Top-*N* Query: Query Language, Distance Function and Processing Strategies

Yuxi Chen, Weiyi Meng

Department of Computer Science, State University of New York at Binghamton
Binghamton, NY 13902, USA, meng@cs.binghamton.edu

Abstract. The top-*N* query problem is to find the *N* results that satisfy the query condition the best but not necessarily completely. It is gaining importance in relational databases and in e-commerce where services and products are sold on the Internet. This paper addresses three important issues related to the top-*N* query problem in a relational database context. First, we propose a new query language to facilitate the specification of various top-N queries. This language adds new features to existing languages. Second, we make a case that the *sum* function is a more appropriate distance function for ranking tuples when attributes involved in a top-N query are incomparable. Third, based on the *sum* distance function, we discuss how to process top-*N* queries.
Keywords: Top-*N* query, query language, query processing, distance function.

1 Introduction

In recent years, there has been increasing interest in the top-*N* query problem in relational databases [3, 5, 9]. For a given query against a database table, the problem is how to efficiently find the *N* tuples that satisfy the query condition the best but not necessarily completely.

Example 1. Consider a table *Employees* (*Name*, *Age*, *Salary*). Suppose we want to find young employees who earn a good salary. The query cannot be directly specified using the standard SQL. If SQL must be used, a user may write a query as follows:

SELECT *Name* FROM *Employees* WHERE *Age* < *30* AND *Salary* > *50000*

Suppose there are four employees: *A* = (*Dave, 28, 70000), B* = (*Nicky, 29, 60000), C = (Randy, 31, 48000)* and *D = (Karen, 32, 47000).* Obviously, both *A* and *B* satisfy the query condition but *A* could be interpreted as satisfying the condition better because *Dave* is younger and earns more money than *Nicky*. In addition, neither *C* nor *D* satisfies the query condition but *C* can be considered to satisfy the condition better than D. Therefore, for "finding young employees who earn a good salary", the tuples should be ordered by {*A*, *B*, *C, D*}. If top 1 or top 3 tuples are desired, then {*A*} or {*A*, *B*, *C*} should be selected respectively.

Many applications can benefit from the capability of top-*N* queries. For example, when searching for a used car or a house to buy, it is desirable to have the results ranked based on how well a car or a house matches with a user-provided specification and have only certain number of top results displayed.

There are a number of research issues in the top-*N* query problem. (1) How to specify top-*N* queries? Appropriate extensions to SQL are needed to allow top-*N* queries to be expressed. (2) How to rank the results? If a condition involves only one attribute, this is trivial. When multiple attributes are involved, the problem becomes more complex. Suppose, for example, if the tuple *D* above is changed to (*Karen, 32, 49000*), it is less easy to rank *C* or *D*. (3) How to efficiently evaluate top-*N* queries?

In this paper, we report our research on the above three issues. Our contributions are summarized below. For issue (1), we propose significant extensions to SQL so that a variety of top-N queries can be specified. New operators and new concepts are introduced to better capture new semantics to top-*N* queries. For issue (2), we make a case for the *sum* function for ranking tuples for a given query. While several distance functions, such as Euclidean distance function and the *sum* functions [3, 7], have been used for top-N applications, there is little discussion on the appropriateness of a particular function. We argue that the *sum* function is appropriate when attributes involved in a query are not comparable. For issue (3), we provide a new strategy for the efficient processing of top-*N* queries when the *sum* scoring function is used. The objective is to minimize the size of the search space. This strategy is suitable when the data are approximately uniformly distributed.

1.1 Related Work

Here we briefly review some existing work related to the issues mentioned above.

