Mining Optimized Ranges: A Functional Approach (Preliminary Report)

Haiyan Zhao, Tetsuro Yokoyama, Zhenjiang Hu, Masato Takeichi
Department of Information Engineering
University of Tokyo
({zhhy,yokoyama,hu,takeichi}@ipl.t.u-tokyo.ac.jp)

Abstract

Data mining is a very important technique for discovering useful information from large collection of data. In this paper, we address a new kind of mining problems, called *optimized range problems*, for discovering interesting range information from huge databases. By making use of functional languages for clear specification of the problems, nice reasoning of program's property, and compositional features for constructing algorithms, we obtain a *general* and *constructive* solution to this kind of problems. In addition, by using our library, one can easily calculate optimized ranges of their interest in their daily programs.

Keywords: Data Mining, Program Calculation, Optimized Range Problem, Longest Segment Problem, Predicate, Functional Programming.

1 INTRODUCTION

The last two decades has seen an explosive growth in our capabilities to both generate and collect data. Advances in data collection, widespread use of bar codes for most commercial products, and 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 have been quite a lot of researches [FPSM92, FSSU96, SSU95], among which the well known data mining gains much interest whose general goal is to extract interesting correlated information from large collection of data.

In this paper, we examine a new mining problem, called *optimized range problem*, for discovering interesting range information from a huge database. This problem was greatly motivated by the study of the so-called optimized support range problem in [FMMT96]: given, for example, a huge database DB recording retail transactions of a supermarket, the optimized support range problem is to compute a longest date period during which the customers who bought milk often bought bread too. This can be directly described by

$$\begin{array}{l} \text{maximize } (d_2 - d_1) \\ \quad \text{s.t. } support(DB, rule) \geq \theta \end{array}$$

where *support* is a function to compute the support of an association rule in the database DB, θ is a given support threshold, and *rule* is defined by

$$Date \in [d_1..d_2] \wedge Milk = True \;\; \Rightarrow \;\; Bread = True$$

This problem is practically important, but finding an efficient and correct algorithm has proved to be much difficult [FMMT96]. Things turn out to be more interesting and more difficult if we want to find the longest ranges where in addition to the condition for the support of the rule, we require that the gain of the association rule should be above another threshold.

$$\begin{array}{l} \text{maximize } (d_2 - d_1) \\ \quad \text{s.t. } support(DB, rule) \geq \theta \;\wedge\; gain(DB, rule) \geq \theta' \end{array}$$

However, this problem cannot be solved in [FMMT96], and it remains unclear how to solve it efficiently.

In this paper, we aim at a general study of this kinds of optimized range problems:

> *Given a data sequence, find the longest range satisfying a certain property.*

By defining suitable properties, we can get a variety of interesting optimized range problems. Though being practically interesting, as far as we are aware, this problem has not been well developed and it remains open how general such problems can be dealt with. The existing work [FMMT96, FMMT96b, AIS93, BRS00] concentrates only on some *specific* problems, and provides special solutions for these specific problems one by one. It is certainly indispensable to put forth a solution with certain generality. The challenge is how to clarify the condition for the existence of efficient algorithms. It is a folklore in the database community that computation time of algorithms for manipulating a huge databases should be no more than $O(n^2)$ and preferably less than or equal to $O(n \log^k n)$, where n denotes the size of the database, and k is a constant.

In this paper, we propose a functional approach to address this kinds of problems in a general way. By making use of the nice features of functional languages, we can specify the problems in a very concise way, abstract over program structures using higher order functions, improve and reason about the programs, and provide a constructive and practical implementation of the algorithms. The main contributions of this paper are three folds.

- First, we propose an *unified* and *general* solution to optimized range problems, as summarized in our theorem (Section 5) showing that a wide class of optimized range problems, including the optimized gain range problem and the optimized support range problem as intensively studied in [FMMT96], can be solved efficiently.

- Second, we show that efficient algorithms for solving the optimized range problems can be *constructively* built according to the structure of predicates used to specify range conditions. This constructive approach is in sharp contrast with the existing case-by-case study as in [Zan92, Jeu93].
- Last but not least, our work shows that functional programming is not only efficient in specifying problems and reasoning about programs, but also good for practical implementation.

The rest of this paper is organized as follows. Section 2 explains some notation conventions and basic concepts used in this paper, and gives the problem specification informally. Section 3 addresses a class of optimized range problems with base predicates efficiently. On the basis of the solution in Section 3, Section 4 presents a general form solution, and gives its overall analysis. Section 5 shows how to constructively build solution for complicated optimized range problems with composite predicates from the base cases, and at last proposes a theorem for mining a wide class of optimized range. Section 6 presents an example to show how to use our solution to solve practical problems. Section 7 concludes this paper and highlights some future work.

2 OPTIMIZED RANGE PROBLEMS

Before formally addressing our general solution to the optimized range problems, we shall give a more formal and more precise definition of our problems. The notation we will use in this paper is Haskell [Bir98], a purely functional language. For those who are not familiar with Haskell, we briefly explain some basic conventions for functions below.

Functional application is denoted by a space and the argument 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 $(f\,a)\,b$. Functional application is regarded as more binding than any other operator, so $f\,a \oplus b$ means $(f\,a) \oplus b$, but not $f\,(a \oplus b)$. Functional composition is denoted by a centralized circle $\circ$, by definition, $(f \circ g)\,a = f\,(g\,a)$. Functional composition is an associative operator, and the identity function is denoted by *id*. 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 $a \oplus b = (a\oplus)\,b = (\oplus\,b)\,a = (\oplus)\,a\,b$.

2.1 Specification

The optimized range problem is essentially not a new problem at all. As a matter of fact, it is nothing but the longest segment problem [Zan92, Jeu93] known in program calculation community. It can be concisely specified by

$$orp\ p = maximum \circ map\ length \circ filter\ p \circ segs$$

which first enumerates all segments (continuous sublists) of the input item list, then keeps those satisfying the predicate p and computes their length, and finally return the largest value as the result. Here *segs*, a function for enumerating all segments, can be straightforwardly defined as follows.

$$\begin{array}{lcl} segs\ [\,] & = & [\,] \\ segs\ (a:x) & = & inits\ (a:x) +\!\!+\ segs\ x \\ \\ inits\ [\,] & = & [[\,]] \\ inits\ (a:x) & = & [\,] : map\ (a\,:)\ (inits\ x) \end{array}$$

Two remarks are worth making. First, this straightforward solution serves only as the specification rather than a real solution, because it is too inefficient to be used for computing on huge databases in practice. The purpose of this paper is therefore to provide an efficient solution to a class of these problems. Second, just to simplify our presentation, rather than returning the longest segment as the result, we just return the length of the longest segment. It should be direct to adapt our solution which returns the longest segment.

