. . . .

Dr. Nysret Musliu

TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

M.Sc. Arbeit

Data Mining on Empty Result Queries

ausgeführt am

Institut für Informationssysteme
Abteilung für Datenbanken und Artificial Intelligence
der Technischen Universität Wien

unter der Anleitung von

Priv.Doz. Dr.techn. Nysret Musliu
und Dr.rer.nat Fang Wei

durch

Lee Mei Sin

Wien, 9. Mai 2008

. . . .

Lee Mei Sin

. . . .

Dr. Nysret Musliu

TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

MASTER THESIS

Data Mining on Empty Result Queries

carried out at the

Institute of Information Systems
Database and Artificial Intelligence Group
of the Vienna University of Technology

under the instruction of

Priv.Doz. Dr.techn. Nysret Musliu
and Dr.rer.nat Fang Wei

by

Lee Mei Sin

Vienna, May 9, 2008

. . . .

Lee Mei Sin

Abstract

A database query could return an empty result. According to statistics, empty results are frequently encountered in query processing. This situation happens when the user is new to the database and has no knowledge about the data. Accordingly, one wishes to detect such a query from the beginning in the DBMS, before any real query evaluation is executed. This will not only provide a quick answer, but it also reduces the load on a busy DBMS. Many data mining approaches deal with mining high density regions (eg: discovering cluster), or frequent data values. A complimentary approach is presented here, in which we search for empty regions or holes in the data. More specifically, we are mining for combination of values or range of values that do not appear together, resulting in empty result queries. We focus our attention on mining not just simple two dimensional subspace, but also in multi-dimensional space. We are able to mine heterogeneous data values, including combinations of discrete and continuous values. Our goal is to find the maximal empty hyper-rectangle. Our method mines query selection criteria that returns empty results, without using any prior domain knowledge.

Mined results can be used in a few potential applications in query processing. In the first application, queries that has selection criteria that matches the mined rules will surely be empty, returning an empty result. These queries are not processed to save execution. In the second application, these mined rules can be used in query optimization. It can also be used in detecting anomalies in query update. We study the experimental results obtained by applying our algorithm to both synthetic and real life datasets. Finally, with the mined rules, an application of how to use the rules to detect empty result queries is proposed.

Category and Subject Descriptors:

General terms:

Data Mining, Query, Database, Empty Result Queries

Additional Keywords and Phrases:

Holes, empty combinations, empty regions

Contents

List of Figures

List of Tables

We are drowning in information,
but starving for knowledge.
— John Naisbett

A special dedication to everyone who has helped make this a reality.

Acknowledgments

I would like to express my heartfelt appreciation to my supervisor, Dr. Fang Wei, for the support, advice and assistance. I am truly grateful to her for all the time and effort he invested in this work. I would also like to thank Dr. Nysret Musliu for being the official supervisor and for all the help he has provided.

My Master's studies were funded by the generous Erasmus Mundus Scholarship, which I was awarded as a member of European Master's Program in Computational Logic. I would like to thank the Head of the Program, Prof. Steffen Holldobler, for giving me this opportunity, and the Program coordinator in Vienna, Prof. Alexander Leitsch for kind support in all academic and organizational issues.

Finally, I am grateful to all my friends and colleagues who have made these last two years abroad a truly enjoyable stay and an enriching experience for me.

1 Introduction

Empty result queries are frequently encountered in query processing. Mining empty result queries can be seen as mining empty regions in the dataset. This also can be seen as a complimentary approach to existing mining strategies. Much work in data mining are focused in finding dense regions, characterizing the similarity between values, correlation between values and etc. It is therefore a challenge to mine empty regions in the dataset. In contrast to data mining, the following are interesting to us: outliers, sparse/negative clusters, holes and infrequent items. In a way, determining empty regions in the data can be seen as an alternative way of characterizing data.