1. *Syntax of Top-N Queries*: In [4], the ORDER BY clause was extended to allow a distance function to be specified for ranking tuples and a STOP AFTER clause was added to specify how many tuples are desired in the result. However, only tuples that satisfy the query conditions will be ranked. Thus, if *N* tuples are desired but fewer than *N* tuples satisfy the WHERE clause of a query, then less than N tuples will be returned. In [3], the ORDER BY clause was extended to allow both the desired number of tuples and the distance function to be specified (i.e., order *N* by score). However, only equality conditions are considered, i.e., no range condition is considered. We believe more general query conditions should be supported. In [7], general conditions are allowed and tuples satisfying the condition are always ranked ahead of tuples not satisfying the condition. Again, the ORDER BY clause was extended to allow a distance function to be specified. [7] also supports different priorities for different conditions in the same query to be specified. None of the existing extensions to SQL for top-*N* query specification has the flexibility and expressiveness of our proposed language.
2. *Distance Function*: The condition C of a top-*N* query may involve multiple attributes. For a given tuple and a given attribute involved in C, a distance measuring how well the tuple satisfying C can be obtained easily. The problem here is how to combine the distances on individual attributes into a combined distance so that all tuples can be ranked. Several distance functions for combining purpose have been mentioned and they include the *Euclidean* distance function [3,7], the *sum* (or *Manhattan*) distance function [3,7] and the *min* function [3,6]. Current proposals have two weaknesses. First, users are either given too much control or not enough flexibility. In several proposals, the user is required to provide the distance function as part of a query [3,4,7]. In practice, most ordinary users don't know what distance function is appropriate to use. In other proposals, a pre-determined distance function is used and users do not have any choice [6]. A single function is unlikely to be suitable for different applications. Our approach is a middle of the road approach. We allow a user to provide his/her preferred distance function if he/she knows what function is appropriate for his/her application. However, a default distance function will be used by the system if a user does not provide his/her own function. Second, there is a lack of in-depth

study on why a particular function should be used. In this paper, we argue that the *sum* function is reasonable when the involved attributes are not comparable.

3. Top-N Query Processing: There have been several proposals in the literature (e.g., [3,7]). Most of them are working on how to determine the search range by exploiting the statistics available to a relational DBMS, and the impact of the quality of these statistics on the retrieval efficiency of the resulting scheme. In this paper, we propose a simple approach that is suitable when the data are approximately uniformly distributed.

The remainder of this paper is organized as follows. In Section 2, we propose a new query language called topNSQL for specifying top-N queries. Only conjunct conditions are considered in this paper. In Section 3, we make a case that the *sum* distance function is more appropriate for top-N queries when attributes involved in the query condition are incomparable. In Section 4, we propose a new strategy for processing top-N queries based on the *sum* function. Section 5 concludes the paper.

2 Extending SQL

In this section, we discuss different possible semantics of top-*N* queries. Based on the discussion, we propose our extensions to the syntax of standard SQL so that all possible semantics can be represented.

Regular operators and emphatic operators

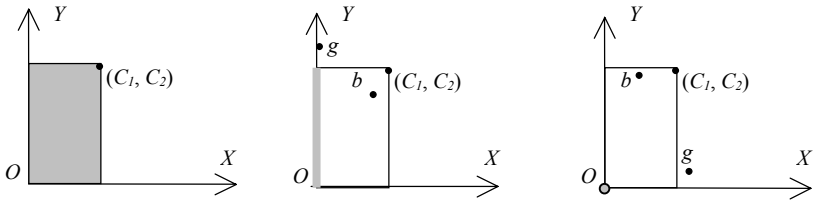
Example 2. Let *price* be an attribute of Books. In standard SQL, a simple condition 'price < 20' indicates that the user is interested in books that cost less than $20. Now consider how to interpret this condition in the context of top-N queries. There are two reasonable interpretations. (1) All books less than $20 are equally good. If N or more books satisfy the condition, arbitrarily select N of them to return to the user. If there are less than N such books, then select additional books whose prices are closest to $20 but higher. In this case, interval [0, 20) is used as a *reference space* to calculate the distance of a book with the query. Specifically, let *p* be the price of a book *b*. If $0 \leq p < 20$, then the distance of *b* is zero; if p >= 20, then the distance of b is p – 20. (2) Cheaper books are better, regardless of whether the price is less than $20 or not. Based on this interpretation, 0 is the *reference point* for calculating the distances of books.

Both interpretations discussed above are reasonable and can be useful in practice. Therefore, it is important to differentiate the two interpretations syntactically. We propose to use 'price < 20' for the first interpretation only, consistent with the standard SQL. For the second interpretation, we propose to use 'price << 20'. In other words, the operator "<<" has the meaning of "the smaller, the better". Similarly, ">>" can be used to represent "the larger, the better". For easy discussion, the standard operators such as '<' and '<=' will be called *regular operators* while new operators '<<', '<<=', '>>' and '>>=' will be called *emphatic operators*. Another interesting observation that can be made from Example 2 is that the reference point for calculating the distances of tuples for ranking purposes is dependent on the interpretation.