2.2 Examples

Recall the two problems in the introduction. Suppose that the database is sorted by date and each record in the database records items bought at each transaction:

```
[ (1985/1/1,  {Milk,Bread,Egg,...}),
  (1985/1/1,  {Paper,Milk,Sugar,Pencil,...}),
  (1985/1/2,  {}),
  ...,
  (2001/3/1,  {Milk,Bread,...})
]
```

The optimized support range problem for finding longest date period can be easily coded as following with three passes:

$$orsp = orp\ supportable \circ grp \circ mb.$$

In detail, first, we use the function *filterMilkBread* defined by

$$\begin{array}{lcl} mb\ x & = & [\,\text{if}\ Milk, Bread \subset is\ \text{then}\ (d,1)\ \text{else}\ (d,0) \\ & & \mid (d, is) \leftarrow x] \end{array}$$

to set the record if it contains Milk and Bread, and get

```
[ (1985/1/1,  1),
  (1985/1/1,  0),
  (1985/1/2,  0),
  ...,
  (2001/3/1,  1)
]
```

Then, we use the function *sumEachDay* defined by

$$\begin{array}{l} grp\ [\,] = [\,] \\ grp\ ((d,s):x) = grp'\ x\ (d,s,1) \\ grp'\ [\,]\ (d',s',i) = s'/i \\ grp'\ ((d,s):x)\ (d',s',i) = \\ \quad\quad \text{if}\ d == d'\ \text{then}\ grp'\ x\ (d', s+s', i+1) \\ \quad\quad \text{else}\ (d', s'/i) : grp'\ x\ (d,s,1) \end{array}$$

to compute support for each day, and get

```
[ (1985/1/1,  0.5),
  (1985/1/2,  0.0),
  ...,
  (2001/3/1,  1.0)
]
```

Finally, we calculate the longest date range by simply calling *orp* as follows.

$$\begin{array}{l} orp\ supportable \\ \quad \text{where} \\ \qquad supportable\ x = average\ (map\ snd\ x) \geq \theta \end{array}$$

Clearly, this last step is very time consuming if implemented naively.

The second problem which deal with more complicated range condition can be simply specified by

$$\begin{array}{l} orsp' = orp\ p\ \circ\ grp\ \circ\ mb \\ \quad \text{where } p\ x = supportable\ x \wedge gainable\ x \\ \qquad\quad gainable\ x = sum\ (map\ snd\ x) \geq \theta'. \end{array}$$

In fact, given a set of data, by preprocessing them to a suitable form with certain order, we can then mine interesting longest range by defining various predicate p for *orp*. More examples can be seen in the rest of the paper.

3 BASE CASES

The examples given in Section 2 are practical in digging interesting knowledge from the databases. In reality, there are a considerable variety of mining range problems, some are simple, and some are very complicated. It's not practical to solve all the optimized range problems in one stroke, so this paper will concentrate on one class of interesting optimized range problems.

From specification of Section 2, we can see that an optimized range problem is equivalent to compute a longest continuous sub-range satisfying a certain property p. Different definition for predicate p gives different optimized range problem. So we start with studying the structure of predicates. After full study on the optimized ranges, and taking advantage of the result already achieved on program calculation, we come to address one class of predicates that will be explained in later this paper.

First of all, we start with base cases, which will not only be solved efficiently but also used as basis for expressing larger optimized range problems later.

Relation between two End Elements

One of the simplest relation is the order relation between only two end elements (leftmost element and rightmost element) of a segment, which is known as the leftmost at most rightmost problem [Zan92]. The leftmost at most rightmost problem can be described as finding the longest segment of which the leftmost element is less than or equal to the rightmost element. That is,

$$\begin{array}{lcl} lr & :: & Ord\ Item \Rightarrow [Item] \rightarrow [Item] \\ lr & = & orp\ p \\ & & \text{where } p\ x = lm\ x \leq rm\ x \end{array}$$

where *lm* and *rm* are functions for computing the leftmost and rightmost element respectively, which can be implemented by `head` and `tail` in Haskell, and p representing the predicate the result range should satisfy.

The leftmost at most rightmost problem only describes the order relation between the two end elements, actually, it can be generalized to describe more complicated relation between the two end elements, for example, we may want to apply some computation over the end elements such as the "sum of leftmost and rightmost is less than or equal to a given constant C", which can be described as follows.

$$\begin{array}{lcl} lr & = & orp\ p \\ & & \text{where } p\ x = lm\ x\ +\ rm\ x\ \leq\ C \end{array}$$

i.e.

$$p\ x = lm\ x\ \leq\ C\ -rm\ x$$

it can be generalized to the following form,

$$p\ x\ =\ f\,(lm\ x)\ \leq\ g\ (rm\ x)$$

here, f and g are functions that can be computed in constant time, in other words, the leftmost and rightmost elements can be computed in constant time. Therefore, we get the following lemma.

Lemma 1 The optimized range problem

$$\begin{array}{l} orp\ p \\ \text{where } p\ x\ =\ f\ (lm\ x)\ \mathcal{R}\ g\ (rm\ x) \end{array}$$

can be solved in linear time, if function f and g can be computed in constant time, and $\mathcal{R}$ is a transitive relation in which the complement $\mathcal{R}^{\mathcal{C}}$ is also transitive.

