A Compositional Framework for Mining Longest Ranges

Haiyan Zhao, Zhenjiang Hu, Masato Takeichi

METR 2002-05 May 2002

A Compositional Framework for Mining Longest Ranges

Haiyan Zhao, Zhenjiang Hu, Masato Takeichi
Department of Information Engineering
University of Tokyo
7-3-1 Hongo, Bunkyo-ku
113-8656 Tokyo, Japan
{zhhy,hu,takeichi}@ipl.t.u-tokyo.ac.jp

Abstract

This paper proposes a *compositional* framework for mining interesting range information from huge databases, in which a domain specific query language is provided to specify the range of interest, and a *general* algorithm is given to mine the range specified in this language quickly. A wide class of longest range problems, including the intensively studied optimized support range problem [FMMT96], can be solved efficiently and systematically in this framework. Experiments with real world databases have been done and the results show that our framework is efficient not only in theory but also in practice.

Keywords Data Mining, Optimized Ranges, Longest Ranges, Longest Segment Problem, Program Calculation, Multidimensional Minima Searching.

1 Introduction

The last two decades have seen an explosive growth in our capabilities to both generate and collect data. The computerization of many business and government transactions have flooded us with information, and generated an urgent need for new techniques and tools that can automatically assist us in transforming this data into useful knowledge. There has been quite a lot of research [FPSM92, FPSSR96, SMJ95], among which the well known data mining gains much interest whose general goal is to extract interesting correlated information from large collections of data.

In this paper, we examine a new mining problem, called *longest range problem*, for discovering interesting range information from a huge database. Consider a relation, namely *callsDetail*, in a telecom service provider database containing detailed call information. The attributes of the relation may include *date*, *time*, *src_city*, *src_country*, *dst_city*, *dst_country* and *duration*. A simple tuple in the relation captures information about the two ends of each call, as well as the temporal elements of the call.

Some useful knowledge hidden in this relation can be used by the telecom service provider to make some decision. Suppose the telecom service provider is interested in offering a discount to customers. In this case, the timing of discount may be critical for its success, for example, to make the campaign more profitable, it would be advantageous to offer it close to a time interval in which the number of calls is not so few, say, during one year the number of calls exceeds 10000. And the telecom service provider wants to do this campaign in a time interval as long as possible, that is, to find the longest time interval which satisfies the above conditions. This can be described as

$$\begin{array}{l}\textbf{find longest } \mathit{time} \textbf{ range}\\ \textbf{from } \mathit{callsDetail}\\ s.t.\ \ \mathit{count}(*) \geq 10000.\end{array}$$

This problem is practically important, but finding an efficient and correct algorithm is very difficult. This has been seen by [FMMT96], which took great pains in solving *optimized range problem*, a special case of our *longest range problem* (see Section 5). Things turn out to be more interesting and difficult if we want to find the longest time interval under a more complicated condition, say, for the above example, to encourage the customers to make long time calls, besides the number of calls are big enough, the average talking time during the interval is not so long, say, less than 10 minutes, that is, the average of *duration* is less than 10 minutes.

$$\begin{array}{l}\textbf{find longest } \mathit{time} \textbf{ range}\\ \textbf{from } \mathit{callsDetail}\\ s.t.\ \mathit{count}(*) \geq 10000\ \ \wedge\\ \qquad \mathit{average}(\mathit{duration}) < 10\end{array}$$

However, as far as we know, this kind of problems has not been systematically solved yet, and it remains open how efficiently it can be solved.

In this paper, we propose a *compositional* framework to address this kind of problems in a general way. The main contributions of this paper are as follows.

- First, we design a general and powerful range querying language for mining the longest ranges. A wide class of longest range problems, including the optimized range problem as intensively studied in [FMMT96, SBS99, RS99], can be expressed and solved by our language.
- Second, we show that the language can be efficiently implemented by using techniques in program calculation. And moreover, our solution demonstrates that efficient algorithms for solving the longest range problems can be *compositionally* built according to the structure of predicates used to specify range properties. This composite approach is in sharp contrast with the existing case-by-case study as in [Zan92, Jeu93, vdE90].

- Last but not least, we test our framework with a POS database from a coffee shop with two years sales data for mining various range information. The experiments with the real world databases demonstrate that our framework is efficient not only in theory but also in practice, and can be applied in data mining field.

The rest of this paper is organized as follows. First, in Section 2, we explain some notation conventions and basic concepts used in this paper, and highlight the problem and the goal of this paper. Then, in Section 3, we propose our language for specifying longest ranges, and explain how it can be efficient implemented in Section 4. Section 5 exemplifies a special application, while Section 6 demonstrates our experimental results. Finally, we conclude our work in Section 7.

2 Longest Range Problems

In this section, we apprehend the longest range problems and highlight the purpose of this work, after briefly reviewing our notational conventions.

2.1 Notation

For concise and clear specification, throughout this paper we will use Haskell [JH99], a purely functional language, as our notation to specify our problems as well as to express our algorithms. For those who are not familiar with Haskell, we briefly explain some basic conventions for functional language below.

Function application is denoted by a space and the argument which may be written without brackets. Thus $f\,a$ means $f\,(a)$. Functions are curried, and application associates to the left. Thus $f\,a\,b$ means $(f\,a)\,b$. Function application is regarded as more binding than any other operator, so $f\,a \oplus b$ means $(f\,a) \oplus b$, but not $f\,(a \oplus b)$. Function composition is denoted by a centralized circle $\circ$, by definition, $(f \circ g)\,a = f\,(g\,a)$. Function composition is an associative operator. Infix binary operators will often be denoted by $\oplus, \otimes$ and can be *sectioned*; an infix binary operator like $\oplus$ can be turned into unary functions by $a \oplus b = (a\oplus)\,b = (\oplus\,b)\,a = (\oplus)\,a\,b$.

Lists are finite sequences of values of the same type. A list is either empty, a singleton, or the concatenation of two other lists. We write $[\,]$ for the empty list, $[a]$ for the singleton list with element a, and $x +\!\!+\, y$ for the concatenation of list x and y. Concatenation is associative, and $[\,]$ is its unit. For example, $[1] +\!\!+ [2] +\!\!+ [3]$ denotes a list with three elements, often abbreviated to $[1, 2, 3]$. We also write $a : x$ for $[a] +\!\!+\, x$.

