array}$$

Figure 4: (a) Initial Matrix suitable for top-2 problem (b) after rearrangement of columns (c) Conflict in row 1 columns 1,4. The 0^* in column 1 row 1 is moved to column 1 row 4 and the 0^* is placed at row 1 column 4. In the next iteration another conflict appears between row 2 columns 5,2. The 0^* in column 2 row 2 is moved to row 5 column 2 and a 0^* is placed at column 5 row 2. (d) final solution is, t_1, t_2

procedural SQL are offered by all major RDBMS systems. Figure 5 contains variable declarations and initializations required by the main procedure body shown in figure 6. The schema associated with this procedure is not provided due to space limitations; it can be easily derived from the procedure body, however.

Block 1 initializes the Top relation by placing a 0^* in each of the last $n - k$ columns. We then iterate over each remaining column. Within every iteration, block 2 tries to place a 0^* directly in the current column. If that is not possible, block 3 determines whether a 0^* can be placed in the current column by carrying out a series of transfers, as described in section 3.1. If possible, block 4 performs these transfers. Block 5 handles the final case when we find a set of $k_m < n$ lines which cover all the zeroes of the matrix.

3.3 Algorithm SSP

In this section we propose an alternate solution to the top- k selection problem which we refer to as *Successive Shortest Paths (SSP)*. We first provide the intuition behind this algorithm with the example shown in figure 7.

The figure presents an example of a top- k selection problem represented as an instance of a minimum cost perfect matching problem. With a database consisting of five tuples the corresponding full bipartite graph G is constructed. An edge between a node t_i and a position j in figure 7 is assumed to have a ranking cost of $\sigma^r(t_i, j)$. We refer to the nodes in G corresponding to tuples (t_1 to t_5) as tuple vertices and to the nodes corresponding to positions (1 through 5) as position vertices (or simply positions). Let's observe, in an inductive way, issues arising when, having constructed a top- i answer, we wish to extend it and construct a top- $(i + 1)$ answer.

First, constructing a top-1 answer given the graph of figure 7 is an easy task. We just select the edge with minimum ranking cost incident to position 1. The corresponding tuple (say tuple t_3 in figure 7) is the top-1 answer. Assume that through some mechanism we have identified the top- i answers (e.g., $i = 3$ in figure 7 and the answer is indicated by the highlighted edges). Using the top- i solution

```

DECLARE COMMONCOL integer DEFAULT -1;
DECLARE pivot INTEGER;
DECLARE rcount INTEGER;
DECLARE rowmin FLOAT;
DECLARE colmin FLOAT;
DECLARE elemposmin FLOAT;
DECLARE r integer;
DECLARE c integer;
DECLARE cold integer;
DECLARE cnew integer;
DECLARE found integer;

SET c = 1;
WHILE (c ≤ K) DO
  select min(cost) into colmin
  from Graph
  where pos = c;
  update Graph
  set cost = cost - colmin
  where pos = c;
SET c = c + 1;
END WHILE;
IF (K < N) THEN
  update GTail
  set cost = 0
  where pos = COMMONCOL;
ELSE
SET r = 1;
WHILE (r ≤ N) DO
  select min(cost) into rowmin
  from Graph
  where elem = r;
  update Graph
  set cost = cost - rowmin
  where elem = r;
SET r = r + 1;
END WHILE;
END IF;
delete from Top;

```

Figure 5: Initialization for MHA

to construct the top- $(i + 1)$ solution two cases of interest arise; we will use $i = 3$ in figure 7 to illustrate these cases. Assume that the top-3 solution $(t_i, j), 1 \leq j \leq 3$ identified is $S_{top-3} = \{(t_2, 1), (t_3, 2), (t_1, 3)\}$. Thus the associated total cost of the top-3 solution is $C_{top-3} = \sigma^r(t_2, 1) + \sigma^r(t_3, 2) + \sigma^r(t_1, 3)$. Constructing the top-4 solution involves including a new tuple in the solution.

