Expressing and Optimizing Similarity-Based Queries in SQL

(Extended Abstract)*

Like Gao[1], Min Wang[2], X. Sean Wang[1], and Sriram Padmanabhan[2]

[1] CS Dept., University of Vermont, VT
{lgao,xywang}@emba.uvm.edu
[2] IBM T.J. Watson Research Center, NY
{min,srp}@us.ibm.com

Abstract. Searching for similar objects (in terms of near and nearest neighbors) of a given query object from a large set is an essential task in many applications. Recent years have seen great progress towards efficient algorithms for this task. This paper takes a query language perspective, equipping SQL with the near and nearest search capability by adding a user-defined-predicate, called NN-UDP. The predicate indicates, among a set of objects, if an object is a near or nearest-neighbor of a given query object. The use of the NN-UDP makes the queries involving similarity searches intuitive to express. Unfortunately, traditional cost-based optimization methods that deal with traditional UDPs do not work well for such SQL queries. Better execution plans are possible with the introduction of a new operator, called NN-OP, which finds the near or nearest neighbors from a set of objects for a given query object. An optimization algorithm proposed in this paper can produce these plans that take advantage of the efficient search algorithms developed in recent years. To assess the proposed optimization algorithm, this paper focuses on applications that deal with streaming time series. Experimental results show that the optimization strategy is effective.

1 Introduction

In many applications, searching for similar objects of a given query object from a large given set is important. Similarity measure is best intuited as some distance and similarity is then usually expressed in terms of near and nearest neighbors. When complex objects such as time series are involved and object sets are large, the task of finding near and nearest neighbors becomes rather costly. A large body of research has been devoted to reducing this cost and efficient algorithms and indexing structures have been developed (see, e.g., [13, 12, 1, 5, 11, 10]).

To the best of our knowledge, however, there has not been any systematic study on how to incorporate near and nearest neighbor searches into the popular query language SQL and, more important, how to optimize the resulting queries. The purpose of this paper is to initiate such a study, considering the two aspects of the query language, namely expression and optimization, when it needs to deal with similarity searches.

* This is an abbreviated version of the technical report [6].

P. Atzeni et al. (Eds.): ER 2004, LNCS 3288, pp. 464–478, 2004.
© Springer-Verlag Berlin Heidelberg 2004

As an example application, consider the problem of monitoring different sources of time series data to detect certain events (e.g., onset of a flu season). Assume we have collected many patterns (in the form of time series) from historical data on school attendance and flu-related medicine sales at pharmacies. Using data analysis tools, we may have learned the strong correlation between the presence of certain events and the appearance of certain patterns in school and pharmacy data. For example, a sharp decrease in school attendance accompanied by a sharp increase in pharmacy sales three days in a row is a strong indication of the beginning of a flu season. Based on such learned "rules" and the time series data reported everyday regarding current school attendance and pharmacy sales, we may detect the appearance of certain events.

To be more specific, suppose we have two sets of historical patterns, S for school attendance and P for pharmacy sales. For simplicity, let us assume $S = \{\mathsf{s1}, \mathsf{s2}, \mathsf{s3}\}$ and $P = \{\mathsf{p1}, \mathsf{p2}, \mathsf{p3}\}$, respectively. We can store the rules learned from historical data in a relation called *Events* shown in Figure 1. Each row in the relation represents a rule learned from the historical data. For example, the first row says that when school attendance data matches pattern s1 and pharmacy data matches pattern p1, it usually indicates the peak of a flu season.

EID	*eName*	*School*	*Pharmacy*
e1	Flu_peak	s1	p1
e2	Flu_start	s2	p2
e3	Flu_end	s3	p3

Fig. 1. *Events* table with the rules learned from historical data

The meaning of "match" in the above example can be in terms of near and nearest neighbor based on a similarity (or distance) measure. Near neighbors are defined in terms of the distance between a pair of objects (e.g., time series) irrespective of the existence of other objects, while nearest neighbor is a relative notion, defined with respect to a set. Both notions are best used together. Thus, that the current school data matches pattern s1 may indicate that among all the school patterns, s1 is closest to the current school data (nearest neighbor notion), and *at the same time*, they are not too far away from each other (near neighbor notion).

Clearly, algorithms and data structures that can provide efficient evaluation of near and nearest neighbors can be very helpful in the above example application. However, it is also very important that the users should have an intuitive language to express these types of queries. At the same time, the system should figure out how to efficiently answer these queries, invoking efficient algorithms and indexing structures. For this purpose, we propose to add a user-defined-predicate (UDP) to SQL for users to express the notion of near and nearest neighbors in their queries. The UDP, called NN-UDP, indicates, among a set of objects, if an object is a near/nearest-neighbor of a query one.

The use of the NN-UDP makes the queries involving near/nearest neighbors easy to express, since the user can intuitively treat it as a selection condition. However, when

the query is evaluated, we probably do not always want the predicate as a selection condition. Indeed, we would like to take advantage of the algorithms and data structures for finding near and nearest neighbors. For example, in our detection application, we may want to report the corresponding event name by performing the following three steps:

1. In pattern set S, find the nearest neighbor (call it si) of the current school series.
2. In pattern set P, find the nearest neighbor (call it pj) of the current pharmacy series.
3. Issue an SQL to select the $eName$ of table $Events$ with $School$ = si and $Pharmacy$ = pj as the conditions.