It is observed that in [Gryz and Liang, 2006], a join of relations in real databases is usually much smaller than their Cartesian product. Empty region exist within the table itself, this is evident when ploting a graph between two attributes in a table reveals a lot of empty regions. Therefore it is reasonable to say that empty regions exist, as it is impossible to have a table or join of tables that are completely packed. In general, high dimensional data and attributes with large domains are bound to have an extremely high number of empty and low density regions.

Empty regions formed by categorical values means that certain combination of values are not possible. As for continuous values, this indicates that certain ranges of attributes never appear together. Semantically, the empty region implies the negative correlation between domains or attributes. Hence, these regions have been exploited in semantic query optimization in [Cheng et al., 1999].

Consider the following example:

Flight Insurance					
FID	CID	Travel Date	Airline	Destination	Full Insurance
119	XDS003	1/1/2007	SkyEurope	Madrid	Y
249	SDS039	2/10/2006	AirBerlin	Berlin	N
...					

Customer			
CID	Name	DOB	...
XDS003	Smith	22/3/1978	...
SDS039	Murray	1/2/1992	...
...			

Flight Company		
Airline	Year of Establishment	...
SkyEurope	2001	...
AirBerlin	1978	...
...		

Table 1.1: Schema and Data of a Flight Insurance database

In the above data set, we would like to discover if there are certain ranges of the attributes that never appear together. For example, it may be the case that no flight insurance for SkyEurope flight before 2001, SkyEurope does not fly to destinations other than European countries or Full Insurance are not issues to customers younger than 18. Some of these empty regions may be foreseeable and logical, for example, SkyEurope was established in 2001. Others may have more complex and uncertain causes.

1.1 Background

Empty result queries are queries sent to a RDBMS but after being evaluated, return the empty result. Before execution, if a RDBMS is unable to detect an empty result queries, it will execute the query and thus waste execution time and resources. Queries that join two or more tables will take up a lot of processing time and resources even if the result is empty, because time and resoures has to be allocated for doing the join. According to [Luo, 2006], empty results are frequently encountered in query processing. eg: in a query trace that contains 18,793 SQL queries and is collected in a Customer Relationship Management (CRM) database application developed by IBM, 18.07% (3,396) queries are empty-result ones.

The empty-result problem has been studied in the research literature. It is known as *empty-result problem* in [Kießling and Köstler, 2002]. According to [Luo, 2006], existing solutions fall into two category:

1. Explain what leads to the empty result set
2. Automatically generalize the query so that the generalized query will return some answers

Besides being used in the form of query processing, empty results can also be characterized in terms of data mining. Empty result happens when ranges of attributes do not appear together [Gryz and Liang, 2006] or the combination of attribute values do not exist in a tuple of a database. Empty results can be seen as a negative correlation between two attribute values. Empty regions can be thought of as an alternative characteristic of data skew. We can use the mined results to provide another description of data distribution in a universal relation.

The causes that lead to an empty result query are:

1. Values do not appear in the database
   This refers to values that do not appear in the domain of the attributes.

2. Combination or certain ranges of values do not appear together
   These are individual values that appear in the database, however do not appear together with other attribute values.

1.2 Motivation

By identifying empty result queries, DBMS can avoid executing them, thus reducing unfavorable delay and reducing the load on the DBMS, and thus further improving system performance. The results can help facilitate the exploration of massive datasets.

Mining empty hyper-rectangles can be used in the following cases:

1. Detection of empty-result queries.
   One might think that empty-result queries can finish in a short amount of time. However, this is often not the case. For example, consider a query that joins two relations. Regardless of whether the query result set is empty, the query execution time will be longer than the time required to do the join. This can cause unnecessary load on a heavily loaded RDBMS. In general, it is desirable to quickly detect empty-result queries. Not only does it facilitate the exploration of massive dataset but also it provides important benefits to users. First, users do need to wait for queries to be processed, but in the end, turned out to be empty. Second, by avoiding the unnecessary execution of empty-result queries, the load on the RDBMS can be reduced , thus further improving the system performance.