Condition space and reference space

For a given query Q, let the space defined by the condition of Q be called the *condition space* of Q. Furthermore, if a condition space is a point, the condition is called a *point condition*. For example, 'weight = 120 and height = 170' is a point condition as (120, 170) represents a point in the two-dimensional space of (weight,

height). If a condition space is a rectangle, the condition is called a *rectangle condition*. An example of a rectangle condition is 'weight < 120 and height >= 170'. For the purpose of defining the condition space of a query, regular operators and emphatic operators are not differentiated.



(a) X < C1 and Y < C2 (b) X << C1 and Y < C2 (c) X << C1 AND Y << C2

Figure 2.1: Example Condition Spaces and Reference Spaces

For query Q, its *reference space* needs to be derived for ranking tuples. The distance of a tuple for Q is the distance between the point corresponding to the tuple and the reference space of Q. The reference space of Q is determined by the operators used in the condition specification and may be different from the condition space of Q. In Example 2, for 'price < 20', the condition space and the reference space are both [0, 20) (note that for any price p >= 20, the distance between p and [0, 20) is the same as that between p and 20). For 'price << 20', the condition space is still [0, 20) but the reference space becomes [0, 0]. In general, for a given query, when emphatic operators are not used, the condition space and the reference space are the same, and when emphatic operators are used, the condition space and the reference space are different. When one or more emphatic operators are involved, the reference space is derived as follows. Let each attribute represent one dimension in the condition space. Let min(X) and max(X) be the smallest and the largest possible values of attribute X. If 'X << v' or 'X <<= v' appears in the condition, where v is a constant, then the reference space is obtained by collapsing the condition space along the dimension of X to min(X) (i.e., the X-dimension becomes a point, min(X)); If 'X >> v' or 'X >>= v' appears in the condition, the reference space collapses along the X-dimension to max(X). In Figure 2.1, three condition spaces and three reference spaces for three different query conditions are drawn. For all three conditions, the condition space is the same rectangle (0<=X<C1, 0<=Y<C2), but the three reference spaces are different (see shaded areas).

Distance factor and satisfaction factor

The above discussion can be summarized as follows. From a given query condition, a *condition space* and a *reference space* can be determined. Both types of spaces can affect the ranking of a tuple as discussed below.

1. *Distance factor*. Each tuple has a distance with respect to the reference space of the query. Tuples should be ranked in ascending distance values.
2. *Satisfaction factor*. Tuples satisfying the condition (i.e., falling into the condition space) should be ranked ahead of tuples not satisfying the condition. In this case, tuples satisfying and not satisfying the condition can be ordered separately by a distance function, and the tuples in the first group are placed ahead of those in the second group.

The following example illustrates that the above two factors are not always consistent when rectangle conditions are involved. It is possible that a tuple satisfying the query condition has a larger distance than a tuple not satisfying the condition.

Example 3. Consider a book search service that can find books for customers to buy from multiple sources. Suppose a user wants to find a book (of a given title) with the lowest total cost (cost of the book plus shipping fee), but the user is willing to pay up to $45 for the book. As the query condition must be specified based on the cost of the book and the shipping fee, the user breaks $45 into $40 (for cost of the book) and $5 (for shipping). As a result, the query condition involving the attributes *cost* and *shipping_fee* becomes 'cost <<= 40 and shipping_fee <<= 5'. For this query, the condition space is (0<=cost<=40, 0<=shipping_fee<=5) while the reference space is the point (0, 0). Consider two books, B1 and B2, of the same title from two sources. B1 sells for $39 with a $5 shipping fee and B2 sells for $41 with a $2 shipping fee. According to the satisfaction factor, B1 should be ranked ahead of B2. But according to the distance factor (obviously the appropriate distance function for this example should be *sum* and the distance should be with respect to (0, 0)), B2 should be ranked ahead of B1. Also in Figure 2.1 (b) and (c), point *b* satisfies the query condition while point *g* does not, but *g* has a smaller distance to the query than *b*.

In general, there are applications that prefer to rank tuples strictly by distances and there are also applications that would like to take the satisfaction factor into consideration. We believe that both types of applications should be supported. To the best of our knowledge, no existing research has pointed out the inconsistency between the satisfaction factor and the distance factor. Indeed, such an inconsistency does not exist when the query condition is a point condition. For the rest of this paper, top-N queries that rank tuples strictly by distances will be called *Type-1* queries and those that take the satisfaction factor into consideration are called *Type-2* queries.