Proof sketch. Here, we will omit the redundant explanation for the algorithm, and just give the scheme of our solution in Haskell code, as follows.

$$\begin{array}{l} lr :: (a \rightarrow b) \rightarrow (a \rightarrow b) \rightarrow (b \rightarrow b \rightarrow Bool) \rightarrow [a] \rightarrow Int \\ lr\ f\ g\ \mathcal{R}\ [] = 0 \\ lr\ f\ g\ \mathcal{R}\ x = lr'\ f\ g\ \mathcal{R}\ x\ ms\ ws \\ \\ lr'\ f\ g\ \mathcal{R}\ []\ ms\ ws = length\ ws \\ lr'\ f\ g\ \mathcal{R}\ x'@(a:x)\ (ms +\!\!+ [m])\ (w:ws) = \\ \quad \text{if } g\ a\ \mathcal{R}^{\mathcal{C}} f\ m \\ \qquad \text{then } lr'\ f\ g\ \mathcal{R}\ x'\ ms\ ([m,w] +\!\!+ ws) \\ \qquad \text{else } lr'\ f\ g\ \mathcal{R}\ x\ (ms +\!\!+ [m,w'])\ (ws +\!\!+ [f\ a]) \\ \quad\ \text{where } w'\ =\ min\ (f\ w)\ m \end{array}$$

From the scheme, we can see that the time cost mainly depends on adding or deleting element to the head or tail of a list via operation $+\!\!+$. Because the operation only involves the two ends of the list, so we can adopt some trick structure, for example, [Oka98]' efficient implementation of the double-end queue by using a pair of lists, as the implementation structure for `ms` and `ws`, then we can guarantee that the insertion and deletion can be done in constant time. For each input list, it calls `lr'` recursively at most $2n$ times, where n is the length of the input list. Thus we get the conclusion that the solution is linear.

□

When function f, g are identity function, i.e.,

$$\begin{aligned} f &= id \\ g &= id \end{aligned}$$

the scheme of the algorithm is just like

$$\begin{aligned}
&lr :: [a] \rightarrow Int \\
&lr\ [] = 0 \\
&lr\ x = lr'\ x\ ms\ ws \\
\\
&lr' :: [a] \rightarrow [a] \rightarrow [a] \rightarrow Int \\
&lr'\ []\ ms\ ws = length\ ws \\
&lr'\ x'@(a:x)\ (ms +\!\!+ [m])\ (w:ws) = \text{if } a\ >\ m \\
&\qquad \text{then } lr'\ x'\ ms\ ([m,w] +\!\!+ ws) \\
&\qquad \text{else } lr'\ x\ (ms +\!\!+ [m,w'])\ (ws +\!\!+ [a]) \\
&\quad \text{where } w'\ =\ min\ w\ m
\end{aligned}$$

it is very similar to the solution given by [Zan92], which is claimed to be linear.

Relation on Computation of all Elements

Lemma 1 only considers relation between two end elements, and cannot cope with cases where the predicate describes computation over the whole range such as aggregate functions *sum* and *average*.

Fortunately, we can deal with this case on the basis of Lemma 1, and at the same time, we can extend Lemma 1 to comprehend the new case. The main point is to extend relation between two end elements to that on computation over whole segments from some starting element to the two end elements, which can be summarized as follows.

$$\begin{aligned}
&orp\ p \\
&p\ x\ =\ \forall z.\ f\ (foldl\ (\oplus)\ e_1)\ z\ \mathcal{R}\ g\ (foldl\ (\otimes)\ e_2)(z +\!\!+ x)
\end{aligned}$$

here, $\oplus$ and $\otimes$ represent functions of two arguments, and $foldl$ is informally defined as follows:

$$foldl\ (\oplus)\ e\ [x_1, x_2, .., x_n]\ =\ (...((e \oplus x_1)...) \oplus x_n$$

Therefore, for any point z in the range, we should apply $f\ (foldl\ (\oplus)\ e_1)$ and $g\ (foldl\ (\otimes)\ e_2)$ to it, in other words, for the range x, we should apply $foldl$ operations to every initial segment of range x, this can be done by the function *scan*, which is defined in the following way [Bir98]:

$$scan\ (\oplus)\ e\ [x_1, x_2, \ldots,] = [e, e \oplus x_1, (e \oplus x_1) \oplus x_2, \ldots,]$$

So, $p\ x\ =\ \forall z.\ f\ (foldl\ (\oplus)\ e_1)\ z\ \mathcal{R}\ g\ (foldl\ (\otimes)\ e_2)(z +\!\!+ x)$ can be solved by generating a list of pair, first of which is the result applied by $f\ (foldl\ (\oplus)\ e_1)$, and second of which is that by $g\ (foldl\ (\otimes)\ e_2)$.

For this purpose, we can define a new function `scan'` as follows.

$$\begin{aligned}
&scan'\ (\oplus, \otimes)\ (e_1, e_2)\ [] = [] \\
&scan'\ (\oplus, \otimes)\ (e_1, e_2)\ (x:xs) = (a, b) : xs' \\
&\qquad \text{where } a = e_1 \oplus x \\
&\qquad\qquad b = e_2 \otimes x \\
&\qquad\qquad xs' = scan'\ (\oplus, \otimes)\ (a, b)\ xs
\end{aligned}$$

In matter of this case, we have the following lemma hold.

Lemma 2 The optimized range problem

$$\begin{aligned}
&orp\ p\ x \\
&\text{where } p\ x\ =\ \forall z.\ f\ (foldl\ \oplus\ e_1)\ z\ \mathcal{R} \\
&\qquad\qquad g\ (foldl\ \otimes\ e_2)\ (z +\!\!+ x)
\end{aligned}$$

can be transformed to the following form by being preprocessed.

$$\begin{aligned}
&orp\ p'\ \circ\ (scan'\ (\oplus, \otimes)\ (e_1, e_2)) \\
&\text{where } p'\ x\ =\ fst\ (lm\ x)\ \mathcal{R}\ snd\ (rm\ x)
\end{aligned}$$

It's almost the similar with the form in Lemma 1. Therefore it can be solved in linear time.

Proof sketch In fact, we can further use fusion to merge the preprocessing into our algorithm, and there is no need to preprocess the range under consideration separately.