For a nonempty list xs, function $head\ xs$ returns the first element, while $last\ xs$ gives the last element. Function $map\ f\ xs$ is used to establish a mapping for list xs (by applying function f to each element), its definition is as follows:

$$map\ f\ xs \ = \ [f\ x \mid x \leftarrow xs].$$

Function *filter* $p\ xs$, removing elements of a list that do not satisfy the predicate p, is defined as:

$$\mathit{filter}\ p\ xs \quad = \quad [x \leftarrow xs \mid p\ x].$$

2.2 Specification

Let us move on to the longest range problem. In fact, it is not a new problem at all. After preprocessing like bucketing according to the concerned range attribute proposed in [FMMT96], it boils down to the longest segment problem [Zan92, Jeu93] known in program calculation community, if we regard a relation in database as a list of records (tuples). It can be concisely specified by

$$\begin{array}{lcl} \mathit{lrp} & :: & ([\alpha]^+ \rightarrow \mathit{Bool}) \rightarrow [\alpha]^+ \rightarrow \mathit{Int} \\ \mathit{lrp}\ p\ xs & = & \mathit{maximum} \circ \mathit{map}\ \mathit{length} \circ \mathit{filter}\ p \circ \mathit{segs}\ xs \end{array}$$

It accepts a predicate p and an input list, and computes the longest range by the following steps:

- firstly enumerating all segments (contiguous sublists, also called *range* in this paper) of the input list by function *segs*,
- then keeping those satisfying the predicate p by *filter*,
- finally computing the length by *length* of the satisfied segments, and returning the largest value as the result by *maximum.*

Here, function *segs* for enumerating all segments can be straightforwardly defined as in [Bir89] by

$$\mathit{segs} \;=\; \mathit{concat} \;\circ\; \mathit{map}\ \mathit{inits} \;\circ\; \mathit{tails}.$$

among which, *inits* returns the list of all the initial segments of a given list, *tails* returns the list of all subsequences of a given list, and *concat* concatenates a list of lists into one long list.

$$\begin{array}{lcl} \mathit{concat} & :: & [[\alpha]] \rightarrow [\alpha] \\ \mathit{concat}\ xss & = & [x | xs \leftarrow xss; x \leftarrow xs] \end{array}$$

To simplify our presentation, we use the *length* of the longest segment as the result rather than the longest segment itself, and we assume that the input list to *lrp* is nonempty, as indicated by $[\alpha]^+$ in the type declaration of *lrp.*

Note that although *lrp* is an executable Haskell program, this straightforward definition serves just as the specification instead of a real solution, because it is too inefficient (the number of segments is about $n^2/2$, the algorithm require $O(n^2)$ computations of p on the segments) to be applied to data mining where database is always huge. Practical use of it requires that such computation should use less than $O(n^2)$ time, where n denotes the size of the concerned relation.

Unfortunately, it is known that there does not always exist an efficient algorithm for computing *lrp* p for any p [Zan92]. However, if p has a particular shape, or has

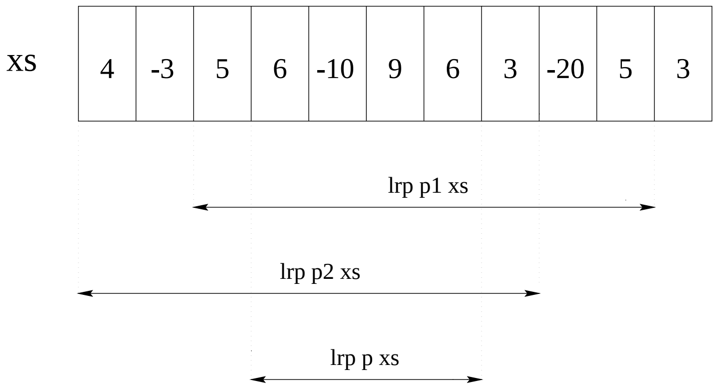

Figure 1: A Compositional Example

some nice properties, then a strong optimization can often be made, resulting in an efficient algorithm using less than O(n^2) time or even linear time.

Although solutions to various kinds of segment problems have appeared in the literature [Bir89, vdE90, Jeu93, Rem85, Rem86, Rem88, Zan92], one major problem which prevents them from being used in mining longest ranges is that their solutions are not *compositional*. Consider, for example, to compute the longest range z from a given list xs that satisfies the following two conditions:

(p_1) the two end elements in the range are equal, i.e.,

$$p_1\ z\ =\ (head\ z)\ =\ (last\ z);$$

(p_2) the sum of the elements in the range is no less than 10, i.e.,

$$p_2\ z\ =\ (sum\ z)\ \geq\ 10.$$

Using *lrp*, we can specify the problem by

$$\begin{array}{l} lrp\ p\ xs \\ \quad \textbf{where}\ p\ z\ =\ p_1\ z\ \wedge\ p_2\ z \end{array}$$

where p is defined by using two simpler predicates p_1 and p_2. Now supposing that we have got ranges for both $lrp\ p_1\ xs$ and $lrp\ p_2\ xs$. Unfortunately, we cannot obtain the result for $lrp\ p\ xs$ from those for $lrp\ p_1\ xs$ and $lrp\ p_2\ xs$. Figure 1 exemplifies the result for $lrp\ p_1\ xs$, $lrp\ p_2\ xs$, and $lrp\ p\ xs$ respectively, where xs is the input list. The longest range satisfying p_1 is $[5, 6, -10, 9, 6, 3, -20, 5]$, and that satisfying p_2 is $[4, -3, 5, 6, -10, 9, 6, 3]$, whereas the longest range satisfying both p_1 and p_2 is $[6, -10, 9, 6]$, which cannot be derived from $[5, 6, -10, 9, 6, 3, -20, 5]$ and $[4, -3, 5, 6, -10, 9, 6, 3]$.

It thus demonstrates that knowing efficient solutions to *lrp* with specific kinds of predicates is not helpful for solving *lrp* with the combination of these kinds of predicates.