The first case arises when following a direct edge from position 4 (position $(i + 1)$) to a tuple which is currently unmatched (i.e., not present in the top-3 solution). In this case a *candidate solution* can be constructed by adding the new tuple, say, $(t_4, 4)$ to the current top-3 set. A top-4 candidate solution is obtained as a direct extension of the current top-3 solution. If the total cost of the top-3 solution plus the ranking cost of the new tuple (say tuple t_4 in figure 7) assigned at position 4 in minimum (among all possible four-tuple sets assigned to positions 1 through 4), then this is the top-4 solution. A second case arises when we follow a path from position 4 via some other tuple vertex (or vertices) currently in S_{top-3} in the bipartite graph, to a tuple which is currently unmatched. Consider for example the path from position 4 to tuple vertex t_3 to position 2 and back to tuple t_4 (which is currently unmatched). This path defines a new candidate solution which can be obtained by modification of S_{top-3} by removing $(t_3, 2)$ and adding $(t_3, 4), (t_4, 2)$ with a total cost $C_{top-3} + \sigma^r(t_3, 4) + \sigma^r(t_4, 2) - \sigma^r(t_3, 2)$.

```

CREATE PROCEDURE MHA(IN K integer, IN N integer)
LANGUAGE SQL
BEGIN
    – Initialization for MHA
    1. SET pivot = N;
    WHILE (pivot > K) DO
        insert into Top values(pivot, pivot);
    SET pivot = pivot - 1;
    END WHILE;

    SET pivot = K;
    pivotiter:

    WHILE (pivot ≥ 1) DO
    2 delete from SimpleAugment;
    insert into SimpleAugment
    select Graph.elem, pivot
    from Graph
    where pos = pivot and cost = 0
    and elem not in (select elem from Top)
    fetch first 1 rows only;
    select count(*) into rcount from SimpleAugment;

    IF (rcount > 0) THEN
        insert into Top
        select elem, pos
        from SimpleAugment;
        set pivot = pivot - 1;
    iterate pivotiter;
    END IF;

    3. delete from Reach4;
    insert into Reach4(elem, parent)
    select Graph.elem, N+1
    from Graph
    where pos = pivot and cost = 0;
    SET cold = -1;
    SET found = 0;
    select count(*) into cnew from Reach4;

    WHILE (found = 0 AND cnew > cold) DO
    SET cold = cnew;
    insert into Reach4(elem, parent)
    select Graph.elem, min(Reach4.elem)
    from Reach4, Top, Graph
    where Top.elem = Reach4.elem
    and Top.pos ≤ K
    and Graph.pos = Top.pos and Graph.cost = 0
    and Graph.elem not in (select elem from Reach4)
    group by Graph.elem;
    insert into Reach4(elem, parent)
    select GTail.elem, min(Reach4.elem)
    from Reach4, Top, GTail
    where Top.elem = Reach4.elem
    and Top.pos > K
    and GTail.pos = COMMONCOL and GTail.cost = 0
    and GTail.elem not in (select elem from Reach4)
    group by GTail.elem;

    select count(*) into cnew from Reach4;
    select count(*) into found
    from Reach4
    where Reach4.elem not in (select elem from Top);
    END WHILE;

    4. delete from Augment;
    insert into Augment
    select Reach4.elem
    from Reach4
    where Reach4.elem not in (select elem from Top)
    fetch first 1 rows only;
    select count(*) into rcount from Augment;

    IF (rcount > 0) THEN
        delete from AugmentingPath1;
        insert into AugmentingPath1
        with AugmentingPath(elem, parent, level) as (
            select Reach4.elem, Reach4.parent, 0
            from Reach4, Augment
            where Reach4.elem = Augment.elem
            UNION
            select Reach4.elem, Reach4.parent,
            AugmentingPath.level+1
            from Reach4, AugmentingPath
            where Reach4.elem = AugmentingPath.parent
            and AugmentingPath.level < N)
        select elem, parent
        from AugmentingPath;
        insert into Top values(N+1, pivot);

        delete from TempTop;
        insert into TempTop
        select AugmentingPath1.elem, Top.pos
        from Top, AugmentingPath1
        where Top.elem = AugmentingPath1.parent;

        delete from Top
        where elem in
        (select parent from AugmentingPath1);
        insert into Top
        select * from TempTop;

    SET pivot = pivot - 1;
    iterate pivotiter;
    END IF;

    5. select min(cost) into elemposmin
    from Graph
    where elem not in
    (select elem from Reach4) and (
        pos = pivot or pos in (
            select Top.pos from Top, Reach4
            where Top.elem = Reach4.elem))
    and pos ≤ K;

    update Graph
    set cost = cost - elemposmin
    where elem not in (select elem from Reach4) and (
        pos = pivot or pos in
        (select Top.pos from Top, Reach4
        where Top.elem = Reach4.elem))
    and pos ≤ K;

    update Graph
    set cost = cost + elemposmin
    where elem in (select elem from Reach4) and (
        pos != pivot and pos not in (
            select Top.pos from Top, Reach4
            where Top.elem = Reach4.elem))
    and pos ≤ K;

    update GTail
    set cost = cost + elemposmin
    where elem in (select elem from Reach4);
    END WHILE;
    delete from Top
    where pos > K;
    END