The main point is that there is extensible room in the scheme given in Lemma 1. For example, we can apply arithmetic operation to x and m on the premise that the operation can be computed in constant time, then we can enforce this operation every time we inspect an element from left to right. To do this, all we need to is to add several parameters in the scheme to memorize the previous computed result so far. The updated scheme is as follows.

$$\begin{aligned}
&lr\ f\ g\ \mathcal{R}\ (\oplus)\ (\otimes)\ c_1\ c_2\ [] = 0 \\
&lr\ f\ g\ \mathcal{R}\ (\oplus)\ (\otimes)\ c_1\ c_2\ x = lr'\ f\ g\ \mathcal{R}\ (\oplus)\ (\otimes)\ c_1\ c_2\ x\ ms\ ws \\
\\
&lr'\ f\ g\ \mathcal{R}\ (\oplus)\ (\otimes)\ c_1\ c_2\ []\ ms\ ws = length\ ws \\
&lr'\ f\ g\ \mathcal{R}\ (\oplus)\ (\otimes)\ c_1\ c_2\ x'@(a:x)\ (ms +\!\!+ [m])\ (w:ws) = \\
&\quad \text{if } c_2'\ \mathcal{R}^C\ c_1' \\
&\qquad \text{then } lr'\ f\ g\ \mathcal{R}\ (\oplus)\ (\otimes)\ c_1'\ c_2'\ x'\ ms\ ([m,w] +\!\!+ ws) \\
&\qquad \text{else } lr'\ f\ g\ \mathcal{R}\ (\oplus)\ (\otimes)\ c_1'\ c_2'\ x\ (ms +\!\!+ [m,w'])\ (ws +\!\!+ c_1') \\
&\quad \text{where } w'\ =\ min\ (f\ w \oplus c_1)\ m \\
&\qquad\quad c_1' = c_1 \oplus f\ a \\
&\qquad\quad c_2' = c_2 \otimes g\ a
\end{aligned}$$

With the help of four additional parameter, we can still guarantee the algorithm is online and linear.

□

With Lemma 2, we can address optimized range problems like:

$$\begin{aligned}
&orp\ p \\
&\text{where } p\ x\ =\ average\ x \geq C
\end{aligned}$$

where C is a constant. For it can be transformed to

$$\begin{aligned}
p\ x\ &=\ average\ x \geq C \\
&=\ sum\ x / length\ x \geq C \\
&=\ \forall z,\ sum\ x\ /\ length\ x \geq C \\
&=\ \forall z,\ (sum\ (z +\!\!+ x) - sum\ z)\ / \\
&\qquad\quad (length\ (z +\!\!+ x) - length\ z) \geq C \\
&=\ \forall z,\ sum\ (z +\!\!+ x) - sum\ z\ \geq \\
&\qquad\quad C \times length(z +\!\!+ x) - C \times length\ z \\
&=\ \forall z,\ sum\ z - C \times length\ z \leq \\
&\qquad\quad sum\ (z +\!\!+ x) - C \times length(z +\!\!+ x) \\
&=\ \forall z,\ f\ (foldl\ (\lambda\ x\ \lambda\ (s,l) \rightarrow\ (s+x, l+1), (0,0))\ z \leq \\
&\qquad g\ (foldl\ (\lambda\ x\ \lambda\ (s,l) \rightarrow\ (s+x, l+1), (0,0))\ (z +\!\!+ x)
\end{aligned}$$

with f and g defined as

$$\begin{aligned}
f\ (s,l)\ &=\ s\ -\ C\ \times\ l \\
g\ (s,l)\ &=\ s\ -\ C\ \times\ l
\end{aligned}$$

So far, we can efficiently solve optimized range problems whose predicates are in the following form:

$$
\begin{array}{lll}
p\ x & = & f\ (lm\ x)\ \mathcal{R}\ g\ (rm\ x) \\
 & | & \forall z.\ f\ (foldl\ (\oplus)\ e_1)\ z\ \mathcal{R}\ g\ (foldl\ (\otimes)\ e_2)(z +\!\!+ x)
\end{array}
$$

and we will call the two kinds of predicates *base predicates* later.

The base predicate makes the basis upon which we can construct more complicated and practical problems.

4 GENERAL FORM

From the solution for the base predicates, we can generalize the algorithm's scheme as follows.

$$
\begin{array}{l}
gen\ f\ g\ \mathcal{R}\ (\oplus)\ (\otimes)\ c_1\ c_2\ x\ = \\
\qquad gen'\ f\ g\ \mathcal{R}\ (\oplus)\ (\otimes)\ c_1\ c_2\ x\ acc\ win \\
\\
gen'\ f\ g\ \mathcal{R}\ (\oplus)\ (\otimes)\ c_1\ c_2\ []\ acc\ win\ =\ length\ win \\
gen'\ f\ g\ \mathcal{R}\ (\oplus)\ (\otimes)\ c_1\ c_2\ x'@(a:x)\ acc\ win\ = \\
\quad \text{if } accIsMin\ acc\ c_2' \\
\quad \text{then } gen'\ f\ g\ \mathcal{R}\ (\oplus)\ (\otimes)\ c_1'\ c_2'\ x' \\
\qquad (accDelete\ acc)\ (winExtend\ win\ e) \\
\quad \text{else } gen'\ f\ g\ \mathcal{R}\ (\oplus)\ (\otimes)\ c_1'\ c_2'\ x \\
\qquad (accInsert\ acc\ h)\ (winSlide\ win\ c_1') \\
\quad \text{where } \ c_1' = c_1\ \oplus\ f\ a \\
\qquad c_2' = c_2\ \otimes\ g\ a \\
\qquad h = winHdLeft\ win \\
\qquad e\ =\ accLastElement\ acc
\end{array}
$$

in which, *gen* is general function that compute the optimized range that satisfies the given condition. It is a generalized form to the scheme showed in Lemma 1 and Lemma 2. In Lemma 1 and Lemma 2, the auxiliary storage `ms` and `ws` are both lists, which work well for the base cases, but for the more general and complicated cases, we now give a more general definition for them, i.e., the types for `acc` and `win` are `Acc a` and `Win a`, respectively. And there are operations defined on `acc` and `win`.