Therefore, the goal of this work is to provide a *compositional* framework, in which,

p	$::=$	$\mathsf{f}\ (head) \olessthan \mathsf{g}\ (last)$	`End Element Relation Property`
	$\mid$	$(sum\ \mathsf{attr}) \olessthan \mathsf{c}$	`Sum Property`
	$\mid$	$(average\ \mathsf{attr}) \olessthan \mathsf{c}$	`Average Property`
	$\mid$	$(count\ \mathsf{attr}) \olessthan \mathsf{c}$	`Count Property`
	$\mid$	$(min\ \mathsf{attr}) \olessthan \mathsf{c}$	`Min Property`
	$\mid$	$(max\ \mathsf{attr}) \olessthan \mathsf{c}$	`Max Property`
	$\mid$	$p_1 \ \wedge\ p_2$	`Conjunction`
	$\mid$	$p_1 \ \vee\ p_2$	`Disjunction`
	$\mid$	$not\,(p)$	`Negation`

Figure 2: Predicates for Specifying Range Property

- Users can specify a wide class of longest range problems in a straightforward way, by using predicates composed of the primitive ones using the predicate connectives of $\wedge$, $\vee$ and *not*.
- Efficient solutions for implementing the above specification are guaranteed to exist and can be constructed systematically.

As a result, the problem turns to be two folds. One is how to choose a wide class of primitive predicates, in the sense that it can not only be solved efficiently, but also be used as the basis for building the compositional ones. And the other is how to construct efficient solution for the longest range specified by these predicates.

3 A Range Querying Language

To find the longest ranges of interest from databases efficiently and easily, the first step is how to specify the desired longest ranges.

3.1 The find Statement

Our range query language for the user to specify the range of his interest resembles SELECT in SQL.

find longest *attr* **range**
from *tab*
where *property*

in which, **find**, **longest**, **range**, **from** and **where** are reserved words, whereas the other three are specified by user. Roughly speaking, it finds the longest range of the attribute *attr* from the relation *tab* under the condition given by *property*. Note that *attr* must be a numeric attribute of the relation *tab*.

It is evident that both *attr* and *tab* come from the database under consideration. However, how to specify *property* is not so easy as it looks. Just as discussed in

Section 2, it cannot be guaranteed that there always exists an efficient algorithm to compute the longest range with respect to any *property*. It is thus crucial how to define proper *property*.

3.2 Querying Longest Ranges

We design a class of predicates, as shown in Figure 2, for describing the range properties. In Figure 2, $\olessthan$ denotes a *transitive total order* relation, like $\leq$, and f, g, c, attr and $\olessthan$ are all *changeable* and can be specified by user. Our predicates are classified into three groups:

- the primitive predicates describing the two end element relation property,
- the aggregate predicates specifying constraints on aggregation result on an attribute (field), and
- the composite predicates combining the former two kinds of predicates.

The former two groups are used to specify simple and concise properties, while the third group is for expressing more complicated properties in a compositional manner.

Primitive Property on Two-End Element Relation

We start by explaining the primitive property we can specify. As argued in [Zan92], one of the simplest but very useful range property is the order relation between two end elements (the leftmost element and the rightmost element) of the range, based on the fact that for many application, one is often able to put suitable information about the range to the two end elements by some preprocessing.

Therefore we provide the primitive predicate

$$\mathsf{f}\ (head) \olessthan \mathsf{g}\ (last)$$

to describe this property, meaning that the leftmost element (after applied function f) of the range z has a *transitive total order* relation $\olessthan$ with the rightmost element (after applied function g) of z. f and g can be any functions. Being simplest, this relation on two end elements can be used to describe many interesting ranges.

Example 1 In the telecom relation *callsDetail* given in Introduction, we may want to find the longest *time* interval in which the average of the calling time *duration* for the first and the last call is no less than 10 minutes

$$(head.duration + last.duration)/2 \ \geq \ 10$$

which can be reformulated to

$$head.duration \ \geq \ 20 - last.duration.$$

Thus this query can be specified in our language by

> **find longest** *time* **range**
> **from** *callsDetail*
> **where** *head.duration* $\geq$ $20 - $ *last.duration.*

Note that both *time* and *duration* are attributes of the relation *callsDetail*, and $x.y$ means to get the the value of attribute y from the record x. It is worth noting that we do not actually need to write f and g explicitly in the property description as seen above. If lhs is an expression computing on *head* and rhs an expression computing on *last*, then such f and g can be automatically derived: the f and g derived from

$$head.duration \ \geq \ 20 - last.duration$$

is

$$\begin{aligned} f\ x &= x.duration \\ g\ x &= 20 - x.duration. \end{aligned}$$

It should also be noted that [Zan92] only allows the case where $\mathsf{f} = \mathsf{id}$ and $\mathsf{g} = \mathsf{id}$. Our extension enables us to solve more complicated relations between two end elements as shown above.

Aggregation Property

Aggregate functions *sum*, *average*, *count*, *min* and *max* are often used to describe searching conditions in the database community. We therefore provide them for specifying the aggregate range properties. The general form is

$$(agg\ \mathsf{attr}) \oslash \mathsf{c}$$

where *agg* is an aggregate function, $\oslash$ a total transitive order, attr an attribute, and c a constant value. Suppose the relation $\oslash$ be $\leq$, then this property means that aggregate computation over the attribute attr in the range should be no more than c.

Example 2 For the same relation of *callsDetail*, consider to find the longest time interval during whose total calling time is less than a given threshold, say 50 hours. We may specify this query by

> **find longest** *time* **range**
> **from** *callsDetail*
> **where** (*sum duration*) $<$ $50 * 60$.

Composite Property

Though we can describe many interesting range properties using the primitive property and aggregate predicates, the practical usage needs to specify more complicated and interesting range properties. For this purpose, we provide three composite predicates $\wedge$, $\vee$, and *not* in our language to combine the primitive predicate and aggregate predicates easily, the precedence of which is descending from *not* to $\vee$.

Just as their names suggest, *not* denotes the logical negation of predicate, which means to compute the longest range not satisfying the given predicate followed *not*, and $\wedge$ is used to describe the logical conjunction of predicates, which requires the concerned segment should satisfy each components of $\wedge$ simultaneously, while for the logical disjunction predicate $\vee$, it's enough for the concerned segment to satisfy any of its components.