Here, we use the direct method to find the near/nearest neighbors from sets of objects instead of using the NN-UDP as selection conditions. Obviously, various strategies can apply to each of the above steps.

However, the above strategy may not always be the best. For example, once the school pattern si is found from Step 1, the number of tuples in the *Events* relation that satisfy the condition $School =$ si may be so small that, in fact, a near/nearest neighborhood test of (in contrast to search for) the corresponding patterns from the Pharmacy column will be more beneficial.

In order to make the correct decision on selecting the appropriate evaluation strategy, we introduce a heuristic optimization method which is based on a new operator NN-OP and the derived algebraic equivalence rules involving NN-UDP and NN-OP. Our experiments have confirmed the superiority of this systematic cost-based approach.

By treating each NN-UDP in the query either as a traditional UDP or as the output of NN-OP, our proposed optimization method can find better execution plans than those appearing in previous work on UDP query optimization [4, 9, 3, 2], In [4], Chimenti et al. propose an algorithm for LDL system to optimize the queries with UDPs. In LDL, each UDP is treated as a relation during the query optimization process. However, since it uniformly treats each UDP as a relation, it may fail to consider some efficient plans. Hellerstein and Stonebraker propose Predicate Migration algorithm [9, 8] that improves the LDL approach by pushing down selections on both operands of a join. Later, Chaudhuri and Shim present several efficient algorithms that are able to guarantee the optimal plan over the desired execution space and show these proposed algorithms can either find the optimal plan or efficiently find a plan that is very close to the optimal one [3]. However, all these algorithms uniformly treat UDPs as selection conditions.

Our work is also related to [2] by Chaudhuri and Gravano, which deals with query optimization when external searches are involved. They study the optimization of queries over multimedia repositories, and assume that query predicates are independent. However, in this paper, the similarity searches on different sources are less likely to be independent since they are involved in the detection of the same event.

The contribution of this paper can thus be summarized as follows. Firstly, we take a query language perspective to deal with similarity searches. The novelty of our language is on the incorporation of the nearest neighbor searches. This makes similarity-based queries easy to write and provides a powerful tool for various related tasks. Secondly, we provide a heuristic optimization algorithm to derive efficient evaluation plans for these queries, fully taking advantage of the efficient algorithms and index structures for near/nearest neighbor for evaluation. Thirdly, we use experiments to demonstrate the effectiveness of our optimization algorithm for the (streaming) time series case.

The remainder of the paper is organized as follows. In Section 2, we define our extension of SQL, called SQL/sim, to incorporate the similarity search capability into SQL. In Section 3, we discuss our optimization algorithm, including algebraic equivalence rules, and our heuristic method for deriving optimized evaluation plans. We report experimental results in the (streaming) time series case in Section 4, and in Section 5, we conclude our paper with some future research directions.

2 SQL/sim

In this section, we provide a simple extension of SQL, called SQL/sim, that offers the capability of expressing similarity searches in RDBMSs. We start with defining the similarity between pairs of objects.

Definition Given two objects p and q, the similarity measure, denoted $sim(\mathsf{p}, \mathsf{q})$, is a non-negative real number.

The similarity metrics might have a positive or inverse relationship with respect to the similarity of two objects. For example, Correlation Coefficient is positive, i.e., the large the metric value, the more similar the objects are. On the other hand, Euclidean Distance is inversely related to the similarity of objects, i.e., the smaller the metric value, the more similar the objects are. Without loss of generality, in this paper, we assume that if two objects are more similar, then the similarity metric is smaller. In this case, similarity measure is more like a distance measure.

Definition Let α be a non-negative real number. Given a *query* object q, an object p is said to be its α-*near neighbor* if $sim(\mathsf{p}, \mathsf{q}) \leq \alpha$.

In the above definition, the number α is called the nearness *threshold*. This near-neighbor definition relates similar objects independently of the existence of other objects. In some situations, it is meaningful to obtain similarity between the query object and an object relative to a set of objects. We call this relative measure as the k-nearest neighbors defined as follows:

Definition Let $k \geq 1$ be an integer, $\mathsf{P} = \{\mathsf{p}_1, \mathsf{p}_2, \ldots, \mathsf{p}_m\}$ a set of objects, and q a query object. An object p_i in P is said to be *one of the* k-*nearest neighbors of* q in P if there are at most $k-1$ objects p_j $(j \neq i, 1 \leq j \leq m)$ such that $sim(\mathsf{p}_j, \mathsf{q}) < sim(\mathsf{p}_i, \mathsf{q})$.

Integer k above is called the *rank of similarity*[1]. For $k = 1$, the 1-nearest neighbor of q is normally abbreviated as the *nearest neighbor of* q.

2.1 Required Relations

In order to model the notions of objects and object sets for the purpose of near/nearest-neighbor searches in a RDBMS, an application needs to set up at least two relations. One corresponds to the set of all (pattern) objects (abstractly called *Patterns*), and the other to the collection of (pattern) object sets (called *PatternSets*). Each pattern set must be a subset of the *Patterns*.

[1] For simplicity, we assume no two pairs of objects will have exactly the same similarity measure. In real applications, we remark it's easy to lift this restriction.

For our running example, the *Patterns* are all the historical patterns and the *PatternSets* are those collections of historical patterns related to specific types of events (e.g., all the patterns related to school attendance are collected into $schoolSet$).