The operations for `acc` are

$$
\begin{array}{lll}
accInsert & :: & Acc\ a \to a \to Acc\ a \\
accDelete & :: & Acc\ a \to a \to Acc\ a \\
accIsmin & :: & Acc\ a \to a \to Bool
\end{array}
$$

among which, `accInsert` inserts an element to `acc`, `accDelete` deletes an element from `acc`, and `accIsmin` determines whether the given element is minimal in `acc`. In the base cases, `accInsert` is just a catenation to `acc`, and `accDelete` is `init`, and `accIsmin` is equivalent to `last`.

And the operation for `win` include:

$$
\begin{array}{lll}
winExtend & :: & Win\ a\ \to\ a\ \to\ Win\ a \\
winSlide & :: & Win\ a\ \to\ a\ \to\ Win\ a
\end{array}
$$

among which, `winExtend` extends `win` by one element, and increases its length by one. `winSlide` shifts `win` by one element and keep the length unchanged.

The functions used in this general scheme can be expressed in a simple way like

$$
\begin{array}{l}
orp_g\ ((f,(\oplus),e_1),\ r,\ (g,(\otimes),e_2))\ :: \\
\qquad Acc\ a\ \to\ Win\ a\ \to\ [a]\ \to\ Int
\end{array}
$$

Its definition is as follows. For simplicity, we will use orp_g stands for the whole $orp_g\ ((f,(\oplus),e_1),r,(g,(\otimes),e_2))$.

$$
\begin{array}{l}
orp_g\ acc\ win\ []\ =\ length\ win \\
orp_g\ acc\ win\ x'@(a:x)\ = \\
\quad \text{if } accIsMin\ acc\ c_2' \\
\quad \text{then } orp_g\ acc\ win\ \ x' \\
\qquad (accDelete\ acc)(winExtend\ win\ e) \\
\quad \text{else } orp_g\ acc\ win\ x\ (accInsert\ acc\ h) \\
\qquad (winSlide\ win\ c_1') \\
\quad \text{where } \ c_1' = c_1\ \oplus\ f\ a \\
\qquad c_2' = c_2\ \otimes\ g\ a \\
\qquad h = winHdLeft\ win \\
\qquad e\ =\ accLastElement\ acc
\end{array}
$$

As a special case, the optimized range problem

$$
\begin{array}{l}
orp\ p \\
\text{where } p\ x\ =\ f\ (lm\ x)\ \mathcal{R}\ g\ (lm\ x)
\end{array}
$$

can be solved by

$$
\begin{array}{l}
orp\ p\ =\ orp_g\ ((f,(\oplus),e_1),r,(g,(\otimes),e_2)) \\
orp_g\ ((f,(\oplus),e_1),r,(g,(\otimes),e_2))\ ::\ [a]\ \to\ [a]\ \to\ [a]\ \to\ Int \\
\quad \text{where } \oplus\ a\ b\ =\ b \\
\qquad \otimes\ a\ b\ =\ b \\
\qquad r\ =\ \mathcal{R}
\end{array}
$$

and the optimized range problem

$$
\begin{array}{l}
orp\ p \\
\text{where } p\ x\ =\ \forall z.\ f\ (foldl\ \oplus\ e_1)\ z\ \mathcal{R} \\
\qquad g\ (foldl\ \otimes\ e_2)\ (z +\!\!+ x)
\end{array}
$$

can be solved by

$$
\begin{array}{l}
orp\ p\ =\ orp_g\ ((f,(\oplus),e_1),r,(g,(\otimes),e_2)) \\
orp_g\ ((f,(\oplus),e_1),r,(g,(\otimes),e_2))\ ::\ [a]\ \to\ [a]\ \to\ [a]\ \to\ Int \\
\quad \text{where } f\ =\ fst \\
\qquad g\ =\ snd \\
\qquad r\ =\ \mathcal{R}
\end{array}
$$

The characteristics of the algorithm are reflected in three folds.

- One is that it is on-line. An algorithm computing $fun\ z$ is called on-line if in the range z for each $x \in inits(z)$, the result $fun\ x$ is available before the elements in the right of x have been inspected, in other words, the on-line algorithm inspects the input elements from left to right, and each element is only processed once.
- The other is that in the process of scanning the input list, it uses two auxiliary storage, the accumulation `acc` to store the intermediate data, the window `win` to store result achieved so far.
- Moreover, in the algorithm, when we process each element x, we can enforce some operations on it, such as applying functions f, g, doing arithmetic operations $\oplus$, $\otimes$ with c_1, c_2.

From the scheme of the algorithm, we can see that the time cost of this algorithm mainly depends on the following operations.

- Apply function f and g on element x, assume its cost is t_f.
- Arithmetic operation $\oplus$ and $\otimes$ with c_1 and c_2, respectively, assume its cost is $t_\oplus$.
- Operations on the accumulation parameter `acc`, including:

  accDelete: delete the element from `acc`, assume its time cost is t_{del}.

  accInsert: insert an element to `acc`, time cost is t_{ins}.

  accIsmin: to determine whether the new element is minimal in the `acc` , time is t_{min}.
- Operations on window `win`, including:

  winSlide: shift `win` to right by one element, assume its time cost is t_{slide}.

  winExtend: extend `win` to left by one element, increase its length by one, its time is t_{ext}.

Therefore, we get the conclusion that the time cost of the algorithm is $O(max(t_f, t_\oplus, t_{ins}, t_{del}, t_{min}, t_{ext}, t_{slide})\ n)$.

For the base predicates, the `acc` and `win` are implemented using list, so the operation on them only involves the two ends of the list, we can adopt some trick, such as, [Oka98]'s efficient implementation of the double-end queue by using a pair of lists, to guarantee the insertion and deletion on them can be done in constant time, in other words, $t_{ins}, t_{del}, t_{ext}, t_{slide}$ are $O(1)$. when building the `acc` from scratch, each time we insert a new element in it, we can arrange the minimum one in the head of the `acc`, so we can just compare the first element with the new element to determine whether it is minimal, then t_{min} is also $O(1)$. Thus for the optimized range whose predicate is base predicate, the time cost is only depends on $t_f, t_\oplus$.

Up to now, the function f and g and $\oplus, \otimes$ used in base cases are all be computed in constant time. So we have the following conclusion.