Example 3 Recall the example in Introduction, which can be coded easily in our language by

find longest *time* **range**
from *callsDetail*
where $count(*) \geq 10000 \ \wedge$
    $average(duration) < 10.$

Example 4 For the example in Section 2, we can regard the list xs as a relation with a single attribute, say *value*, then it can be coded in our language as follows.

find longest *value* **range**
from xs
where
    $(head.value) \ \leq \ (last.value) \wedge$
    $not \ ((head.value) \ \leq \ (last.value)) \wedge$
    $(sum \ value) \ \geq \ 10.$

Note that $(head.value) \ \leq \ (last.value)$ is an abbreviation for $f(head) \ \leq \ f(last)$, where f is defined by $f \ x = x.value$.

Consequently, our range query language, for its compositional feature, is powerful and easy for user to specify a wide class of longest ranges.

Remark

It is worth noting that we discuss only the properties related with computing the range, and omit the general selection conditions, which, in fact, can be easily filtered by preprocessing. For example

find longest *time* **range**
from *callsDetail*
where
    $sum\,(\,duration) < 50 * 60 \ \wedge$
    $src\_city = Tokyo \ \wedge$
    $dst\_city = HongKong$

can be transformed to

> **find longest** *time* **range**
> **from** *callsDetail′*
> **where**
> $sum\,(duration) < 50 * 60,$

and *callsDetail′* is a view defined in SQL as

$$\begin{array}{l} SELECT\ * \\ FROM\ callsDetail \\ WHERE\ src\_city = Tokyo\ AND \\ \quad\quad dst\_city = HongKong \end{array}$$

In fact, this preprocessing does not raise any additional cost, for it is fused into the process of bucketing, which will be discussed concisely in next section.

4 Implementing the Range Querying Language

This section explains how to implement our query language in an efficient way. Our result can be summarized in the following theorem.

Theorem 1

> **find longest** *attr* **range**
> **from** *tab*
> **where** *property*

can be implemented using at most

$$\mathrm{O}(n \log^{k-1} n)$$

time, if every f and g used in the definition of primitive predicates inside property p can be computed in constant time. Here n denotes the size of *relation* and k is a constant depending on the definition of *property* (Lemma 2).

□

We prove this theorem by giving a concrete implementation, which consists of the following four phases:

- Bucketing the relation,
- Normalizing the range property,
- Refining the longest range problems,
- Computing the longest range.

In the rest of this section we will explain these phases one by one.

4.1 Bucketing the Relation

Our idea is to bucket the relation according to the range attribute and reduce the query to a longest segment problem whose input is a list of tuples.

Before showing how to solve longest segment problem, we briefly explain the bucketing process. This bucketing process is necessary, because the original relation may not be sorted in the order of the range attribute. For instance, the relation *callsDetail* may be in the order of $(date, time)$, but for the query like

$$\begin{array}{l}\textbf{find longest } time \textbf{ range} \\ \textbf{from } callsDetail \\ \textbf{where } (sum\ duration) \ < \ 50,\end{array}$$

we need to sort the relation according to *time* in order to compute the *time* range. As discussed in [FMMT96], rather than using expensive sorting algorithm, we may apply some efficient bucketing algorithm (almost linear) to achieve it. We adopt the algorithm in [FMMT96], and to keep the information for later aggregate computation, we extend the algorithm by associating with each bucket with the number of records it has and the summation of the attributes manipulated by aggregate functions in the property description, its cost is the same as that in [FMMT96] (linear) by using fusion technique [Chi92, OHIT97].

The remainder of this section will be explained on the ground that the relation has been bucketed into equal-size blocks according to the range attribute.

4.2 Normalizing the Range Property

For efficient implementation, the range property specified by user should be first normalized into canonical form to eliminate the redundancy.

Elimination of Aggregate Functions

Recall the example in Section 2, where the second condition p_2, saying that the sum of the elements of the range is no less than 10, which is expressed by our predicate as

$$(sum\ \ xs)\ \geq 10.$$

How to eliminate this *sum* function? The trick is to use *scanl1* [1]

to compute every prefix sum of the input list xs and get a new list ss = *scanl1* $(+)$ xs:

$$\begin{array}{l}[x_1, x_2, \ldots, \underline{x_h}, \ldots, \underline{x_l}, \ldots] \\ [s_1, s_2, \ldots, \underline{s_h}, \ldots, \underline{s_l}, \ldots].\end{array}$$

[1] *scanl*1 $(\oplus)$ xs accumulates function $(\oplus)$ for every initial segment of the given list xs, for example,

$$scanl1\ (+)\ [1,2,3,4,5,6] \ = \ [1,3,6,10,15,21]$$

To compute the sum of a range $xs' = [x_h, \ldots, x_l]$, we may use the end elements of xs' and corresponding ss', i.e., x_h, s_h, x_l, s_l:

$$\mathit{sum}\ xs' = x_h + (s_l - s_h).$$

Thus, $\mathit{sum}\ xs' \geq 10$ is coded as $10 + s_h - x_h \leq s_l$. Accordingly, for any segment z, we can use a preprocessing as the following function

$$\mathit{preproc}\ z = \mathit{zip}\ z\ (\mathit{scanl1}\ (+)\ z)$$

to make a new segment with each element changed to a pair. Here, *zip* is a function tupling two lists into one by taking a pair of lists and returning a list of pairs of the corresponding elements, for instance,

$$\begin{array}{l} \mathit{zip}\ [1,2,3,4,5,6]\ (\mathit{scanl}1\ (+)\ [1,2,3,4,5,6])\ = \\ \quad [(1,1),(2,3),(3,6),(4,10),(5,15),(6,21)]. \end{array}$$

With the same trick, we can eliminate the other four. Consequently, the aggregate predicates can be expressed in the form of $\mathsf{f}\ (\mathit{head}) \otimes \mathsf{g}\ (\mathit{last})$.

It should be noted that this preprocessing does not raise additional cost by using accumulation and fusion techniques [Bir84, Chi92].

Normalization of Composite Property

Next, let us turn our attention to the composite property. With respect to it, we have the following Lemma holds.