2. Query Optimizaion
   Queries can be rewritten based on the mined empty regions, so that query processing can be optimized.

1.3 Organization

The thesis is organized as follows: Chapter 2 explores the existing algorithms and techniques in mining empty regions. In Chapter 3, the proposed Solution is discussed and in Chapter 4, the algorithm is presented in detail. Chapter 5 outlines an extension to the existing algorithm for mining multiple relations. Results and algorithm evaluation are presented in Chapter 6. We discuss the potential application for the mined result in chapter 7. And lastly, we present our conclusion in Chapter 8.

2

Related Work

2.1 Different definitions used

In previous research literature, many different definitions of empty regions have been used. It is known as holes in [Liu et al., 1997] and Maximal empty Hyper-rectangle(MHR) in [Edmonds et al., 2001]. However, in [Liu et al., 1998], the definition of a hole is relaxed, where regions with low density is considered 'empty'. In this context, our definition of a hole or empty hyper-rectangle is simply a region in the space that contains no data point. They exist because certain value combinations are not possible. In a continuous space, there always exist a large number of holes because it is not possible to fill up the continuous space with data points.

In [Liu et al., 1997], they argued that not all holes are interesting, as they wish to discover only holes that assist them in decision making or discovery of interesting data correlation. However in our case, we aim to mine holes to assist in query processing, regardless of their 'interestingness'.

2.2 Existing Techniques for discovering empty result queries

2.2.1 Incremental Solution

In [Luo, 2006], Luo proposed an incremental solution for detecting empty-result queries. The key idea is to remember and reuse the results from previously executed empty-result queries. Whenever empty-result queries are encountered, the query is analysed and the lowest-level of empty result query parts are stored in C_{aqp}, known as the collection of *atomic query parts*. It claims that the coverage detection capability in this method is more powerful than that of the traditional materialized view method. Consequently, with stored atomic query parts, it can handle empty result queries based on two following scenarios:

1. a more specific query
   It is able to identify using the stored atomic query part, queries that are more specific. In this case, the query will not be evaluated at all.

2. a more general query
   However, if a more general query is encountered, the query will be executed to

determine whether it returns empty. If it does, the new atomic query parts that are more general will replace the old ones.

Below are the main steps of breaking a lowest-level query part P whose output is empty into one or more atomic query parts:

1. **Step 1:** P is transformed into a simplified query part P_s.

2. **Step 2:** P_s is broken into one or more atomic query parts.

3. **Step 3:** The atomic query parts are stored in C_{aqp}.

This method is not specific to any data types, as the atomic queries parts are directly extracted from queries issued by users. Queries considered here can contain point-based comparisons and unbounded-interval-based comparison.

- point-based comparisons: $(A.c =' x')$
- unbounded-interval-based comparisons: $(10 < A.a < 100, B.e < 40)$

Unlike other data mining approaches, this method is built into a RDBMS and is implemented at the query execution level. First, the query execution plan is generated and the lowest-level query part whose output is empty is identified. These parts are then stored in C_{aqp} and kept in memory for efficient checking.

One of main disadvantage of this solution is that it may need to take a long time to obtain a good set of C_{aqp}, due to its incremental nature. Many empty result queries need to be executed first before the RDBMS can have a good detection of empty result queries.

2.2.2 Data Mining Solutions

Mining empty regions or holes is seen to be the complementary approach to existing data mining techniques that focuses on the discovery of dense regions, interesting groupings of data and similiarity between values. There mining methods however can be applied to discover empty regions in the data. There have been studies on mining of such regions, and they can be categorized into the following groups, described individually in each subsection below.

Discovering Holes by Geometry

There are two algorithms that discover holes by using geometrical structure.

(1) Constructing empty rectangles

The algorithm proposed in [Edmonds et al., 2001] aims at constructing or 'growing' empty cells and output all maximal rectangles. This method requires a single scan of a sorted data set. This method works well in two dimension space, and can be generalized to high dimensions, however it might be complicated. The dataset is depicted as an $|X| \times |Y|$ matrix of 0's and 1's. 1 represents the presence of a value and 0 for non-existence of a value, as shown in Figure 2.2(a). First it is assumed that the set X of distinct values (the smaller) in the dimension is small enough to be stored in memory.