The time cost for base cases is

$$\mathcal{T}_{base} \ = \ O(n).$$

5 COMPOSITE CASES

In practice, we usually want to find the optimized ranges that satisfy not only a single predicate, but also a combination of a number of predicates with connective operators, namely

$$\begin{array}{lcl} p\ x & = & not\ (p_1\ x) \\ & | & p_1\ x \vee p_2\ x \\ & | & p_1\ x \wedge p_2\ x \end{array}$$

Speaking intuitively, *not* computes the optimized range that satisfies the complement of predicate p.

The $\vee$ computes the optimized range that satisfies either predicate p_1 or predicate p_2.

And, the $\wedge$ computes the optimized range that satisfies both predicate p_1 and predicate p_2.

In the matter of composite predicate, we have the following Lemma hold.

Lemma 3 Any composite predicate can be expressed as the disjunction of a number of conjunction of base predicates, i.e.,

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

where, p_{ij} is base predicate.

Proof sketch. First, let's look at the case

$$not\ p\ x$$

where p is base predicate,

- If $p\ x\ =\ f\ (lm\ x)\ \mathcal{R}\ g\ (rm\ x)$, then

  $not\ p\ x\ =\ f\ (lm\ x)\ \mathcal{R}^C\ g\ (rm\ x)$, because $\mathcal{R}$ and $\mathcal{R}^C$ are all transitive, So *not* p is also base predicate.
- If $p\ x\ =\ f\ (foldl\ \oplus\ e_1)\ z\ \ \mathcal{R}\ g\ (foldl\ \otimes\ e_2)\ (z +\!\!+ x)$, then $not\ p\ x\ =\ f\ (foldl \oplus e_1)\ \mathcal{R}^C\ )\ g\ (foldl \otimes e_2)\ (z +\!\!+ x)$, because $\mathcal{R}$ and $\mathcal{R}^C$ are all transitive, So *not* p is also base predicate.

therefore, we get the conclusion that if p is a base predicate, then *not* p is also a predicate.

By A. de Morgan's law, and distributive and associative law on the logic operators, the other part is self-evident.

□

Taking advantage of the solution for base predicate, we now show how to constructively build the solution for composite predicates.

With Lemma 3, we only need to address two cases, one is that for $\vee$, another is for $\wedge$.

Connective $\vee$

As for the cases where the range conditions consist of a number of connective $\vee$, i.e., the predicate in the form defined by

$$p\ x\ =\ p_1\ x \vee p_2\ x \vee ... \vee p_k\ x$$

where, k is the number of base predicates.

First, let's start from the choice between two predicates, which can be solved intuitively as follows:

$$orp_{or}\ p_1\ p_2\ x\ \ =\ max\ (orp\ p_1\ x)\ (orp\ p_2\ x)$$

On the analogy of two predicates, we can derive the solution for the case where the number of predicates is k, which can be achieved recursively.

$$\begin{array}{l} orp_{or}\ ::\ Int\ \rightarrow\ [Bool]\ \rightarrow\ [a]\ \rightarrow\ Int \\ orp_{or}\ k\ (p : ps)\ x\ =\ if\ k\ >\ 1 \\ \qquad then\ max\ (orp\ p\ x)\ (orp_{or}\ (k-1)\ ps\ x) \\ \qquad else\ orp\ p\ x \end{array}$$

From Section 4, we know that the solution for single predicate is within $O(n)$ time, i.e, the time for `orp p` is linear; and from the program above, the connective logical $\vee$ can be solved in linear time, if all the predicates within it are base ones.

Connective $\wedge$

This case is related with the optimized range whose predicate is in the form defined by

$$p\ x \ = \ p_1\ x \wedge p_2\ x \wedge ... \wedge p_k\ x$$

where, k is the number of base predicates.

In this case, we want to compute `orp` that satisfies all of these predicates, in other words, if the $p_i\ x$ is expressed by $f_i\ (lm\ x)\ \ \mathcal{R}_i\ \ g_i\ (rm\ x)$, then it means that `orp` has to satisfy

$$(f_1, ..., f_k)\ (lm\ x)\ \ (\mathcal{R}_1, ..., \ \mathcal{R}_k)\ \ (g_1, ..., g_k)\ (rm\ x)$$

where

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

So when we inspect the element x, we have to check whether all the predicates are satisfied.

Without loss of generality, we will use the $p_1\ \wedge\ p_2$ in which p_1 and p_2 are base predicates as our example.

$$\begin{array}{l} orp\ p \\ \quad \text{where } p\ x \ = \ p_1\ x \wedge\ p_2\ x \\ \qquad\qquad p_1\ x \ = \ f_1\ (lm\ x)\ \leq\ g_1\ (rm\ x) \\ \qquad\qquad p_2\ x \ = \ f_2\ (lm\ x)\ \leq\ g_2\ (rm\ x)\end{array}$$

We know from Section 4, $orp\ p_1$ and $orp\ p_2$ can be solved linearly, with

$$orp\ p_1 \ = \ orp_g\ (f_1, \oplus_1, e_{11}), r_1, (g_1, \otimes_1, e_{21})$$

$$orp\ p_2 \ = \ orp_g\ (f_2, \oplus_2, e_{12}), r_2, (g_2, \otimes_2, e_{22}))$$

respectively.

For the case $p_1\ x\ \wedge\ p_2\ x$, we can combine $orp\ p_1$ with $orp\ p_2$ together, i.e., combine $orp_g\ (f_1, \oplus_1, e_{11}), r_1, (g_1, \otimes_1, e_{21})$ with $orp_g\ (f_2, \oplus_2, e_{12}), r_2, (g_2, \otimes_2, e_{22}))$ to

$$\begin{array}{l} orp_g\ (f', \oplus', e_1{'}), r', (g', \otimes', e_2{'})) \\ \quad \text{where } f' \ = \ (f_1,\ f_2) \\ \qquad\qquad g' \ = \ (g_1\ ,\ g_2\ ) \\ \qquad\qquad \oplus' \ = \ (\oplus_1,\ \oplus_2) \\ \qquad\qquad \otimes' \ = \ (\otimes_1,\ \otimes_2) \\ \qquad\qquad e_1' \ = \ (e_{11},\ e_{12}) \\ \qquad\qquad e_2' \ = \ (e_{21},\ e_{22}) \\ \qquad\qquad r' \ = \ (r_1, r_2)\end{array}$$