Lemma 2 (Normalization) Any composite predicate can be expressed in its canonical form, that is, the disjunction of a number of conjunction of primitive predicates, i.e.,

$$\begin{array}{rll} p\ x = & p_{11}\ x \wedge p_{12}\ x \wedge \ldots \wedge p_{1k_1}\ x & \vee \\ & p_{21}\ x \wedge p_{22}\ x \wedge \ldots \wedge p_{2k_2}\ x & \vee \\ & \ldots\ldots & \vee \\ & p_{m1}\ x \wedge p_{m2}\ x \wedge \ldots \wedge p_{mk_m}\ x & \end{array}$$

where, p_{ij} is primitive predicate, and the maximum of k_1, k_2, ..., k_m is exactly the k in Theorem 1.

Proof sketch. With the aggregate functions transformed into $\mathsf{f}\ (\mathit{head}) \otimes \mathsf{g}\ (\mathit{last})$ form, the first two groups of predicates in nature have only one form. In addition, by A. de Morgan's law, as well as the distributive and associative law on logic operators, we can get that each p_{ij} in the above formula is either $\mathsf{f}\ (\mathit{head}) \otimes \mathsf{g}\ (\mathit{last})$ or $\mathit{not}\,(\mathsf{f}\ (\mathit{head}) \otimes \mathsf{g}\ (\mathit{last}))$.

According to the semantics of *not*, we can get that

$$\mathit{not}\,(\mathsf{f}\ (\mathit{head}) \otimes \mathsf{g}\ (\mathit{last}))\ =\ \mathsf{f}\ (\mathit{head}) \otimes^{\mathsf{C}} \mathsf{g}\ (\mathit{last}),$$

here we use $\mathcal{R}^C$ to denote the complement relation of $\mathcal{R}$. Because $\otimes$ is a transitive total order relation, so $\otimes^C$ is also *transitive total order.* Thus we can say that for any primitive predicate p, *not* p is also primitive. Consequently, this lemma is self-evident.

□

4.3 Refining the Longest Range Problem

Lemma 2 shows that a range property specified by our language can be normalized into a disjunction of simpler components, which is either a primitive one in the form of $\mathsf{f}\ (head) \otimes \mathsf{g}\ (last)$ or a conjunction of several primitive ones. On the ground of this normalization, what we need to do is how to deal with these three cases. If we can address both the primitive and the conjunction case, then the disjunction case is easy to solve, for it computes the longest range that satisfies any of its components. We can get the result by calculating that for each component and select the longest one as the result. Thus, the crucial part is how to deal with the conjunction case and primitive case.

The primitive case is about a *transitive total order* relation on the two end elements of the range, while the conjunction case is to compute the longest range that satisfies a number of primitive predicates simultaneously, that is,

$$\begin{aligned} p\ z = \ & f_1\ (head) \otimes g_1\ (last)\ \wedge \\ & f_2\ (head) \otimes g_2\ (last)\ \wedge \\ & \ldots\ \wedge \\ & f_k\ (head) \otimes g_k\ (last) \end{aligned}$$

where k is the number of primitive predicates. If we capsule this composite conjunction by tupling all of its primitive components together, as follows,

$$(f_1, \ldots, f_k)\ (\mathcal{R}_1, \ldots,\ \mathcal{R}_k)\ (g_1, \ldots, g_k)$$

with each $\mathcal{R}_i$ is a *transitive total order* relation, and

$$\begin{aligned} & (x_1, ..., x_k)\ \ (\mathcal{R}_1, ..., \mathcal{R}_k)\ (y_1, ..., y_k) \\ \equiv\ \ & x_1\ \mathcal{R}_1\ y_1\ \ \wedge\ ...\ \wedge\ \ x_k\ \mathcal{R}_k\ y_k. \end{aligned}$$

Then, what we need to implement boils down to the following problem:

> Given a list, compute the length of a longest nonempty range such that the computation on the leftmost element is related with that on the rightmost element by a relation $\bar{\mathcal{R}}$ (which is not necessary to be a total order).
>
> $$\bar{f}\ (head)\ \bar{\mathcal{R}}\ \bar{g}\ (last)$$
>
> where,
>
> $$\begin{aligned} & \bar{f} = f_1 \vartriangle \cdots \vartriangle f_k \\ & \bar{g} = g_1 \vartriangle \cdots \vartriangle g_k \\ & (x_1, \ldots, x_k)\ \bar{\mathcal{R}}\ (y_1, \ldots, y_k) = \wedge_{i=1}^{k} x_i\ \mathcal{R}_i\ y_i \end{aligned}$$
>
> where every $\mathcal{R}_i$ be a *transitive total order* relation, and the split operator $\vartriangle$ be defined by
>
> $$(f_1 \vartriangle \cdots \vartriangle f_k)\ xs = (f_1\ xs, \ldots, f_k\ xs).$$

Notice that $\bar{\mathcal{R}}$ is usually a partial order, and that if $\bar{\mathcal{R}}$ itself is a transitive total order relation, it is just the primitive form $\mathsf{f}\ (head) \otimes \mathsf{g}\ (last)$, and a special case of it, where $f = id$ and $g = id$, is known as the leftmost at most rightmost problem [Zan92].

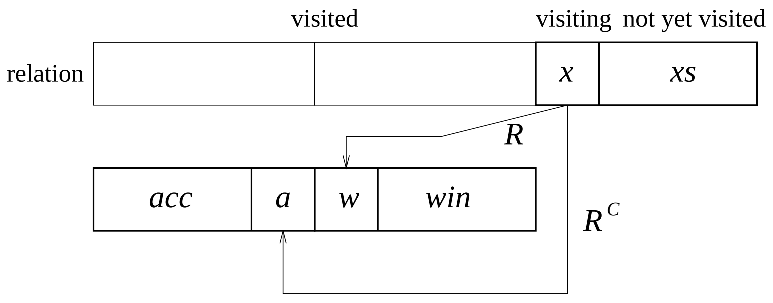

Figure 3: Demonstration of the solution

4.4 Computing the Optimized Ranges

To address this refined problem, let us first identify the core of it: How to determine whether the relation between the two end elements of the segment holds or not, and the end element is either a individual one (primitive case) or a tuple (conjunction case).

Main Point of our Solution

One naive solution is that in the process of scanning the input from left to right, for each element x under consideration, we compare it with every element preceding it, to determine whether the relation between this two elements holds, and then choose the longest segment (that is, whose left-end is the leftmost one among all the elements preceding it) that satisfies the relation as the result.