Importance factor of a condition

Conditions of the same query may have different importance for ranking tuples.

Example 4. Suppose a user wants to purchase a used car worth no more than $10,000 with the mileage no more than 10,000. Using our notation, this query is expressed as "price <<= 10000 and mileage <<= 10000". The reference space of the query is (0, 0). Consider two used cars C1 = ($11,000, 10,000 miles) and C2 = ($10,000, 11,000 miles). The distance between C1 and the reference space of the query for price is 11,000 and the distance between C2 and the reference space of the query for mileage is also 11,000. In practice, C2 should be ranked ahead of C1 because one-dollar difference is more significant than one-mile difference. To reflect the difference of different conditions in importance, appropriate *importance factors* can be associated to these conditions [7]. For example, if one-dollar difference is equivalent to ten-mile difference, we can multiply the distance for mileage by 10 and multiply the distance for price by 1, where 10 and 1 are the importance factors. (Note: Since smaller distance is better, a smaller importance factor indicates higher significance.)

In general, importance factors can be used to adjust scaling/unit differences of different attributes for ranking tuples in top-N queries.

Strict-N and Loose-N

Let us take a closer look at how many tuples should be returned for a top-*N* query.

Obviously, if ≤ N tuples are in the database, then all tuples should be returned. Furthermore, if the distance of the (N+1)-th ranked tuple is larger than that of the N-th ranked tuple, then exactly N tuples should be returned. When the distances of the N-th and the (N+1)-th ranked tuples are identical, two possibilities exist:

- *Strict-N*: Exactly N tuples are returned. In this case, the returned set of tuples is not unique. This may cause non-deterministic result.
- *Loose-N*: All tuples that have the same distance as the N-th ranked tuple are returned. Clearly, this may cause more than N tuples to be returned to the user. However, the non-deterministic problem is avoided.

The proposed top-N query syntax supports the specification of a user's preference on Strict-N or Loose-N.

TopNSQL Syntax

Based on the discussion above, we propose the following syntax for top-*N* queries.

SELECT <expression> FROM <table_reference> WHERE <**query_condition**>
ORDER BY <sort specification> **STOP AFTER** [exact] <value expression>

Notes:

- The SELECT and FROM clauses are the same as in standard SQL.
- <query-condition> may contain both regular operators and emphatic operators. Furthermore, each condition is of the format "A op v (r)", where A is the name of an attribute, op is an operator, v is a constant and r is the importance factor of the condition. The default importance factor is 1. Namely, when (r) is absent, it is implied that r = 1.
- <sort specification> specifies how tuples should be ordered. <sort specification> has two components with the format 'order mode, distance function'. When "order mode = 1", tuples will be ordered solely based on their distances with the reference space of the query condition. In other words, the query is a Type-1 query. When "order mode = 2", tuples satisfying the query condition will be ranked ahead of tuples not satisfying the condition, namely the query is a Type-2 query. The second component, namely "distance function", can be provided by the user according to the semantics of the application. If the user does not provide the function, the system provides a default one.
- The STOP AFTER clause was introduced in [4] and we adopt it here in our query language and extend it with the "exact" option. If the "exact" is present, exactly N tuples will be returned if ≥ N tuples are in the database (i.e., the Strict-N is chosen); else, more than N tuples may be retrieved (i.e., the Loose-N option is chosen). <value expression> is any expression that evaluates to an integer *N*, the number of tuples desired.

Example 3 (continued). Suppose a user wants to find the top 5 matches for a book titled "Advanced Database Query Processing" and no more than 5 results are needed. Based on the syntax for top-N queries introduced above, this query should be expressed as follows:

SELECT * FROM *BOOKS*
WHERE *title = 'Advanced Database Query Processing'* AND *price<<=40* AND *shipping_fee<<=5*
ORDER BY 1, sum STOP AFTER exact *5*

3 Distance Functions

In this section, we discuss what distance function is appropriate for ranking tuples for top-N queries. We argue, through the theory of Raw Relation Sets [1], that the *sum* function is reasonable when the attributes referenced in a query condition are not semantically comparable.