In terms of relation schemas, the relation for *Patterns* should have an ID attribute (PID) that is the primary key of the relation; and the relation for *PatternSets* should have two attributes that are for the ID of the sets (SID) and the ID of the Pattern (PID). The primary key must be these two attributes together, and the PID must reference to the ID of the *Patterns* relation. These two relations only need to use the identifiers of the objects and the object sets, while the objects themselves may be stored elsewhere inside or outside of the relational database. For example, the objects themselves may be stored as BLOBS in a RDBMS.

The relations corresponding to our running example are shown in Figure 2. Here, the two required relations are given by *Patterns* and *PatternSets*, respectively. We assume that all patterns in the *Event* relation (under the attributes *School* and *Pharmacy*) in Figure 1 all refer to attribute *PID* in the *Patterns* relation.

Patterns (P)

PID	pName
s1	$Fast\_Up$
s2	Up_Down
s3	$Slow\_Up$
p1	$Slow\_Down$
p2	$Quick\_Change$
p3	$Fast\_Down$

PatternSets (PS)

SID	PID
$schoolSet$	s1
$schoolSet$	s2
$schoolSet$	s3
$pharmacySet$	p1
$pharmacySet$	p2
$pharmacySet$	p3

Fig. 2. For our running example, the required *Patterns* and *PatternSets* relations

Query objects need not belong to relation P. They can be stored in relations or can be constant IDs that are understood by SQL/sim. In this paper, we will use constant IDs.

2.2 NN-UDP

To equip SQL with the near/nearest-neighbor search capability, we introduce a user-defined-predicate (UDP) as follows.

Definition NN is a 5-ary predicate such that for each given query object QID, pattern set SID (from relation PS), pattern PID (from relation P and in the set SID), integer $\mathsf{RK} \geq 1$, and real number $\mathsf{TH} \geq 0$,

$$NN(\mathsf{QID}, \mathsf{SID}, \mathsf{PID}, \mathsf{RK}, \mathsf{TH}) = \textbf{True}$$

if and only if PID is one of the RK-nearest neighbors of QID in the pattern set SID, and the similarity measure between them is no greater than TH. NN is called the NN-UDP.

For example, NN('s', '$schoolSet$', 's1', 1, 0.2)=**True** if and only if s1 is the nearest neighbor of s in $schoolSet$, and the similarity measure of s1 and s is no greater than 0.2. Unlike most other proposals, NN-UDP incorporates both near and nearest neighbor tests.

In the above definition, the input relation takes five parameters. Among them, QID and PID are specific values, while SID, RK and TH can take NULL values. The semantics of these three attributes with NULL values are:

1. If SID=NULL, the pattern set is all patterns in P.
2. If RK=NULL, the similarity rank is infinity.
3. If TH=NULL, the similarity threshold is infinity.

Therefore, SID=NULL is a special case to take all the patterns as a "global" set for the purpose of nearest neighbor test. Furthermore, by allowing NULL either for RK or TH, we can use NN-UDP for either near or nearest neighbor tests, respectively. In the cases where both RK and TH are NULL, $NN(\mathsf{QID}, \mathsf{SID}, \mathsf{PID}, \mathsf{RK}, \mathsf{TH}) =$**True** means that PID is in the set SID. On the other hand, if neither RK nor TH is NULL, then $NN(\mathsf{QID}, \mathsf{SID}, \mathsf{PID}, \mathsf{RK}, \mathsf{TH}) =$**True** means that PID is both a near and nearest neighbor of QID.

2.3 SQL/sim Examples

With SQL/sim, users can write similarity-based queries in an intuitive manner. Here we give two example queries. The first corresponds to the flu detection scenario mentioned in the introduction.

Example 1. Report the event name based on the observed school time series s and pharmacy time series p. More specifically, the event name is decided by: (1) the nearest neighbor of school attendance time series in $schoolSet$ with similarity measure no greater than 0.2; (2) the nearest neighbor of pharmacy time series in $pharmacySet$; and (3) the rules in the $Events$ table, as described in Figure 2. The query in SQL/sim is:

```
SELECT E.eName
FROM   Events E
WHERE  NN('s', 'schoolSet', E.School, 1, 0.2)
  AND  NN('p', 'pharmacySet', E.Pharmacy, 1, NULL)
```

The result is a list of event names.

Example 2. As another example, we want to find the PIDs of the nearest neighbor of the school attendance time series s in $schoolSet$ if this nearest neighbor satisfies the following two conditions: (1) the similarity measure is no greater than 0.2; and (2) according to the rules in table $Events$, this nearest neighbor pattern is correlated to the event named Flu_peak, i.e., this nearest neighbor appears in a row of $Events$ table with eName='Flu_peak'. The query is as follows, and the result is a list of school IDs.

```
SELECT E.School
FROM   Events E
WHERE  E.eName= 'Flu_peak'
  AND  NN('s', 'schoolSet', E.School, 1, 0.2)
```

3 Optimizing SQL/sim

In this section, we develop a strategy to optimize queries in SQL/sim. We start with an example to show various options for evaluating queries in SQL/sim. We then describe

a heuristic method that can, in many cases, automatically take the best option when evaluating a query.