It is evident that this naive solution is inefficient, because an element has to be compared with all the prefix elements. Therefore, we consider reusing the earlier comparison results by accumulation. In addition, one fact about the longest range is that the length of the longest range sometimes increases but never decreases in the process of scanning the input, just as the maximum in a set.

Based on these two points (reusing the comparison results and the length only increasing), we can get a practical and efficient solution to the problem, which is motivated by the so-called windowing technique proposed by [Zan92] and is a significant extension to windowing technique.

Without loss of generality, henceforth suppose the relation $\circledcirc$ in the primitive predicate be $\leq$, the input be a list of number, and the result be the length of the length if the longest segment rather the segment itself.

To reuse the earlier comparison results, one measure of our solution is to build an additional storage to memorize the minima of all the elements that have been scanned before the current element; and at the same time, to take use of the feature that the length of the longest segment never decreases, the length of the current longest range is also memorized.

Figure 3 demonstrates this idea. $(acc +\!\!+ [a] +\!\!+ [w] +\!\!+ win)$ denotes the additional storage for the elements preceding x in descending order, x is the current element to be processed, and xs is the input not yet processed. The length of $([w] +\!\!+\, win)$ is just that of the current longest range.

Because the additional storage is organized in the descending order, to element x, we can skip the range win (whose length is that of the current longest segment), and only need to compare it with $acc +\!\!+ [a] +\!\!+ [w]$ backwards in the following way: if the relation $\mathcal{R}$ between x and w does not hold, then the longest length keeps unchanged, win slides rightwards by one element, and the leftmost element of win appends to $(acc +\!\!+ [a])$; Otherwise, we can increase the length and extend win leftwards while shrinking acc, and repeat this process until win cannot be extended, that is, there is no element in $(acc +\!\!+ [a])$ holding the relation $\mathcal{R}$ with x.

It is worth noting that the additional storage is built synchronously with the scanning of the input, so this algorithm is on-line in nature.

We illustrate this idea by a simple example of computing the length of the longest range whose leftmost element is less than or equal to its rightmost one. Assume the input is

$$xs = [4, -3, 5, 6, -10, 9, 6, 3, -20, \underline{5}, 3]$$

the same as that in Figure 1. To compute the longest range, we scan xs from left to right, while building an additional list ys, which contains the minimum of all elements that have been scanned before the element. For instance, when we finish scanning -20 and is scanning 5 (as underlined), the corresponding ys should be

$$ys = [\underline{4, -3}, \underline{-3, -3, -10, -10, -10, -10, -20}],$$

and the longest length obtained so far is 7. [2]

Now we divide ys into two parts with the second part (which is often called *window*) having the length of the longest range obtained so far. To see if there is any longer range including 5, we skip checking the elements in the window except the leftmost element -3. Since 5 is greater than -3, we know there must be a longer range containing 5, and we thus extend the window leftwards while shrinking the first part:

$$[\underline{4}, \underline{-3, -3, -3, -10, -10, -10, -10, -20, -20}].$$

Repeating this process until the window cannot be extended. After that, we move to scan the next element of the input till the end.

Algorithm for Computing Longest Range

From the above illustration, we can see the main operations include extending win, shrinking acc, as well as sliding win and inserting an element to acc. And which

[2] The longest range before scanning 5 is

$$[\underline{4, -3, 5, 6, -10, 9, 6}],$$

whose length is 7.

operations is enforced depends on the comparison result of the current element x and the elements in the additional storage. For example, for relation $\leq$, the key point is to determine whether there exists an element in $(acc +\!\!+ [a] +\!\!+ [w])$ less than x, in other word, whether x is minimum in $(acc +\!\!+ [a] +\!\!+ [w])$. If the answer is true, we extend *win* leftwards, and shrink *acc* simultaneously.

To be precise, we define the following operations for *acc* and *win*, respectively.

- *accIsmin* :: determines whether a given element is minimal in *acc*, if so, it returns a *true* value
- *accInsert* :: inserts an element to *acc*
- *accDelete* :: deletes an element from *acc*
- *accLastElement* :: returns the last inserted element

as well as

- *winExtend* :: extends *win* to left by one element and increases its length by one
- *winSlide* :: shifts *win* to right by one element and keep the length unchanged
- *winHdLeft* :: returns the first element of *win*

We can express our solution in the following Haskell code. flr is the function to find the longest range, which receives four parameters, the functions f and g being applied to the leftmost and rightmost elements respectively, the given relation r, and the input list $(x : xs)$.

```
flr f g r (x : xs) =
  flr' f g r accEmpty_r winInit xs
    where winInit = winExtend winEmpty x

flr' f g r acc win [] = length win
flr' f g r acc win (x : xs) =
  if accIsmin_r acc' (g x)
    then flr' f g r (accDelete_r acc lastAcc)
                (winExtend win lastAcc) (x : xs)
    else flr' f g r acc'
                (winSlide win (f x)) xs
  where hwin = winHdLeft win
        acc' = accInsert_r acc hwin
        lastAcc = accLastElement_r acc
```

Here, flr calls a recursive function flr', which use two additional parameters *acc* and *win* to memorize the intermediate results discussed above.

Notice that there are at most $2n$ recursive calls to flr' for an input list of length n, and that in each recursive call, besides the cost for f and g, the major computation time is spent on insertion and deletion of elements for *acc* and *win*, as well as the determination whether the current element is minimal (suppose that the given relation is $\leq$) in *acc*. In other words, the time cost of this algorithm depends on the efficiency of operations on *acc* and *win*. Suppose the time cost for these operations is $\mathcal{T}_p$, then function flr will be in $O(n\mathcal{T}_p)$ time.

While the time cost of these operations has great relations with the underlying data structures, so now the problem becomes how to choose proper data structure for *acc* and *win*.

For the primitive case, the relation ($\circledleq$) is in total order, in other words, if we assume $\circledleq$ be $\leq$, the minima is unique for a sequence of numbers, that is the minimum of the sequence. Therefore as shown in the figure, we can use an double-end list [Oka98] as the data structure for *acc* and *win*, it is known from [Oka98] that all these operations can be efficiently implemented in constant amortized time. Consequently, flr is an amortized linear algorithm for the primitive case.