```

Figure 6: *MHA* in Procedural SQL

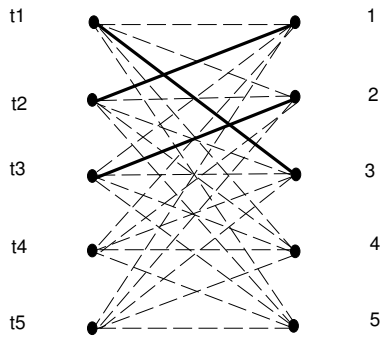


Figure 7: Example Matching

Notice that if more than one already matched vertices in S_{top-3} are present in the path a similar expression for the total cost can be obtained by adding to C_{top-3} the ranking costs associated with the edges that are not present in S_{top-3} and subtracting the ranking costs associated with the edges present in S_{top-3} . The top-4 solution can be derived by identifying the path with the lowest cost, among all possible paths from position 4 to an un-matched tuple vertex and making the suitable modifications to S_{top-3} in order to derive S_{top-4} .

One observation from the above discussion is that during the construction of the top- i solution the entire graph G is not required. The computation can be correctly performed by manipulating G_i , the bipartite subgraph of G corresponding to the first i positions. Paths involving positions greater than i will be considered at later steps, incrementally towards k .

The above example leads to a natural procedure, the *successive shortest paths (SSP)* algorithm to incrementally evaluate the answer to a top- k selection problem. The algorithm is presented in pseudo-code in figure 8. Starting from a new position i the algorithm obtains the subgraph G_i of interest at this iteration, identifies the transitive closure to un-matched tuple vertices in graph G_i and maintains the one with the cheapest cost, accounting for positive edge costs when the edge is not present in solution S_{i-1} and negative edge costs when it is present. Finally, it adjusts solution S_{i-1} to obtain the new top- i solution, incrementally up to the desired value of k .

The incremental construction of the top- k answer provided by algorithm *SSP* in figure 8 can be optimized. We introduce an optimization of this basic scheme that can significantly reduce the computation involved. We refer to this optimization as *Early Stopping (ES)*. It provides a criterion for terminating the search for the least cost path early on in the search and subsequently avoiding the computation of the transitive closure.

Early Stopping:

During the incremental construction of the top- k answer, algorithm *SSP* considers positions successively. When a new position is considered the paths to all un-matched tuples are computed. It is easy to show that if at iteration $i + 1$ of the algorithm the edge of minimum ranking cost incident to position $i + 1$ is also incident to an un-matched vertex tuple, then this edge belongs to the top- $(i + 1)$ answer:

```

Algorithm SSP(k) {
  for (i = 1 to k) {
    obtain subgraph  $G_i$  of  $G$ 
    if (i == 1) {
      let  $t_i$  the tuple vertex incident to the cheapest
      edge at position 1; add  $(t_i, 1)$  to  $S_1$ 
    }
    else {
      1) Starting from position  $i$  compute the
      transitive closure to all currently
      un-matched vertex nodes in  $G_i$ 

      2) Compute the cost of each path in the
      closure accounting for the negative edge
      cost when traversing an edge in  $S_{i-1}$  and
      positive edge cost when the edge is not in  $S_{i-1}$ 

      3) Identify all edges in the cheapest path  $e_1, \dots, e_m$ 
      and modify  $S_{i-1}$  (add/remove edges) accordingly to
      obtain  $S_i$ 
       $S_i = S_{i-1}$ 
      for (j = 1 to m) {
        if  $e_j \in S_{i-1}$  then  $S_i = S_i / e_j$ 
        else  $S_i = S_i \cup e_j$ 
      }
    }
  }
}

```

Figure 8: *SSP* pseudo code

Optimization 1 (Early Stopping) *At iteration $i + 1$, if the cheapest edge out of the $(i + 1)$ st position is unmatched then the top- $(i + 1)$ answer contains this edge. ■*

This optimization enables us to terminate the search process for the top- $(i + 1)$ answer immediately, without having to evaluate the transitive closure from the $(i + 1)$ -th position. In this case the top- $(i + 1)$ answer can be obtained by extending the top- i answer by the edge of smaller cost out of position $i + 1$ to an un-matched tuple vertex.