Consider the query in Example 1. Using traditional optimization methods for UDPs, two execution plans are possible:

(1) One may consider the order of applying the two NN-UDPs in the query. Cost and selectivity should both be considered for this ordering as in [3, 9].
(2) Another method is to consider each NN-UDP as a relation as in [4]. In this case, both occurrences of NN will be evaluated on the entire relation and the results are joined together (with *Events* relation again). The join order needs to be carefully considered as in [4].

Each of the above strategies has its advantages in particular situations as explained below. This is due to the fact that, in most situations, algorithms that test if an object is a near/nearest neighbor are faster than those that search for near/nearest neighbors[2]. More specifically, in our example, if the number of patterns in *Events* under the *School* attribute is large, then it is beneficial to find the nearest neighbor (with RK=1 and TH=0.2) of the query object s by using some indexing method. Since at most one school attendance pattern will be found by this process, once this is done, we can select the tuples in the *Events* relation that contain that particular school attendance pattern, and project out the patterns under the *Pharmacy* attribute in these tuples. If the number of the resulting patterns is small, we can then use the NN-UDP to test each one. If the number is still large, we can use an index-based algorithm to find the nearest neighbor of p, and join back to the relation *Events* to obtain the final result. (This last case, where index-based algorithms are used twice, corresponds to the strategy found in [4]). Of course, this whole process can be done starting with the second occurrence of the NN-UDP.

Another possibility is that there are only a few tuples in *Events*. Then testing each one of them using the two NN predicates will probably be the best strategy. Here, the strategy of [3, 9] should be considered.

The above example shows that each of the traditional methods mentioned previously may be best in certain situations. However, a combination of these methods may be called for in certain other situations. The choice must be made by considering the cost, selectivity and sizes of the involved operations and intermediate results.

3.1 Near/Nearest-Neighbor Operator

From the above example, we can see that we cannot simply treat NN-UDPs as selection conditions or their output as relations. Rather, we need to choose different plans for different query instances. Sometimes we need to use index-based algorithms to directly find near/nearest-neighbors, and sometimes we may use the NN-UDP directly.

[2] For example, we used a scan method for both testing and searching in our experiments. In our scan method, search needs to look through the entire pattern set, while test can stop much earlier when a nearer object is found and hence is faster in general. If multidimensional index is used for multidimensional objects, a test is only to ask whether there is any object that is within a range and hence is generally faster than searching for the exact objects that are the near/nearest neighbors.

$R =$

QID	SID	RK	TH
s	*schoolSet*	1	0.4
p	*pharmacySet*	1	0.5

$D(R) =$

QID	SID	PID	RK	TH
s	*schoolSet*	s1	1	0.4
p	*pharmacySet*	p3	1	0.5

Fig. 3. Example of NN-OP, assuming s1 is the near/nearest neighbor of s in *schoolSet* (with RK $= 1$ and TH $= 0.4$), and p3 is the near/nearest neighbor of p in *pharmacySet* (with RK $= 1$ and TH $= 0.5$)

In order to derive such optimized evaluation plans, we need to define an operator that encodes the use of an indexing algorithm for finding near/nearest-neighbors.

Definition Let $S = \{\mathsf{QID}, \mathsf{SID}, \mathsf{RK}, \mathsf{TH}\}$. The relational operator D is defined as follows: For each relation R whose schema contains all the attributes in S, $D(R)$ is the relation with the schema $S \cup \{\mathsf{PID}\}$ such that a tuple t is in $D(R)$ if and only if (1) $t[S]$ is in $\pi_S(R)$, and (2) $NN(t[\mathsf{QID}], t[\mathsf{SID}], t[\mathsf{PID}], t[\mathsf{RK}], t[\mathsf{TH}])$ = **True**. Operator D is called the NN-OP.

Intuitively, for each tuple t in R, D finds the near/nearest neighbors for the query object $t[\mathsf{QID}]$ among the patterns in the set $t[\mathsf{SID}]$ with rank and threshold RK and TH, respectively. Figure 3 shows an example of the D operator.

The above definition of D is extended to relations that have some of the attributes in S missing. More specifically, R may not contain any of the attributes SID, RK, TH. In these cases, the output of D will not have these attributes either, and the condition (2) in the above definition will take NULL value in place of $t[\mathsf{SID}], t[\mathsf{RK}], t[\mathsf{TH}]$.

It should be noted that the output size of NN-OP may be even bigger than that of the input relation. For example, if $\mathsf{RK} = k$, then for each tuple in R, $D(R)$ may contain k tuples derived from it.

3.2 Equivalence Rules

As explained in the beginning of this section, our goal is to use the NN-OP in place of some NN-UDPs in evaluating a query. In this section, we give a set of transformation rules for this purpose.

Our transformation rules work on the relational algebra expression derived from SQL/sim. In a relational algebra expression, in a natural way, we treat NN-UDP as a selection condition on a relation R containing attributes QID, SID, PID, RK, and TH, written as $\sigma_{NN}(R)$. If R does not have any of the attribute SID, RK, TH, the corresponding value is treated as NULL.