On the other hand, the primary distinction from the primitive one is that the relation order in conjunction case is no longer total, it becomes a partial order on tuples, whose definition is derived from the orders of its components as:

$$(x_1, \ldots, x_k) \leq (y_1, \ldots, y_k) \;\equiv\; x_1 \leq y_1 \wedge \ldots \wedge x_k \leq y_k,$$

here, k is the number of the primitive predicates in the conjunction.

The real challenge for the partial order is that the minimum is *not unique*, which means that there may exist several minima at each point, as result we can not determine whether an element (e.g., x) is minima in the set (e.g., the minima at point w) in constant time. Rather than the double-end list *acc* used for total order, we need a more comprehensive data structure now to store information of the minima gotten at the point x. Therefore, we use a binary search tree, each of whose nodes corresponds to a k-tuple, to represent the minima at each point that is now a set of k-tuples. In this case, the determination becomes a minima searching (i.e., whether x is minima) in the tree, and at the same time the construction for the minima at each point involves the minima searching and insertion to the tree. Luckily all the trees for each point can be merged as a single one, this can be achieved simply by insertion and deletion to trees in $O(\log\ n)$ time.

Fortunately, the key problem boils down to the multidimensional minima searching problem, which is settled in [ZHM02] by using a so-called k-d-m tree in $O(\log^{(k-1)} n)$ time. k-d-m tree is a multidimensional search tree with minimum attribute. Each node of k-d-m tree represents a k-tuple in our case. So we adopt k-d-m tree as the structure for *acc*.

The time cost for the operations on k-d-m tree is summarized as follows.

- *accIsmin* :: $O(\log^{(k-1)} n)$
- *accInsert* :: $O(\log n)$
- *accDelete* :: $O(\log n)$
- *accLastElement* :: $O(1)$

And as to *win*, which stores the elements in *window* with its length equal to the current longest length, its data structure can use a double-ends list [Oka98]. The time cost for the operations on *win* are given below.

- *winExtend* :: amortized O(1) time
- *winSlide* :: amortized O(1) time
- *winHdLeft* :: amortized O(1) time

Based on the time costs for operations on *acc* and *win*, we can get that the time cost for the algorithm flr in conjunction case is $O(n \log^{(k-1)} n)$.

Theorem 1 has accordingly been proved. In the case of $k = 1$ (primitive case), the longest range can be computed in linear time.

5 One Application: Optimized Support Range Problem

This section will present one application of our framework for data mining. As a special case of our longest range problems, the optimized support range problem, first studied in [FMMT96] and got more studies in [SBS99, RS99], is very useful in extracting correlated information. For example, the optimized association rule for $callsDetail$ in the telecom database,

$$\begin{aligned}(\mathit{date} \in [d_1..d_2]) \wedge (\mathit{src\_city} = \mathit{Tokyo})\\ \Rightarrow\ (\mathit{dst\_city} = \mathit{HongKong})\end{aligned}$$

describes the calls from *Tokyo* during the date $[d_1, d_2]$ are made to *HongKong*. Generally, an optimized association rule R has the form of $A \in [l, u] \wedge C_1 \Rightarrow C_2$, where A is a numeric attribute, C_1 and C_2 are conjunction of conditions, and l and u are uninstantiated variables. Each rule has an associate *support* and *confidence* property, in which, *support*(C_i) be the ratio of the number of tuples satisfying C_i and the number of the tuples in the relation, *support*(R) is then the same as the support of $(A \wedge C_1 \wedge C_2)$, *confidence*(R) be the ratio of the supports of condition $A \wedge C_1 \wedge C_2$ and $A \wedge C_1$.

Suppose that the telecom service provider want to offer discounts to *Tokyo* customers who make calls to *HongKong* at a period of consecutive days in which the maximum number of calls from *Tokyo* are made and a certain minimum percentage of the call from *Tokyo* are to *HongKong*. This is known as the optimized support range problem, which can be described by

$$\begin{aligned}&\text{maximize } \mathit{support}(\mathit{rule})\\ &\quad \text{s.t. } \mathit{confidence(rule)} \geq \theta\end{aligned}$$

where *rule* is the optimized association rule given above, *support* and *confidence* are functions related with the *support* and *confidence* of the *rule*, and θ is a given constant.

To do this, first we preprocess the original relation by adding a new attribute called *support*. It is defined according to the above rule by

$$\begin{array}{rcll} \textit{support} & = & 1 & \textit{src\_city} = \textit{Tokyo} \ \wedge \ \textit{dst\_city} = \textit{HongKong} \\ & = & 0 & \text{otherwise} \end{array}$$

After the bucketing according to the range attribute *date*, it is just to compute the longest *date* range with respect to the average of *support* no less than the given θ. Thus, the optimized support range problem is only a special case of longest range problem, it can be simply expressed by our language as follows.

find longest *date* **range**
from *callsDetail*
where (*average support*) $\geq \theta$

From Theorem 1, we know that it can be solved in O(n) time.

Besides the optimized support problem, our framework can efficiently address many range mining problem, such as maximum gain problem.

6 Performance Evaluation

The proposed language and algorithm have been first tested in Haskell using the schemas described in previous sections. And for the practical application, it has further been re-implemented in Java using JDK1.3 with a friendly GUI for user to specify his range of interest, together with the mining conditions (range property), which is available at site `http://www.ipl.t.u-tokyo.ac.jp /rise2001/`.

Our efficient algorithm for mining longest ranges from database is built on the basis of the preprocessing for the original data, that is the bucketing according to the range attribute of interest. We mainly adopted the technique for bucketing proposed in [FMMT96], which is almost linearly with the number of data when the data sets get larger. Furthermore, for the practical purpose, according to the characteristics of the range attribute, the bucket size can be specified by user in our framework.

We evaluated the performance on a Panasonic CF-M2XR with a CPU clock rate of 650 MHz and 192 MB of main memory, running Microsoft Windows 2000 Professional 5.0. And the tested data is a real world database from a coffee shop with two-year sales data. Figure 4 shows the execution time and memory used when mining the longest range with the buckets numbers range from 10 to 3000. Here, we give the evaluated results for mining longest ranges where the number of primitive predicates the range property should satisfy simultaneously is from 1 to 5, respectively denoted by $k = 1, \ldots, k = 5$.

The experimental results demonstrate the efficiency of our algorithm and shows the promising application prospect. Particularly, the optimized support rule can be easily solved in linear time in our framework.