3.3.1 SQL Statements for *SSP*

Figure 9 presents the *SSP* algorithm in procedural SQL (without any optimization introduced). The schema required by this procedure is omitted due to space constraints; it can be easily derived from the procedure body. Block 1 initializes the Top relation to the cheapest edge out of position 1. We then iterate over all positions from 2 to k . Within each iteration, block 2 computes the transitive closure from the current position to all unmatched positions. Block 3 picks an unmatched tuple with the smallest distance from the current position and block 4 computes the path from the current position to this unmatched tuple. Finally, block 5 updates the old solution using this path to obtain the new solution which includes the current position.

Early stopping can be also incorporated by inserting the statements of figure 10 in the sequence of the statements of figure 9 immediately after the first *WHILE* loop. The statements in figure 10 identify whether the least cost tuple vertex which is a target of the current position, is in the current solution. If not, then they augment the current solution and start a new iteration.

```

CREATE PROCEDURE ssp(IN k integer, IN N integer)
LANGUAGE SQL
BEGIN
DECLARE index INTEGER DEFAULT 1;
DECLARE iter INTEGER DEFAULT 1;
DECLARE rcount INTEGER;
DECLARE matchsum FLOAT DEFAULT 0;

1. delete from Top;
insert into Top(elem, pos)
select min(elem), pos
from graph
where pos = 1 and cost =
(select min(cost)
from graph where pos = 1)
group by pos;

SET index = 2;
start:
WHILE (index ≤ k) DO
2. delete from R1;
insert into R1(selem, delem, cost)
select Top.elem, G2.elem, G2.cost - G1.cost
from Top, Graph G1, Graph G2
where Top.elem = G1.elem and Top.pos = G1.pos
and Top.pos = G2.pos;

delete from Reach4;
insert into Reach4(elem, parent, cost)
select elem, N+1, cost
from Graph
where pos = index;

SET iter = 1;
WHILE (iter ≤ index) DO
delete from RR;
insert into RR(elem, parent, cost)
select Y.elem, X.elem, X.cost + R1.cost
from Reach4 X, R1, Reach4 Y
where X.elem = R1.selem and Y.elem = R1.delem
and X.cost + R1.cost < Y.cost;

delete from R41;
insert into R41(elem, cost)
select elem, min(cost)
from RR
group by elem;

delete from Reach4
where elem in (select elem from R41);
insert into Reach4(elem, parent, cost)
select RR.elem, min(RR.parent), RR.cost
from RR, R41
where RR.elem = R41.elem
and RR.cost = R41.cost
group by RR.elem, RR.cost;

SET iter = iter+1;
END WHILE;
3. delete from Augment;
insert into Augment
select elem
from Reach4
where elem not in
(select elem from Top)
order by cost
fetch first 1 rows only;

4. delete from AugmentingPath1;
insert into AugmentingPath1
with AugmentingPath(elem, parent, level) as (
select Reach4.elem, Reach4.parent, 0
from Reach4, Augment
where Reach4.elem = Augment.elem
UNION
select Reach4.elem, Reach4.parent,
AugmentingPath.level+1
from Reach4, AugmentingPath
where Reach4.elem = AugmentingPath.parent
and AugmentingPath.level < N)
select elem, parent
from AugmentingPath;

5. insert into Top values(N+1, index);
delete from TempTop;
insert into TempTop
select AugmentingPath1.elem, Top.pos
from Top, AugmentingPath1
where Top.elem = AugmentingPath1.parent;
delete from Top
where elem in
(select parent from AugmentingPath1);
insert into Top
select * from TempTop;
SET index = index+1;
END WHILE;
END

```

Figure 9: *SSP* in Procedural SQL

4 Experimental Evaluation

In this section, we present the results of an experimental case study of the proposed algorithms in a real application scenario, varying various parameters of interest in order to understand the algorithms' comparative performance.