Using the same way to process the optimized range problem whose predicate is a conjunction of k base predicates

$$\begin{array}{l} orp\ p \\ \quad \text{where } p\ x \ = \ p_1\ x \wedge\ p_2\ x\ ...\ \wedge\ p_k\ x\end{array}$$

can be combined as follows.

$$\begin{array}{l} orp_g\ (f', \oplus', e_1{'}), r', (g', \otimes', e_2{'})) \\ \quad \text{where } f' \ = \ (f_1,\ f_2, ..., f_k) \\ \qquad\qquad g' \ = \ (g_1,\ g_2, ...,\ g_k) \\ \qquad\qquad \oplus' \ = \ (\oplus_1,\ \oplus_2, ...,\ \oplus_k) \\ \qquad\qquad \otimes' \ = \ (\otimes_1,\ \otimes_2, ...,\ \otimes_k) \\ \qquad\qquad e_1' \ = \ (e_{11},\ e_{12}, ...,\ e_{1k}) \\ \qquad\qquad e_2' \ = \ (e_{21},\ e_{22}, ...,\ e_{2k}) \\ \qquad\qquad r' \ = \ (r_1, r_2, ..., r_k)\end{array}$$

One difference between this $\wedge$ case with that in base predicate is that the order, for the latter, the order is complete, you can directly tell the result about comparison between two elements, and therefore, for example, for $\leq$ case, we always have the minimum element in the process of inspecting the input list; and for the former, the order is no longer monotonous, it now becomes to the comparison between two k-tuples, so it can not be guaranteed that every time when we inspect an element, we can get the minimum.

To address it, we should first define the order for the k-tuple. For simplicity and without loss of generality, here we use 2-tuples, i.e., pair as the example.

$$\begin{array}{l} instance\ (Eq\ a, Eq\ b)\ \Rightarrow\ Eq\ (P\ a\ b)\ where \\ \quad P\ x_1\ y_1 == P\ x_2\ y_2\ = \\ \qquad\qquad x_1 == x_2\ \wedge\ y_1 == y_2 \end{array}$$

$$\begin{array}{l} instance\ (Ord\ a, Ord\ b)\ \Rightarrow\ Ord\ (P\ a\ b)\ where \\ \quad Compare\ (P\ x_1\ y_1)\ (P\ x_2\ y_2) \\ \qquad |\ x_1 == x_2\ \wedge\ y_1 == y_2 =\ EQ \\ \qquad |\ x_1 \leq x_2\ \wedge\ y_1 < y_2\ \vee \\ \qquad\qquad x_1 < x_2\ \wedge\ y_1 \leq y_2\ =\ LT \\ \qquad |\ otherwise\ =\ GT \end{array}$$

The next thing we have to is how to memorize the intermediate result. In the base case, as we explained in Section 4, we use two-end queue as `acc`. Now the monotonous queue works no longer well for the $\wedge$ case, it is necessary to use a data structure that can deal with tuples efficiently. So after study on the structure of the composite predicates, we adopt a so-called a K-D tree [ZYHT01] as the accumulation parameter `acc`, and the main operations on the accumulation parameter includes the following functions:

$$\begin{array}{lll} accInsert & :: & KDtree\ a \to a \to KDtree\ a \\ accDelete & :: & KDtree\ a \to a \to KDtree\ a \\ accIsmin & :: & KDtree\ a \to a \to Bool \end{array}$$

whose functions are same with the explanation in Section 4.

The window `win` is same with that in base case, so the operations on `win` is still $O(1)$.

From the conclusion in Section 4, we can get that the time cost for this algorithm depends on that for operations `accDelete`, `accInsert` and `accIsmin`. Suppose that the time cost for `accDelete`, `accInsert`, `accIsmin` are t_1, t_2 and t_3, respectively. According to [ZYHT01], the insertion and deletion to K-D tree can be implemented in $O(\log\ n)$ time, and the search can be completed in $O(\log^{(k-1)} n)$ time, so we can get that t_1 and t_2 are $O(\log\ n)$, and t_3 is $O(\log^{(k-1)}\ n)$. Therefore we can have the following conclusion.

The time cost for k-conjunction case is

$$\mathcal{T}_{k\wedge} \ = \ O(n\ \log^{(k-1)} n).$$

Connective $\vee$ and $\wedge$

In this case, the predicate takes the form as follows.

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

We can solve this case on the basis of that for connective $\wedge$ and connective $\vee$. First computing every conjunction included in the predicate in the same way for the connective

$\wedge$, and then using the same code as that for connective $\vee$. Because the time cost for conjunction of k_i base predicates is $O(n \log^{(k_i-1)} n)$, so the whole time cost is $O(n \log^{(k-1)} n)$, where k is the maximum one in $(k_1, k_2, ..., k_m)$. So we have following conclusion.

The time cost for optimized range problem

$$
\begin{array}{lll}
\multicolumn{3}{l}{orp\ p} \\
\text{where } p\ x & = & p_{11}\ x \wedge p_{12}\ x \wedge ... \wedge p_{1k_1}\ x \quad \vee \\
 & & p_{21}\ x \wedge p_{22}\ x \wedge ... \wedge p_{2k_2}\ x \quad \vee \\
 & & ...... \quad \vee \\
 & & p_{m1}\ x \wedge p_{m2}\ x \wedge ... \wedge p_{mk_m}\ x
\end{array}
$$

is

$$\mathcal{T}_{\wedge\vee} = O(n\ \log^{(k-1)} n)$$

where k is the maximum of $(k_1, k_2, ..., k_m)$.

Using these conclusions in this section and in last section, we can infer the following theorem.