In general, the right choice of the distance function for a given top-N query depends on the application. For example, in Example 3, when books are ranked in ascending order of the total cost (book cost and shipping_fee), *sum* is the only appropriate function to use. As another example, consider a geographical information system in which the locations of all cities are represented as coordinates in a 2-dimensional space. For a top-N query for finding the closest cities to a given city, the right function to use is the Euclidean function. It is precisely because there does not exist a single distance function that is suitable for all applications, all proposed query languages for top-N queries let users supply the distance function when specifying a query. We also adopt this approach. On the other hand, there are many applications for which it is not clear what is the appropriate distance function to use. In this case, it may not be reasonable to expect an ordinary user to supply the distance function. This situation usually occurs when the attributes involved in the query condition are not comparable. For instance, in Example 1, the two attributes in the query condition, namely *age* and *salary*, have different meanings, and it is not clear what is the appropriate function to combine the distances due to individual attributes. Currently, there is no serious study on how to combine distances due to individual attributes when the attributes have incomparable values.

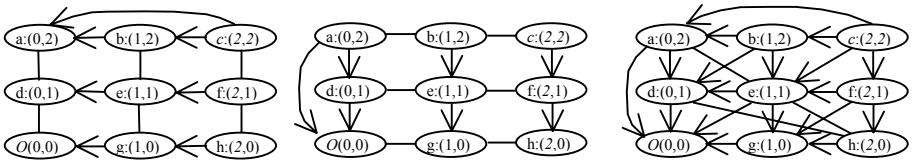
In this section, we provide a theoretic justification for using the *sum* function to combine distances due to individual attributes when the attributes have incomparable values based on the Raw Relation Sets theory [1]. Such a justification makes *sum* a good candidate to serve as the default distance function.

For simplicity, we will generally focus on two-dimensional condition space involving attributes X and Y, but our discussion and conclusion can be extended to condition space of higher dimensions. Without loss of generality, we assume that the two attributes are of the same importance for ranking tuples in the sense that one unit of difference of two values in X is of the same significance as one unit of difference in Y (otherwise, we use adjusted units after appropriate importance factors are applied).

Raw Relation Sets [1,2] is a mathematical theory for solving the problem of generating an overall order from individual orders over the same set of objects. Each individual order is described by a set of binary relationships between every pair of objects in the set. Three types of relationships may exist between two objects p and q: *larger than* (denoted by (p, q); *smaller than* is not considered since "p is smaller than q" is the same as "q is larger than p"), *equal* (denoted by <=p, q=>) and *Unknown* (denoted by <p, q>) and exactly one relationship exists between each pair of objects. Each relationship is modeled as an object in the set of relationships (the next level of objects). The fact that "object p is larger than object q" to a larger extent than "object p is larger than object r" can be represented by (p, q) and (p, r) in the first level and ((p, q), (p, r)) in the second level (i.e., (p, q) is larger than (p, r) in the second level). Informally speaking, a *raw relation set* over a set of objects is a multi-level description of the relationships between a set of objects. The general problem the raw relation set theory attempts to solve can be stated as follows:

There is a set of objects O and there is a set of voters V (e.g., a voter could be a judge in a competition or a criterion for ranking objects); each voter provides an order for the objects in O and the order is represented as a raw relation set; there is also an order for all voters (e.g., some judges/criteria are more important than others) and this order is also represented as a raw relation set. Based on the |V| + 1 raw relation sets, obtain an overall order for the objects in O.

Due to space limitation, details of raw relation sets will not be discussed in this paper. Instead, we are interested in only applying this theory to help finding a default distance function in the context of this paper. The problem at hand is a much simplified version of the general problem described above. Here, each dimension (i.e., an attribute referenced in a query condition) corresponds to a voter and for any two values v1 and v2 (objects) along this dimension, either v1 > v2, v2 > v1 or v1 = v2 is true (there is no *don't know* relationship). Moreover, all dimensions (voters) are considered to be of the same importance (i.e., only *equal* relationship exists between dimensions).



(a) Raw relation set for X (b) Raw relation set for Y (c) Raw relation set after aggregation

Figure 3.1: A raw sets of 9 nodes (not all edges are shown).