4.1 Implementation

We first provide implementation details on the realization of the algorithms in an RDBMS and some observations on the performance of various SQL constructs and then we detail our experimental case study. Our experiments were conducted on DB2 V8.1 Personal Edition running on a Dell PowerEdge server P2600 with two Intel Xeon processors having 3GB of memory and 400 GB of disk. We maintained the default configuration parameters DB2 ships with, only increasing the transaction log size to 524 MB.

The procedural SQL statements of figures 6 and 9 use

recursive SQL statements with UNION semantics as specified in the SQL3 standard. However, no major RDBMS up to date supports an efficient implementation of this construct. In particular, DB2 V8.1 supports this construct only with UNION ALL semantics. As a result, since no duplicate elimination takes place at various stages of the recursion, the number of intermediate tuples generated is very large, and performance is affected. In order to alleviate this problem, we simulated the effects of recursion as follows: instead of using the recursive statement directly, we embedded the join clauses in an iterative statement, issuing duplicate elimination statements after each iteration. This resulted in great performance benefits for all the approaches discussed in this paper. We note that as RDBMSs start implementing recursion with SQL3 semantics, the need for such workarounds will decrease.

```

delete from CheapElem;
insert into CheapElem(elem)
select elem
from graph
where pos = index
and cost = (select min(cost)
from graph where pos = index
and elem not in
(select elem from Top)
fetch first row only;

select count(*) into rcount from CheapElem;
IF (rcount > 0) THEN
insert into Top(elem, pos)
select graph.elem, graph.pos
from graph, CheapElem
where graph.elem = CheapElem.elem
and graph.pos = index;
SET index = index+1;
iterate start;
END IF;

```

Figure 10: Enabling Early Stopping in *SSP*

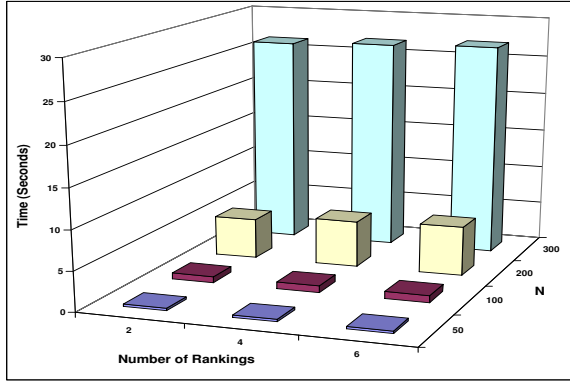


Figure 11: Construction Time for Graph G as a function of n and the number of rankings

4.2 Experimental Case Study

We implemented all three algorithms as outlined using a commercial RDBMS and we conducted experiments to evaluate their performance. In our experiments we used a database containing real customer data with various associated attributes. We utilized the technique of Gravano et al. [18] on various attributes to obtain approximate matches. On a table containing seven million rows it required, in our implementation, approximately 20 seconds to perform an approximate match on an attribute with average length 17 characters and approximately 25 seconds on an attribute with average length 28 characters.

In our first experiment, we seek to quantify the time required to construct the bipartite graph G on which the various techniques operate to provide answers to top- k selection problems. Figure 11 presents this time for different number of rankings and different ranking sizes. The construction time appears more sensitive to the size of the rankings as opposed to the number of rankings used. Notice that, once such a graph is constructed for some value of n (size of the rankings) it can be utilized to answer top- k selection queries for any $k \leq n$.

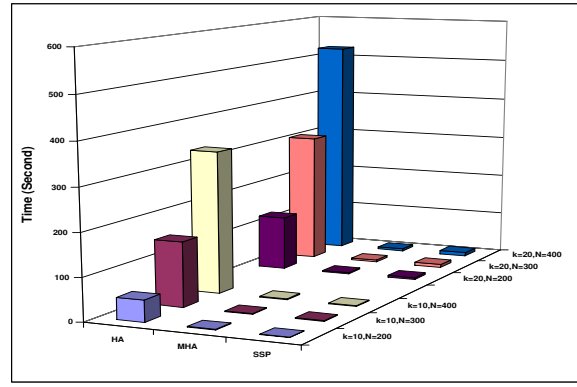


Figure 12: Performance of *HA*, *MHA*, *SSP* as a function of n and k

Our second experiment compares the performance of the three algorithms. Figure 12 presents their performance, as the size of the rankings increases for two distinct values of k . It is evident that the overhead of algorithm *HA* is prohibitive and its scalability is limited. We chose to keep the scale of this experiment small, as the performance of *HA* quickly deteriorated. Algorithm *MHA* and *SSP*, in contrast, appear much faster.