Theorem 4 (Theorem) Any k-dimensional optimized range problem *orp p* where its predicate p is in the following form,

$$
\begin{array}{lll}
\multicolumn{3}{l}{orp\ p} \\
\text{where } p\ x & = & f\ (lm\ x)\ \mathcal{R}\ g\ (rm\ x) \\
 & | & \forall z.\ f\ (foldl\ \oplus\ e_1)\ z \quad \mathcal{R} \\
 & & \qquad g\ (foldl\ \otimes\ e_2)\ (z +\!\!+ x) \\
 & | & p_{11}\ x \wedge p_{12}\ x \wedge ... \wedge p_{1k_1}\ x \quad \vee \\
 & & p_{21}\ x \wedge p_{22}\ x \wedge ... \wedge p_{2k_2}\ x \quad \vee \\
 & & ...... \quad \vee \\
 & & p_{m1}\ x \wedge p_{m2}\ x \wedge ... \wedge p_{mk_m}\ x
\end{array}
$$

can be solved on-linely at the cost of $O(n \log^{(k-1)} n)$ time and $O(n)$ space , where k is the maximum one among k_1, k_2, $\ldots$, k_m.

□

Specifically, the solution holds the following property. When $k = 1$, i.e., for the single base predicate, the solution is linear; while $k = 2$, i.e., for the conjunction with two base predicates, the time cost is $O(n \log n)$; and for $k = 3$, the time cost is just $O(n \log^2 n)$.

6 EXAMPLE

Using the Theorem 4, we can mining a wide class of optimized range problems, including the problems given in Section 1.

Recall the second problem in Section 1, which is expressed in Section 2 as follows.

$$
\begin{array}{l}
orsp' = orp\ p\ \circ\ grp\ \circ\ mb \\
\quad \text{where } p\ x = supportable\ x \wedge gainable\ x \\
\qquad supportable\ x = average\ (map\ snd\ x) \geq \theta \\
\qquad gainable\ x = sum\ (map\ snd\ x) \geq \theta'.
\end{array}
$$

in which, *orp supportable* and *orp gainable* are base cases, and can be formalized in the general forms

$$
\begin{array}{l}
orp\ supportable\ =\ orp_g\ (f_1, \oplus_1, e_{11}), r_1, (g_1, \otimes_1, e_{21}) \\
\quad \text{where } f_1\ s\ l\ =\ s\ -\ \theta \times\ l \\
\qquad g_1\ s\ l\ =\ s\ -\ \theta \times\ l \\
\qquad \oplus_1\ =\ \lambda x\ \lambda(s,l) \rightarrow (s+x, l+1) \\
\qquad \otimes_1\ =\ \lambda x\ \lambda(s,l) \rightarrow (s+x, l+1) \\
\qquad e_{11}\ =\ (0,0) \\
\qquad e_{21}\ =\ (0,0) \\
\qquad r_1\ =\ \leq
\end{array}
$$

and

$$
\begin{array}{l}
orp\ gainable\ =\ orp_g\ (f_2, \oplus, e_{12}), r_2, (g_2, \otimes_2, e_{22}) \\
\quad \text{where } f_2\ =\ fst \\
\qquad g_2\ =\ snd \\
\qquad \oplus_2\ =\ + \\
\qquad \otimes_2\ =\ + \\
\qquad e_{12}\ =\ 0 \\
\qquad e_{22}\ =\ 0 \\
\qquad r_2\ =\ \leq
\end{array}
$$

respectively. According to the solution for conjunction case in Section 5, we can merge these two into

$$
\begin{array}{l}
orsp'\ supportable\text{-}gainable \\
orp_g\ (f', \oplus', e_1'), r', (g', \otimes', e_2')) \\
\quad where\ f'\ =\ (f_1,\ f_2) \\
\qquad g'\ =\ (g_1,\ g_2) \\
\qquad \oplus'\ =\ (\oplus_1,\ \oplus_2) \\
\qquad \otimes'\ =\ (\otimes_1,\ \otimes_2) \\
\qquad e_1'\ =\ (e_{11},\ e_{12}) \\
\qquad e_2'\ =\ (e_{21},\ e_{22}) \\
\qquad r'\ =\ (\leq, \leq)
\end{array}
$$

where the definition for f_1, f_2,...etc. is same with that in *orp supportable* and in *orp gainable*, respectively.

According the Theorem given in Section 5, the time cost for it is $O(n\ \log\ n)$. So we can see that our solution is efficient for this example, which cannot be solved in [FMMT96].

7 CONCLUSION

In this paper, we identify one important class of data mining problems called optimized range problems, and propose a general theorem 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 shows that functional programming is not only adept at the specifying and deriving the problem, but also good at implementing the solution.

We have already finished the algorithm design, and we are now working on implementing a domain specific query language for mining this class of optimized ranges from huge databases on the basis of our main theorem.

At present, we use a so-called *k-d tree* as our accumulation parameter, it's a variant of the Bantley's k-d tree, it's efficient to deal with the multidimensional predicate, however, it's difficult to maintain its balance, so another interesting future work is to how to systematically rebalance Bantley's k-d tree.

References

[AIS93] Rakesh Agrawal, Tomasz Imielinski, Arun Swami. Mining Association Rules between Sets of Items in Large Databases. In *Proceedings of the 1993 ACM SIGMOD Conference (SIGMOD'93)*, pages 207-216, May 1993.

[BdM96] R.S. Bird and O. de Moor. *Algebras of Programming*. Prentice Hall, 1996.

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

[Bir98] R.S. Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, 1998.

[BRS00] Sergey Brin, Rajeev Rastogi, and Kyuseok Shim. Mining Optimized Gain Rules for Numeric Attributes. Not clear.

[CHY96] M.S.Chen, J. Han, and P.S.Yu. Data Mining: An Overview from a Database Perspective. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):866-883, 1996.

[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.

[FMMT96b] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Data Mining Using Two-Dimensional Optimized Association Rules: Scheme, Algorithms, and Visualization In *Proc. ACM PODS'96*, pages 13–23, Montreal Quebec, Canada, 1996.

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

[FSSU96] 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.

[Hut99] Graham Hutton. A tutorial on the universality and expressiveness of fold. *J. Functional Programming*, 9(4)355-372, July 1999.

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

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

[SSU95] A. Silberschatz, M.Stonebraker,and J.D.Ullman. Database reserach: Achievements and opportunities into the 21st century. *In Report of an NSF Workshop o the Future of Database Systems Research*, May 1995.

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

[ZYHT01] Haiyan Zhao, Tetsuo Yokoyama, Zhenjiang Hu, Masoto Takeich. Multiple Dimensional Tree with Minimum Attribute. to appear, *Technical Report, Department of Mathematical Engineering, University of Tokyo, 2001.*