We now use a small example to illustrate how the raw relation set theory works to produce an overall order from individual orders. Consider Figure 3.1. In this example, the origin O(0, 0) is the reference point. The arrows indicate the *larger than* relationships, that is, "p → q" means that p is larger than q. Arrows along dimension X indicate that for any given Y-value, X-values farther away from the origin are larger (see Figure 3.1(a)). Similarly, arrows along dimension Y indicate that for any given X-value, Y-values farther away from the origin are larger (see Figure 3.1(b)). Now the question is how to order all the 9 nodes in the figure when both dimensions are taken into consideration. Raw relation set theory uses the following two key steps to tackle this problem:

1. Establish a direct relationship between any two nodes (objects) after all dimensions are considered. Simple rules are used for this purpose. Consider two nodes p and q. Some of the rules are as follows. (1) If along both dimensions, p > q is true, then p > q overall. (2) If along one dimension, p > q, and along another dimension, p = q, then p > q overall. (3) If along one dimension, p > q, and along another dimension, q > p, then p = q overall. Based on the basic relationships as shown Figure 3.1(a)-(b), 36 direct relationships can be established between the nine nodes (see Figure 3.1(c); for simplicity, not all of the direct relationships are shown there) using these rules and some examples are e > O, h > g, e = h, d = h.
2. Use all the direct relationships established in Step 1 to produce a final order (not necessarily unique) among all nodes. Intuitively, the reason that the direct relationship between two nodes may be different from the final relationship of the two nodes is because other direct relationships may affect the outcome. This can probably be best explained by an analogy in some sports competition: the fact that

team A beats team B does not necessarily mean that A will be ahead of team B in the final standing because the games between these two teams and other teams will be considered in determining the order of teams in the final standing. It can be shown (see below) that there is an *anti-symmetric relationship* among the nodes with respect to the diagonal line from the upper-left corner to the lower-right corner of the two-dimensional space. This relationship indicates that among the widely used distance functions, *sum* is the most appropriate.

Consider the nodes in a general m×m space (Figure 3.1 shows a 3×3 space). The following properties can be observed about these m^2 nodes (let V denote the set of nodes) and the direct relationships among them.

1. There is a diagonal line l from the upper-left corner to the lower-right corner of the two-dimensional space, which consists of nodes: (0, m), (1, m-1), … , (m, 0).
2. The direct relationships among the nodes on l are all *Equal*.
3. The following one-to-one mapping f: $V \rightarrow V$ exists (see Figure 3.2). For any node a, there exists a symmetric node a' regarding line l, such that for any node v on l:
   if a is *Larger* than v then a' is *Smaller* than v;
   if a is *Smaller* than v then a' is *Larger* than v;
   if a is *Equal* to v then a' is *Equal* v.

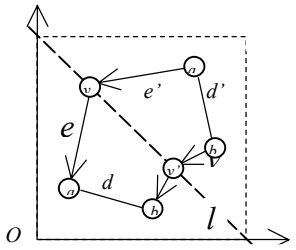


Figure 3.2: Anti-Symmetry Relationship of Nodes

Furthermore, for any edge "a, b", the relationship between a and b is identical to the relationship between b' and a'. For example, for Figure 3.1(c), the mapping: f(O)=c; f(d)=b; f(g)=f; f(a)=a, f(e)=e; f(h)=h, satisfies all of the above properties.

A graph with the above properties is referred to as an *anti-symmetric* graph.

Anti-symmetric rule: If graph G is anti-symmetric, then the nodes on the line l should all have the Equal relationship and be all ranked in the middle of the final ordered list of all nodes.

It can be seen that among the three most widely used distance functions, namely, *min*, *Euclidean* and *sum*, only the *sum* function satisfies the anti-symmetric rule.

4 Top-*N* Query Processing

In this section, we present a simple method for evaluating top-N queries. We assume that the execution engine is a traditional relational database engine that supports single as well as possibly multi-attribute indexes. We consider only the *sum* distance function.

A naïve method to evaluate a top-N query is to compute the distance of every tuple with the query, rank all tuples as specified by the ORDER BY clause of the query and return the top N tuples to the user. This solution is inefficient in several aspects. First, it requires all tuples to be retrieved from the database, incurring high database retrieval cost. Second, it requires the distances of all tuples with respect to the query to be computed, incurring high computation cost. Finally, it requires the sort of all tuples, incurring high sorting cost. The problem of top-N query optimization is to find efficient ways to evaluate top-N queries. The idea is to convert a top-N query to a regular database query that minimizes the retrieval of useless tuples from the database.

In this section, we provide a new method to minimize the retrieval of useless tuples from the database for a given top-N query.