This prompted us to conduct a more detailed experiment, comparing the performance of *MHA* and *SSP*. The result of this experiment is presented in figure 13, for varying values of k and n (the x -axis in figure 13 presents pairs of k, n values). In particular, we vary n from 200 to 1800 and for each value of n we present performance results for four values of k (10,20,30,40). A main observation from this experiment is that there exists a crossover point in the performance of the algorithms which depends on the relative size of k to n . For example, for small values of n (up to 600 in figure 13) *SSP* offers performance advantages over *MHA* for values of k up to 10. In that range, *SSP* is more than 50% faster than *MHA*. For values of k above 10, *MHA* is faster than *SSP*, especially as k increases. As the value of n increases (beyond 600 in figure 13) this crossover point is experienced at a value of k between 20 and 30. In these cases, when k is below 20, *SSP* is up to three times faster than *MHA*, especially for large n . For values of k greater than 30, *MHA* is clearly the algorithm of choice.

These observations lead us to the conclusion that for small values of k comparative to n *SSP* is the algorithm of choice. As the value of k increases, compared to n , *MHA* offers great performance advantages. Overall, if only the first few top ranking results are required then algorithm *SSP* appears the algorithm of choice. For larger values of k , algorithm *MHA* should be used.

5 Dealing With Scores

Up until now, for simplicity of exposition, we have presented techniques to merge ranked lists, where the list tuples did not have associated scores. Approximate matching techniques used in data cleaning, however, typically associate values matching a query string with a score quantifying the degree of similarity (closeness) of the query string