As an example, let R_1 be a relation with schema {QID, SID, RK, TH} that contains only one tuple ('s', '*schoolSet*', 1, 0.2) and R_2 be a relation with schema {QID, SID, RK} that contains only one tuple ('p', '*pharmacySet*', 1). The SQL/sim query in Example 1 can be written in relational algebra form as $\pi_{eName}(R_1'' \bowtie R_2'')$, where

$$R_1'' = \pi_{EID,eName}(\sigma_{NN}(R_1')) \text{ with } R_1' = \rho_{School \to PID}(\, R_1 \times \mathit{Events}), \text{ and}$$
$$R_2'' = \pi_{EID,eName}(\sigma_{NN}(R_2')) \text{ with } R_2' = \rho_{Pharmacy \to PID}(R_2 \times \mathit{Events}).$$

Note that ρ is the renaming operator.

In the following, R can be any relation with the implied attributes. For notational convenience, in these rules and the later plans, we will indicate the ordering of the operators by nested algebraic expressions.

Figure 4 shows the equivalence rules. Rule 1 gives the equivalence transformation between σ_{NN} and D. Rule 2 shows how to move a selection operator σ inside the NN-OP D. Likewise, we can exchange the order of join and NN-UDP as in Rule 3. Rules 4-7 are useful to prune some operators and the associated relations from the plan. Note that Rule 4 and 5 are different since in Rule 4, relation PS refers to the *PatternSets* relation which contains attribute PID, while in Rule 5, the schema of R does not contain PID.

Rule 1	$\sigma_{NN}(\mathrm{R}) \equiv R \bowtie D(R)$
Rule 2	$\sigma_c(D(R)) \equiv D(\sigma_c(R))$, if c only refers to attribute(s) in $\{\mathsf{QID}, \mathsf{SID}, \mathsf{RK}, \mathsf{TH}\}$.
Rule 3	$R_1 \bowtie D(R_2) \equiv R_1 \bowtie D(R_1 \bowtie R_2)$
Rule 4	$PS \bowtie D(R \bowtie PS) \equiv D(R \bowtie PS)$
Rule 5	$R \bowtie D(R \bowtie PS) \equiv D(R \bowtie PS)$ if R's schema is a subset of $\{\mathsf{QID}, \mathsf{SID}, \mathsf{RK}, \mathsf{TH}\}$.
Rule 6	$\pi_p(\mathrm{D}(R)) \equiv \pi_p(R)$ if p is a set of attribute that does not contain PID.
Rule 7	$D(\pi_p(R)) \equiv D(R)$ if p is the subset of $\{\mathsf{QID}, \mathsf{SID}, \mathsf{RK}, \mathsf{TH}\}$ appearing in R's schema.

Fig. 4. Equivalence rules (PS is *PatternSets*)

With the set of equivalence rules shown in Figure 4, we can transform a query plan involving NN-UDPs into equivalent ones with NN-OPs only or a combination of NN-UDPs and NN-OPs. We use the query in Example 1 to illustrate how to use these rules to obtain different query plans.

At the beginning of this subsection, we represent the query as $\pi_{eName}(R_1'' \bowtie R_2'')$. This expression corresponds to the straightforward way of executing the query by testing the two NN-UDPs independently, joining the two testing results on attribute EID, and projecting out the interested attribute.

Alternatively, we can use the equivalence rules to generate another query plan: $\pi_{eName}(R_1'' \bowtie R_2'')$

$\equiv \pi_{eName}(\pi_{EID,eName}(\sigma_{NN}(R_1')) \bowtie \pi_{EID,eName}(\sigma_{NN}(R_2')))$

$\equiv \pi_{eName}(\pi_{EID,eName}(\sigma_{NN}(R_1')) \bowtie \sigma_{NN}(R_2'))$ *//standard equivalence*

$\equiv \pi_{eName}(\pi_{EID,eName}(R_1' \bowtie D(R_1')) \bowtie \sigma_{NN}(R_2'))$ *// Rule 1*

$\equiv \pi_{eName}(\pi_{EID,eName}(R_1' \bowtie D(R_1)) \bowtie \sigma_{NN}(R_2'))$

//since $D(\pi_{\mathsf{QID},\mathsf{SID},\mathsf{RK},\mathsf{TH}}(R_1')) = D(R_1')$ *by Rule 7, and* $\pi_{\mathsf{QID},\mathsf{SID},\mathsf{RK},\mathsf{TH}}(R_1') = R_1$

$\equiv \pi_{eName}(\sigma_{NN}(\pi_{EID,eName}(R_1' \bowtie D(R_1)) \bowtie R_2'))$ *//standard equivalence*

The resulting expression corresponds to the following query plan: We first use an index-based algorithm to discover the nearest neighbor of s in the pattern set $schoolSet$ (this corresponds to $D(R_1)$). We then find the events that use that particular school pattern (i.e., the first join $R_1' \bowtie D(R_1)$). We then join the result with R_2', the input relation of the second NN-UDP, on attribute EID and $eName$. Finally, we test the second NN-UDP on the join output and project out the interested attribute $eName$.

Naturally, we can get another plan in a symmetrical way:
$\pi_{eName}(\sigma_{NN}(\pi_{EID,eName}(R_2' \bowtie D(R_2)) \bowtie R_1'))$.