We make several general assumptions to simplify our discussion. None of these assumptions fundamentally change the nature of the problem under consideration. The first assumption is that all query conditions are based on attributes of numeric type. Currently, only this type of conditions is used to rank tuples. Other types of conditions, e.g., conditions involving character strings, if exist, are assumed to be evaluated first. The second assumption is that each attribute takes values between 0 and 1. This can be achieved by normalizing the values of each attribute. The third assumption is that only two attributes are involved in the query condition. Let X and Y be the attributes involved. The problem of optimizing the query is to find four values x1, x2, y2 and y2 such that all top N tuples of the query are retrieved by condition "x1 ≤ X ≤ x2 and y1 ≤ Y ≤ y2" and the rectangle defined by [x1, x2] and [y1, y2] is as small as possible. The space [x1, x2] and [y1, y2] will be called the *search space*. The following notations will be used in this section: Q is the user submitted top-N query, Q* is the converted database query to be submitted to the underlying database system, and *n* is the number of tuples in the *universe space* $0 \le X \le 1$, $0 \le Y \le 1$.

We now describe our method for evaluating top-N queries based on the assumption that tuples are uniformly distributed.

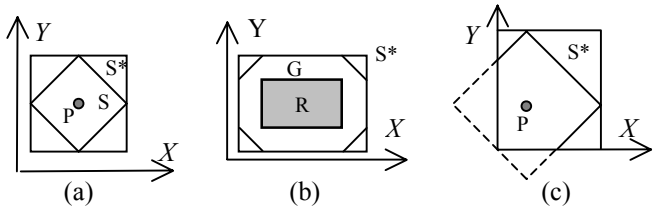


Figure 4.1: Top-*N* search spaces.

We first assume that the top-N query Q has a point condition. In this case, there is no difference between Type-1 query and Type-2 query.

Algorithm Basic:

1. Obtain the reference space from the condition space of the query Q (see Section 2). It is assumed that the reference space is a point for now. Let P(x, y) denote the point, $0 \le x \le 1$, $0 \le y \le 1$.
2. Obtain a tentative search space S for Q*. Intuitively, S should satisfy two conditions. First, the size of S should be large enough to contain N tuples. Based on the uniform distribution of all tuples, the size of S can be estimated to be (N/n)*U, where U is the size of the universe space (U = 1). For now we assume that S is entirely contained in the universe space (this assumption will be removed later). Second, tuples inside S should be closer to P than tuples outside S. Based on the *sum* function, S is a square whose sides have 45° angle with the axes (see Figure 4.1(a)). It can be seen that all points on the edges of S have the same distance from P based on the *sum* distance function. Squares such as S will be called *rotated squares* in this paper.
3. Obtain the final search space. Current database systems are not capable of processing queries whose search space is of shape S. Therefore, another square S* that is the smallest to contain S with sides parallel to the axes is obtained as the

final search space for Q* (see Figure 4.1(a)). It is easy to see that the size of S* is 2N/n.

4. From S*, the condition of query Q* is obtained as "$x - \sqrt{N/2n} \le X \le x + \sqrt{N/2n}$ and $y - \sqrt{N/2n} \le Y \le y + \sqrt{N/2n}$".

It is easy to see that the size of S* is exactly twice of that of the desired search space S. We now discuss how to remove ssome assumptions used by Algorithm Basic.

1. Algorithm Basic assumed that the reference space is a point. We now consider the case when the reference space is a rectangle R defined by $x1 \le X \le x2$ and $y1 \le Y \le y2$ (note that a line segment is a special case with either $x1 = x2$ or $y1 = y2$). All tuples in R have zero distance with the query. It is possible that R may contain N or more tuples (i.e., R is of the same size as or larger than S). In this case, if Loose-N is specified in the query, R will be searched directly and all retrieved tuples will be returned to the user; if Strict-N is specified, then search any rectangle that is contained in R and is of size S. In the following, we assume that R is smaller than S.

   As we pointed out in Section 2, when the reference space is a rectangle, different results may be obtained depending on whether a query is of Type-1 or Type-2. For Type-1 query, the rotated square S* in Figure 4.1(a) needs to be extended to an octagon G in order to guarantee that all points on the edges of the octagon are the same distance away from the rectangle R (see Figure 4.1(b)). The size of G is the same as S. From G, the smallest rectangle containing G can be obtained and will be used as the search space for query Q*.