7 Conclusion

In this paper, we identify one important class of data mining problems called longest range problems, and propose a compositional framework for solving the problems efficiently. This work is a continuation of our effort to investigate how program calculation approach could be used in data mining [HCT00], and it confirms us with its promising result.

At present, our framework provide a wide class of predicates for specifying the range property, we want further to investigate how general predicates our approach can deal with. Another work we want to study is whether our framework can deal with multi-range rather than the current a single range attribute.

References

[Bir84] R. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.

[Bir89] R. Bird. Constructive functional programming. In *STOP Summer School on Constructive Algorithmics, Abeland*, 9 1989.

[Chi92] W.N. Chin. Safe fusion of functional expressions. In *Proc. Conference on Lisp and Functional Programming*, pages 11–20, San Francisco, California, June 1992.

[FMMT96] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Mining optimized association rules for numeric attributes. In *Proc. ACM PODS'96*, pages 182–191, Montreal Quebec, Canada, 1996.

[FPSM92] W. Frawley, G. Piatetsky-Shapiro, and C. Matheus. Knowledge discovery in databases: An overview. *AI Magazine*, pages 213–228, 1992.

[FPSSR96] U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R.Uthurusamy. *Advances in Knowledge Discovery and Data Mining.* AAAI/MIT Press, 1996.

[HCT00] Z. Hu, W.N. Chin, and M. Takeichi. Calculating a new data mining algorithm for market basket analysis. In *Second International Workshop on Practical Aspects of Declarative Languages, LNCS 1753*, pages 169–184, Boston, Massachusetts, January 2000. Springer-Verlag.

[Jeu93] J. Jeuring. *Theories for Algorithm Calculation.* Ph.D thesis, Faculty of Science, Utrecht University, 1993.

[JH99] S. Peyton Jones and J. Hughes, editors. *Haskell 98: A Non-strict, Purely Functional Language.* Available online: `http://www.haskell.org`, February 1999.

[OHIT97] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO. In *IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, pages 76–106, Le Bischenberg, France, February 1997. Chapman&Hall.

[Oka98] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[Rem85] M. Rem. Small program exercises. *Science of Computer Programming*, 5:309–316, 1985.

[Rem86] M. Rem. Small program exercises. *Science of Computer Programming*, 7:87–97, 1986.

[Rem88] M. Rem. Small program exercises. *Science of Computer Programming*, 11:167–173, 1988.

[RS99] R Rastogi and K. Shim. Mining optimized support rules for numeric attributes. In *15th International Conference on Data Engineering*, Sydney, Australia, March 1999. IEEE Computer Society Press.

[SBS99] R. Rastogi S. Brin and K. Shim. Mining optimized gain rules for numeric attributes. In *Proc. of ACM KDD'99*, 1999.

[SMJ95] A. Silberschatz, M.Stonebraker, and J.D.Ullman. Database reserach: Achievements and opportunities into the 21st century. In *NSF Workshop on the Future of Database Systems Research*, May 1995.

[vdE90] J.P.H.W. van den Eijnde. Left-bottom and right-top segments. *Science of Computer Programming*, 15:79–94, 1990.

[Zan92] H. Zantema. Longest segment problems. *Science of Computer Programming*, 18:36–66, 1992.

[ZHM02] H. Zhao, Z. Hu, and M.Takeichi. Multidimensional searching trees with minimum attribute. *JSSST Computer Software*, 19(1):22–28, Jan. 2002.

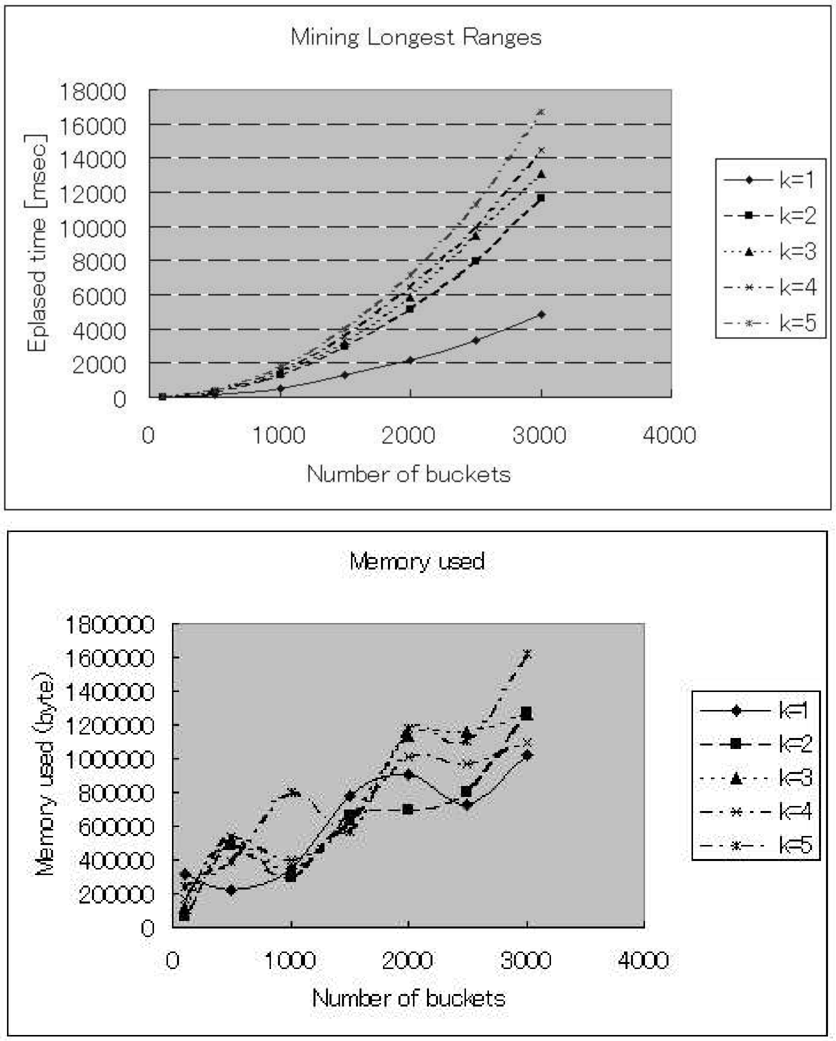

Figure 4: Performance of mining longest range