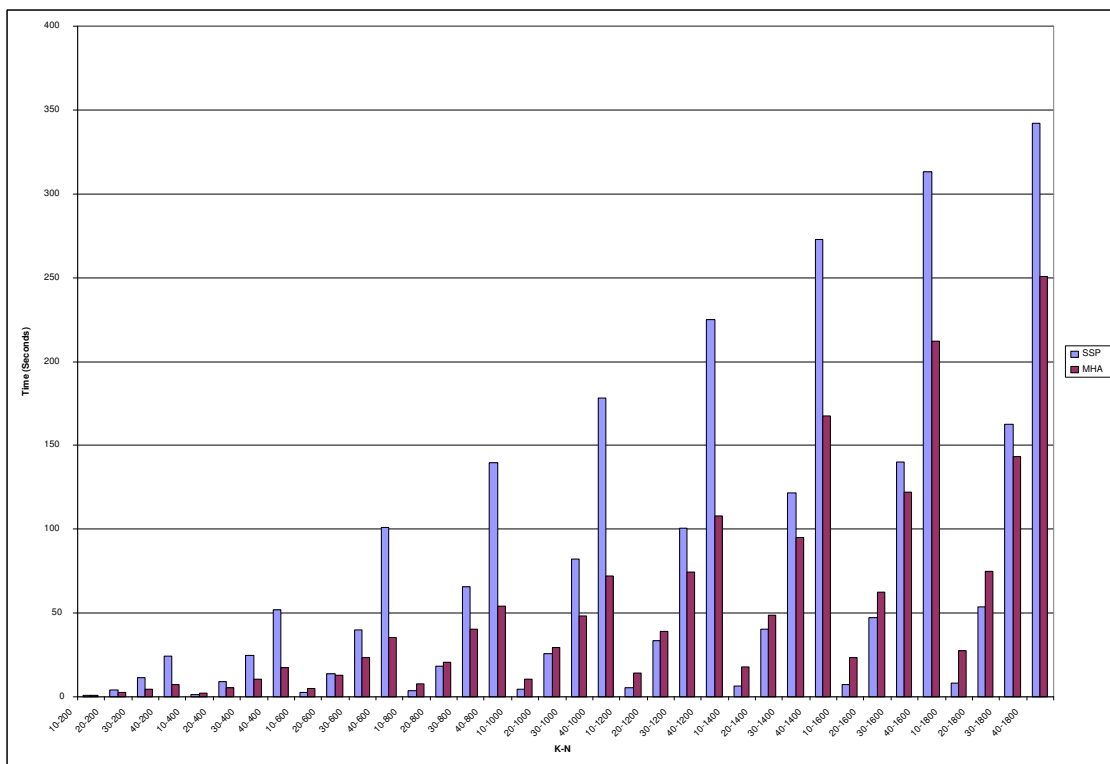


Figure 13: Comparative Performance of *MHA* and *SSP* as a function of n and k

to the attribute value string. These scores thus carry more information than the positions of tuples in the ranked lists. In such situations, it would be desirable for (a) the merging technique to use the scores (and not just the rankings) of the tuples in each of the lists to determine the tuple ranks in the merged list, and (b) the tuples in the merged list to be associated with scores, to preserve the closure property. We discuss both these issues in this section.

5.1 Merging Scored Rankings

Essentially, the basis for merging ranked lists discussed in previous sections was to minimize the overall discrepancy in rank positions of the individual tuples, as quantified by the total ranking cost, where the ranking cost of tuple t_i wrt position j was given by $\sigma^r(t_i, j) = \sum_{\ell=1}^m |\tau_\ell(t_i) - j|$. This ranking cost was used as the edge cost between tuple t_i and position j in the bipartite graph, as described in section 3, and the solution to our top- k selection problem was based on the minimum cost perfect matching problem over this bipartite graph.

Carrying this analogy to use the scores associated with each of the tuples in the input ranked lists, we define the *scored ranking cost* of tuple t_i wrt position j as

$$\sigma^s(t_i, j) = \sum_{\ell=1}^m |\tau_\ell^s(t_i) - s_\ell(j)|$$

where $\tau_\ell^s(t_i)$ is the score associated with tuple t_i in the scored ranked list τ_ℓ , and $s_\ell(j)$ is the score associated with the tuple in the j^{th} rank position in list τ_ℓ . The bipartite graph can now be defined as before, except that the

edge costs are given by the scored ranking costs $\sigma^s(t_i, j)$, instead of the ranking costs $\sigma^r(t_i, j)$. Of course, for the absolute score differences to be meaningfully comparable across lists, the scores in each of the lists would need to be normalized to, say, be a value in $[0, 1]$. This was done in the example of figure 1.

Our techniques, *MHA* and *SSP*, can now be used *unchanged* on the resulting bipartite graph to identify solutions to the top- k selection problem, taking scores into account. The rankings in the merged list obtained by using scores may be different from those obtained by simply using the input rankings. It is, however, noteworthy that if the scores associated with the tuples in the input ranked lists were *uniformly* distributed in accordance with their rank positions (for example, the tuple in rank position j had a score of $(n - j + 1)/n$), the final rankings in the merged list would be identical.

5.2 Computing Scores in the Merged Ranking

Just as scores are useful when provided by the approximate matching techniques used in data cleaning, they can be useful in the result of the merging. However, the scores that are associated with the tuples in the merged list need to satisfy some robustness properties. First, the scores need to be consistent with the rankings, i.e., there should not be any inversions between the rank order and the score order. Second, the merged scoring should be idempotent, i.e., if two identical scored ranked lists were merged, the score of each tuple in the merged list should be identical to its scores in

the input lists.² Third, the score of a tuple in the j^{th} position of the merged list can be no larger than the average score of the tuples in the j^{th} positions of the m input lists. These robustness properties eliminate some obvious candidates, such as computing the score of a tuple in the merged list as the average of its scores in the m input lists.

We describe next an intuitive scoring function that preserves the above robustness properties. Let $sa(j)$, $1 \leq j \leq k$, be the average score of the tuples in the j^{th} positions of the m input lists. Let $dca(j) = sa(j) - sa(j-1)$, $2 \leq j \leq k$; this is the difference in the average scores between consecutive positions in the input. Next if, in the merged ranking, tuple t_i is at position j , let $srca(j) = \sigma^s(t_i, j)/m$; this is the average (over the m input lists) of the score differences used in the scored ranking costs in the solution to the top- k problem. Then, the scores of the tuples at rank positions j in the merged list are computed as follows:

$$score(j) = sa(j) - srca(j), j = 1$$

$$score(j) = score(j-1) - \max(dca(j), srca(j)), j > 1$$

It is easy to verify that the scores computed as described above satisfy the above robustness properties. A more detailed discussion on this merged-scoring function is outside the scope of this paper.