   For Type-2 query, one method is to form two queries for the database system. The first query Q1 is obtained directly from Q after emphatic operators are replaced by corresponding regular operators and the ORDER BY and the STOP AFTER clauses are removed. This query retrieves all tuples that satisfy the query condition (i.e., tuples in the condition space C). The second query Q2 searches the space S* - C. That is, Q2 retrieves all tuples that are closest to the query condition but are not retrieved by Q1. The two sets of tuples are merged into one list by first listing the tuples retrieved by Q1 in ascending order of distance and then tuples retrieved by Q2 in ascending order of distance. The main drawback of this method is that two queries need to be processed by the database system. Even though no redundant tuples are retrieved by the two queries, two invocations to the database system from an application program can be costly. An alternative method for Type-2 query is as follows. Note that the reference space and the condition space may be different due to the possible existence of emphatic operators. As a result, it is possible that C is not contained in S*. The alternative method is to form one query whose search space (a rectangle) is the smallest to contain both C and S*. This may lead to a larger search space than the two-query approach but in requires just one invocation to the database system. Our preliminary experiments (not reported here due to space limitation) indicate that the one-query alternative is usually more cost effective than the two-query method.

2. Algorithm Basic assumed that the entire S* is contained in the universe space. When the reference point P is on or near the boundary of the universe space, the final search space may no longer be a square but a general rectangle. In this case,

step 2 of the algorithm needs to be modified to obtain a rotated square such that the intersection area of the square and the universe space is the same as S and the distance between P and every point on the edges of the square is the same (see Figure 4.1(c) for an example). Generalization to cases where the reference space is rectangle is straightforward (in this case the rotated square will be an octagon).

Even though query Q* is expected to retrieve 2N tuples (S* is twice the size of S), there is no guarantee that Q* will retrieve at least *N* tuples from the database when the data are seriously skewed. If less than *N* tuples are retrieved, the search space needs to be expanded to retrieve additional tuples. Let Q** denote the new query based on the expanded space only (the search space of Q** does not include the search space of Q*) and *r* is the number of additional tuples that need to be retrieved by Q**. The size of the search space of Q** can be computed by R**r/t* [2], where *R* is the size of the unsearched space (universe space minus the search space of Q*) and *t* is the total number of tuples in *R*. The ratio *r/t* will be called the *expansion rate*. The expansion may be repeated until enough tuples are retrieved. It is shown in [2] that when the tuples are uniformly distributed, the expected number of expansions needed is 1.

5 Conclusions

In this paper, we explored the three important issues related to top-*N* queries in a relational database context:

1. Query language: We proposed an extension to the SQL query language to facilitate the specification of various top-N queries. New operators such as << and >> are introduced and difference ranking options (Type-1 versus Type-2) are incorporated to allow users to express their top-N queries more clearly.
2. Distance function: We made a case for the *sum* function arguing that it is a more appropriate for ranking tuples when attributes involved in a top-N query are incomparable.
3. Query processing: We proposed a simple method to process top-N queries based on the *sum* distance function and the uniform distribution assumption of data. This method is easy to implement and yields small search spaces.

References

1. Y. Chen. Raw Sets. The Journal of Fuzzy Mathematics, Vol.8, No.3,2000, Los Angeles. pp. 607-617.
2. Y. Chen. Raw Relation Sets, Order Fusion And Top-*N* Query Problem. Ph.D. Dissertation, Department of Computer Science, Binghamton University, 2002.
3. S. Chaudhuri and L. Gravano. Evaluating Top-*k* Selection Queries. 25th VLDB Conference, Edinburgh, Scotland, 1999. pp.399-410.
4. M. J. Carey and D. Kossmann. On saying "Enough Already!" in SQL. ACM International Conf on Management of Data (SIGMOD'97), May 1997. p219-230.
5. D. Donjerkovic and R. Ramakrishnan. Probabilistic Optimization of Top N Queries. In Proc of the 25th VLDB Conf., Edinburgh, Scotland, 1999. p411-422.
6. R. Fagin. Combining Fuzzy Information from Multiple Systems. PODS'96, Montreal, Canada. pp.216-226.
7. C.Yu, P.Sharma, W. Meng, and Y, Qin. Database Selection for Processing k Nearest Neighbors Queries in Distributed Environments. First ACM/IEEE Joint Conference on Digital Libraries, Roanoke, VA, June 2001.