As another possibility, we may continue the above transformation as follows (picking up from the second to the last step):

$\pi_{eName}(R_1'' \bowtie R_2'')$

$\equiv \pi_{eName}(\pi_{EID,eName}(R_1' \bowtie D(R_1)) \bowtie \sigma_{NN}(R_2'))$

$\equiv \pi_{eName}(\pi_{EID,eName}(R_1' \bowtie D(R_1)) \bowtie \pi_{EID,eName}(R_2' \bowtie D(R_2)))$

//same as done to $\sigma_{NN}(R_1')$

The resulting expression corresponds to the following query plan: We first use an index-based algorithm to discover the nearest neighbor of s in the pattern set $schoolSet$, as well as the nearest neighbor of p in the pattern set $pharmacySet$. We then join the results with R_1' and R_2', respectively, to find the corresponding events, and finally obtain the common events by a join.

In the above plans, we have only used Rules 1 and 7. For other queries, e.g., pattern sets IDs are from a relation instead of being a constant, other rules will be useful.

3.3 Optimization Procedure

From the example given in the beginning of the section, we can see that a good execution plan is more likely to combine the use of both NN-UDPs and NN-OPs. Even though the equivalence rules of the previous subsection, together with the standard equivalence rules from the relational algebra, can be used to search through all the execution plans, it is obviously a very large space for an exhaustive search.

Instead of using exhaustive search, we give an optimization algorithm, called the *UdpOp* algorithm. The major steps of the *UdpOp* algorithm are outlined in Figure 5.

Step 1:	For each subset of NN-UDPs, do the following: - convert NN-UDPs in the subset to NN-OPs; - push NN-OPs as close to leaves as possible; - optimize with a traditional method by treating the output of NN-OPs as static relations.
Step 2:	For each plan found in Step 1, push join and selection into the NN-OPs, and re-evaluate the costs. Find the least costly plan from all the resulting plans.
Step 3:	Find the execution plan for the NN-OPs in the plan obtained in Step 2.

Fig. 5. Major steps of the *UdpOp* algorithm

Step 1 of *UdpOp* is to select a subset of the NN-UDPs to be converted to NN-OPs. In this step, we enumerate all subsets of the NN-UDPs in a query. We argue this is not too much overhead since in real applications with similarity-based queries, the numbers of UDPs should not be too large. In special cases where there are many UDPs, heuristics may need to be adopted to reduce the search space.

For each possible choice of converting NN-UDP to NN-OP, we push NN-OPs as much down to the leaves as possible. This is done for two reasons. The first is to give more flexibility for the traditional optimization algorithms (that deal with UDPs, i.e., all the remaining NN-UDPs). The second is that in an evaluation plan, the cost of an NN-OP does not usually depend where it is performed (there are exception, see below). We then treat each NN-OP as a static relation and hand over the query to an optimizer. This optimizer will treat NN-UDP as traditional UDPs.

In this paper, we assume the optimization algorithm of [3] is used. Since [3] only deals with selection-projection-join (SPJ) queries, we will restrict our queries to SPJ queries as well. The algorithm of [3] treats UDPs as selections and assumes the selectivity and the per-tuple-cost for each UDP, including NN-UDP, are known. For specific applications, we need to develop corresponding methods to provide such estimates.

In addition to the selectivity and per-tuple-cost of each UDP, we also need to know the size of each NN-OP since we treat it as a static relation. Furthermore, in the overall cost of a plan, the cost of deriving the result of the NN-OP should also be known.

Once the plan is obtained from Step 1, we try to push selections and joins into the NN-OP. The reason to do this is that sometimes, selection and join may reduce the number of query objects or even the pattern sets to be considered by the NN-OP. In some cases, the rank and threshold may also be reduced. For each plan obtained, we need to evaluate the overall cost. Step 2 will select the plan with the least cost.

After Step 2, we are "committed" to the evaluation plan. However, there is still some chance to refine the overall plan. Since we are dealing with SPJ queries, if the output of an NN-OP is empty, then the overall query will be empty and we can stop processing any other NN-OPs and the rest of the evaluation plan. Step 3 is mainly for this purpose. We assume we have an estimate of how likely an NN-OP will yield empty result, in addition to its cost. This is of course related to the size of the NN-OP (i.e., empty means 0 tuples) but a more specific statistic.

To conclude this section, we summarize the statistics that we need for the *UdpOp* algorithm: (1) cost per tuple of each NN-UDP, (2) selectivity of each NN-UDP, (3) cost of each NN-OP, (4) size of the output relation of each NN-OP, and (5) probability of having empty output of each NN-OP. For different applications, we need to develop different methods to obtain these statistics. In [7], we provided a method for some streaming time series cases. Also note, since similarity-based searches are typically very time consuming, we omit the cost of obtaining the statistics and running the optimization procedure when performance of query evaluation is assessed.

4 Experimental Results

To assess the effectiveness of our optimization algorithm, we implemented the algorithm proposed in this paper in C/C++. In addition, for similarity-based search with streaming time series, we developed a method to estimate the statistics used by the optimization algorithm (see [7] for detail). In this section, we present the results of performance evaluation with our *UdpOp* algorithm through three experiments. More detailed experimental results can be found in [6].

All experiments are performed on a desktop box with PIII 1.2GHz CPU and 512M memory, running Windows XP Professional. Since we use the relative costs to evaluate the performance, the hardware environment does not make much difference. Because similarity-based queries are mostly CPU bound in all experiments, we load the data into the memory in advance and thus only the computational costs are measured.

NN-UDP on streaming time series. A *streaming time series* is an infinite sequence of real numbers that continuously arrive at a query system. At each time position T, the streaming time series takes the form of a finite sequence, assuming the last real number is the one that arrived at the time position T.