6 Related Work

Data cleaning has attracted lots of research attention in recent years [16, 21, 17, 18, 4, 19, 7]. Most of the recent works [19, 4, 7] deal with various aspects of the classic record linkage problem [15], while others aim to offer a declarative framework for cleaning and approximate matching tasks [16, 17, 18]. Various string distance metrics have been proposed for quantifying approximate string matches including string edit distance, cosine similarity [17, 18, 4] and combinations and extensions thereof [6, 5].

Various functions for merging ranked data have been proposed and studied in the statistical literature [10, 8, 20], including the *Spearman's footrule* and *Kendall's tau*. Such functions have been utilized in the problem of merging the results of various search engines [11]. Variants of such measures have also been considered for similarity search and classification [14]. The metric properties for a large class of functions merging ranked lists have also been studied [13]. We are not aware of any work addressing issues of realization of such merging functions in SQL. A different approach to merging scored lists is to use a combining rule, such as min or average, that combines individual scores to obtain an overall score; this approach has been well investigated in recent years (see, e.g., [12] and references therein).

7 Conclusions

We have considered the problem of merging rankings produced as a result of approximate match operations in relational databases, with the objective of identifying a consensus ranking (under specific metric merging functions) and identifying a few top ranking results. In this context, we

introduced the top- k selection problem for which we have identified and proposed applicable algorithms providing their full declarative specification. Our experimental case study using real application data, identified the cases under which two of the proposed algorithms, namely *MHA* and *SSP*, are beneficial.

Such problems are of profound interest in practical data cleaning scenarios and we believe research in this direction is well warranted. Future work could investigate incorporation of approximations (e.g., in the spirit of [14]) in a declarative (SQL) framework and quality/performance issues arising in this setting. Moreover, being able to efficiently perform such merging operations in bulk, when a set of queries is provided, is an interesting open question.

References

- [1] MindBox Inc. www.mindbox.com.
- [2] R. K. Ahuja, T. Magnanti, and J. Orlin. *Network flows*. Prentice Hall, 1992.
- [3] V. Borkar, K. Deshmukh, and S. Sarawagi. Automatic segmentation of text into structured records. *Proceedings of SIGMOD*, 2001.
- [4] W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. *Proceedings of SIGMOD*, 1998.
- [5] W. Cohen, P. Ravikumar, and S. Fienberg. A comparison of string distance metrics for name-matching tasks. *Proceedings of IWeb Workshop*, Aug 2003.
- [6] W. Cohen, P. Ravikumar, and S. Fienberg. A comparison of string metrics for matching names and records. *Proceedings of KDD Data Cleaning Workshop*, Aug 2003.
- [7] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. *Proceedings of SIGMOD*, 2003.
- [8] D. Critchlow. *Metric methods for analyzing partially ranked data*. LNS, Springer-Verlag, 1985.
- [9] T. Dasu and T. Johnson. *Exploratory data mining and data cleaning*. John Wiley, 2003.
- [10] P. Diaconis and R. Graham. Spearman's footrule as a measure of disarray. *J. of the Royal Statistical Society*, 39(2), pages 262–268, 1997.
- [11] C. Dwork, R. Kumar, M. Naor, and D. Shivakumar. Rank aggregation methods for the web. *Proceedings of WWW10*, 2001.
- [12] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Proceedings of PODS*, 2001.
- [13] R. Fagin, R. Kumar, and D. Shivakumar. Comparing top- k lists. *Proceedings of SODA*, 2003.
- [14] R. Fagin, R. Kumar, and D. Shivakumar. Efficient similarity search and classification via rank aggregation. *Proceedings of SIGMOD*, 2003.
- [15] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328), pages 1183–1210, Dec. 1969.
- [16] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and E. Saita. Declarative data cleaning. *Proceedings of VLDB*, 2001.
- [17] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. *Proceedings of VLDB*, 2001.
- [18] L. Gravano, P. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an RDBMS for web data integration. *Proceedings of WWW*, 2003.
- [19] M. Hernandez and S. Stolfo. The merge purge problem for large databases. *Proceedings of SIGMOD*, 1995.
- [20] J. Marden. *Analyzing and modeling rank data*. Chapman Hall, 1995.
- [21] S. Sarawagi. Special issue on data cleaning. *IEEE Data Engineering Bulletin*, 23(4), 2000.
- [22] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. *Proceedings of KDD*, 2002.

²Note that rank merging satisfies this idempotence property.