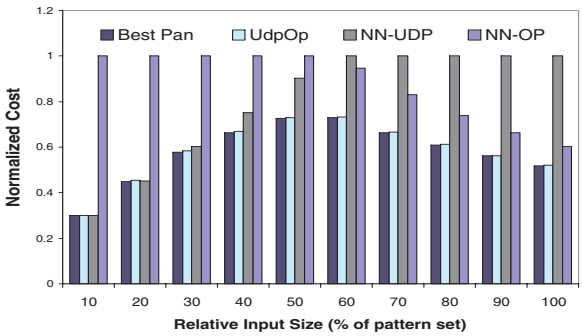

Fig. 6. Optimization comparison for Exp. 1

An NN-UDP (see Section 2.2) may use a streaming time series s as its query object (i.e., QID). In this case, we assume an implicit positive integer parameter w, and the NN-UDP is evaluated at each time position T by using the w-suffix of s up to (and including) time T. Hence, we assume each query containing this kind of NN-UDP need to be evaluated at each time position T when a new value of s arrives.

For our running example, the streaming time series are school attendance stream and pharmacy data stream whose data are continuously collected and sent to a query system[3]. Queries like those in Examples 1 and 2 will be evaluated each time when new school attendance and pharmacy data arrive.

Data generation. We use synthetic time series data sets in the experiments. We generate two types of data: the pattern time series and the streaming time series.

We first use a random-walking function to generate each pattern time series independently. The length of each pattern time series is between 50 and 300. Next, we construct each pattern set by randomly choosing 10^5 patterns. Finally, we construct the streaming time series. To form a streaming time series, we choose some patterns from one pattern set and then concatenate them into one sequence by interpolating a curve between two successive patterns. In choosing patterns from a pattern set, we follow some predefined probability distribution so that some patterns appear in the streaming time series more often than the others. To simulate the real world cases, we also add some white noise into each constructed streaming time series.

In the following experiments, we run each query multiple times and use the average cost to measure the performance of each optimization method. In each evaluation, we first build the estimation models for the statistics that will be used by *UdpOp* algorithm, by running the query for 1000 time positions [7]. We then implement different methods and run the queries for another 500 time positions (called *one run*).

Experiment 1. In the first experiment, we use the query in Example 2. As discussed earlier, there are three possible ways to generate the query plan for this query: (1) NN-UDP: treating UDP as NN-UDP; (2) NN-OP: treating UDP as NN-OP; and (3) *UdpOp*:

[3] We assume these two streaming time series are synchronized.

obtaining the plan using *UdpOp* optimization algorithm. Note that UDP may be treated as NN-UDP or NN-OP in (3), depending on the statistics.

In the experiment, we vary the size of relation $Events$ so that the number of tuples that are fed into the UDP varies from 10% to 100% of the size of pattern set $schoolSet$. For each case, we execute the query using each of the three plans and measure the cost. We also measure the cost of the "best plan", which is obtained by trying all possible plans and choosing the least-costly one.

The results are shown in Figure 6, which are normalized by the maximum cost for the respective input sizes, ranging from 5ms (for the "10% of pattern set" case) to 120 ms (the "whole pattern set" case). We can see that both (1) and (2) may result in very costly plans in some cases while our *UdpOp* algorithm always achieves good plans.

Experiment 2. In the second experiment, we change the schema of relation $Events$ to represent more complicated rules. The new schema is $(\mathsf{EID}, \mathsf{eName}, \mathsf{A1}, \mathsf{A2}, ..., \mathsf{An})$, where attribute Ai $(1 \leq \mathsf{i} \leq \mathsf{n})$ refers to patterns in pattern set $\mathsf{set_i}$. We then populate the $Events$ table and consider the following query for streaming time series $\mathsf{d1}, \mathsf{d2}, ..., \mathsf{dn}$.

```
SELECT E.eName
FROM   Events E
WHERE  NN('d1', 'set1', E.A1, K1, TH1)
  AND  NN('d2', 'set2', E.A2, K2, TH2)
       ...
  AND  NN('dn', 'setn', E.An, Kn, THn)
```

We consider two independent pairs of stream and pattern set, and hence two UDPs. We set up the data so that, dependent on the data in the *Events* table, the near/nearest-neighbor patterns discovered for one of the two streams may or may not help to narrow down the scope of the patterns to be considered for the other stream. More specifically, we generate multiple *Events* relations by using random subsets of the Cartesian Product of two sets (each having 10^5 patterns), one for each of the two Ai columns. The ranks in both UDPs are NULL and the thresholds are both set to 0.07. Two streaming time series are randomly chosen from a set of streams.

There are three possible optimization methods for comparison with the *UdpOp* algorithm, which are, (1) 2-NN-UDP Method: both UDPs are NN-UDPs, (2) 2-NN-OPs Method: both UDPs are converted into NN-OPs, and (3) 1-UDP/1-NN-OP Method, that is, we always convert one UDP into NN-OP and keep the other as NN-UDP.

We run this experiment 300 times. For each run, we use one of the randomly generated *Events* relations and the randomly chosen streaming time series and continuously evaluate the query for all 500 time positions. We take the average cost of these 300 runs as the performance measure for each optimization method. In addition, we also get the performance of the best plan, the least cost plan obtained by trying all possibilities. The result is in Figure 7, with all costs normalized by the maximum cost of 15 milliseconds. From this graph, it is clear that the performance of the *UdpOp* algorithm is very close to the best one.

Experiment 3. In this experiment, we use the same setup as for Experiment 2, i.e., we use the same schema of relation $Events$ and the same SQL query. In contrast to Experiment 2, we test the scalability of our algorithm when the number of UDPs goes up in the following situation. All UDPs deal with the same pattern set but with a different

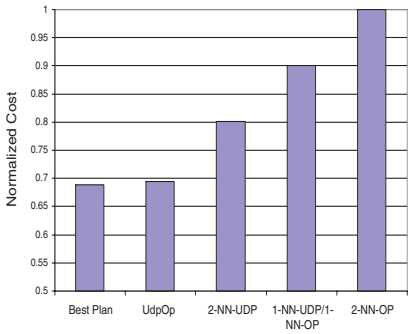

Fig. 7. Optimization comparison for Exp. 2

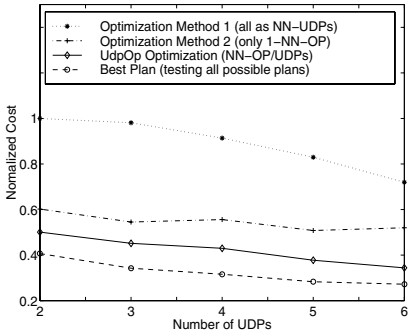

Fig. 8. Optimization comparison for Exp. 3

stream, and all thresholds are 0.07 and the rank of similarity is 1. Hence, we look for the pattern in the pattern set that is similar to *all* the streams. We vary the number of UDPs from 2 to 6, and set the size of the *Events* as 20% of the pattern set. We generate the *Events* relation such that no value appears twice in the same column. This guarantees that if we find a nearest neighbor with the evaluation of one UDP (under one column), then there is only one tuple in relation *Events* which needs to be fed into other UDPs. Clearly, in the optimized plan, other UDPs should remain NN-UDPs since only one tuple needs to be verified, and the ordering of UDPs and which UDP being converted into NN-OP will determine the performance.

Given the number of UDPs, we randomly pick up the same number of streams. Two other optimization methods are used for comparison. The first one is to treat all UDPs as selections, i.e., none of the UDPs are converted to NN-OP. Then we use the algorithm proposed in paper [3] to obtain a plan. More specifically, we use the estimation models to find the *rank*, as defined in [3], for each UDP and then get the ordering among them. The second optimization method will always convert only one NN-UDP into NN-OP, which is the one estimated to have the least cost among all possibly converted NN-OPs.

The performance comparison is shown in Figure 8, as the average cost of 210 runs for each number of UDPs. Again, we normalized all costs by a maximum value that is roughly 15 milliseconds. We can see that the performance of the plans obtained by the *UdpOp* algorithm is close to the best one.

5 Conclusion

In this paper, we introduced a user-defined predicate (UDP) for expressing queries involving similarity searches. We provided an optimization algorithm to derive efficient evaluation plans. In the (streaming) time series cases, our experiments demonstrated the good performance of our optimization algorithm.

We mainly focused on the situation where there are only a few query objects (e.g., time series and streaming time series). In our examples, we mostly used constants to represent them. We believe in real applications, this is mostly the case. However, for applications where the query series are massive, there are opportunities to further opti-

mize the queries. For example, when NN-OP is applied to many combinations of query series and pattern sets, many combinations may not find any pattern within the required threshold. This finding can be useful to optimize other NN-OPs in the same query. Same observation applies to the number of pattern sets.

References

1. R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In *FODO*, pages 69–84, 1993.
2. Surajit Chaudhuri and Luis Gravano. Optimizing queries over multimedia repositories. In *SIGMOD Conference*, pages 91–102, 1996.
3. Surajit Chaudhuri and Kyuseok Shim. Optimization of queries with user-defined predicates. *ACM Transactions on Database Systems*, 24(2):177–228, 1999.
4. D. Chimenti, R. Gamboa, and R. Krishnamurthy. Towards an open architecture for LDL. In *VLDB Conference*, 1989.
5. C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD Conference*, pages 419–429, 1994.
6. L. Gao, M. Wang, X. S. Wang, and S. Padmanabhan. Expressing and optimizing similarity-based queries in SQL. Technical Report CS-04-06, University of Vermont, http://www.cs.uvm.edu/csdb/techreport.shtml, March, 2004.
7. L. Gao, X. S. Wang, M. Wang, and S. Padmanabhan. A learning-based approach to estimate statistics of operators in continuous queries: a case study. In *Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD)*, 2003.
8. Joseph M. Hellerstein. Practical predicate placement. In *SIGMOD Conference*, pages 325–335, 1994.
9. Joseph M. Hellerstein and Michael Stonebraker. Predicate migration: optimizing queries with expensive predicates. In *SIGMOD Conference*, pages 267–276, 1993.
10. Eamonn J. Keogh, Kaushik Chakrabarti, Sharad Mehrotra, and Michael J. Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. In *SIGMOD Conference*, 2001.
11. D. Rafiei and A. Mendelzon. Similarity-based queries for time series data. In *SIGMOD Conference*, pages 13–25, 1997.
12. N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD Conference*, pages 71–79, 1995.
13. T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD Conference*, pages 154–165, 1998.