Tuples from the database D will be read sequentially from the disk . When 0-entry $\langle x, y \rangle$ is encountered, the algorithm looks ahead by querying the matrix entries $\langle x+1, y \rangle$ and $\langle x, y+1 \rangle$. The resulting area resembles a staircase. Then all maximal rectangles that lie entirely *within* that staircase (Fig 2.1) is extracted. This can be summarized by the algorithm structure below:

```
loop y = 1 ..... n
      loop x = 1 ..... m
         1. Construct staircase (x, y)
         2. Output all maximal 0-rectangles with <x, y>
            as the bottom-right corner.
```

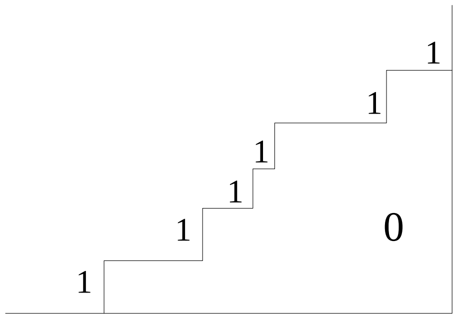


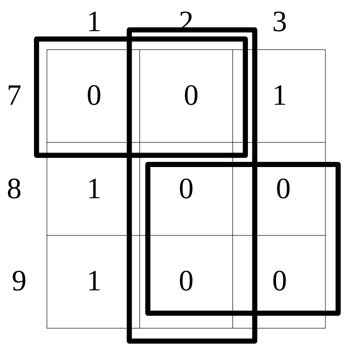
Figure 2.1: Staircase

Consider the following example:

Example 2.1. Let A be an attribute of R with the domain X = (1, 2, 3) and let B be an attribute with domain Y = (7, 8, 9). The matrix M for the data set is shown in Fig 2.2(a). Figure 2.2(b) shows all maximal empty rectangles.

	1	2	3
7	0	0	1
8	1	0	0
9	1	0	0

(a) Matrix table



(b) Overlapping empty rectangles(marked with thick lines)

Figure 2.2: The matrix and the corresponding empty rectangles

This algorithm was extended to mine empty regions in the non-joining portion of two relational tables in [Gryz and Liang, 2006]. It is observed that the join of two relations in real databases are usually much smaller than their Cartesian product. The ranges and values that do not appear together are characterized as empty regions. It is observed that this method generates a lot of overlapping rectangles, as shown in Figure 2.2(b). Even though it is shown that the number of overlapping rectangles is at most O($|X|\,|Y|$), there are still a lot of redundant results and the number of empty rectangles might not accurately show the empty regions in the data.

The number of maximal hyper-rectangles in a d-dimensional matrix is $O(n^{2d-2})$ where $n = |X| \times |Y|$. The complexity of an algorithm to produce them increases exponentially with d. When $d = 2$ dimensions, the number of maximal hyper-rectangles is $O(n^2)$, which is linear in the size $O(n^2)$ of the input matrix. As for $d = 3$ dimensions, the number increases to $O(n^4)$, which is not likely to be practical for large datasets.

(2)Splitting hyper-rectangles.

The second method introduced in [Liu et al., 1997] Maximal Hyper-rectangle(MHR) is formed by splitting existing MHR when a data point is added.

Given k-dimensional continuous space S, and n points in S, they first start with one MHR, which occupies the entire space S. Then each point is incrementally added to S. When a new point is added, they identify all the existing MHRs that contain that point. Using the newly added point as reference, a new lower and upper bound for each dimension is formed to result in 2 new hyper-rectangles along that dimension. If the new hyper-rectangles are found to be sufficiently large, they are inserted into the list of MHRs to be considered. At each insertion, they update the set of MHRs.

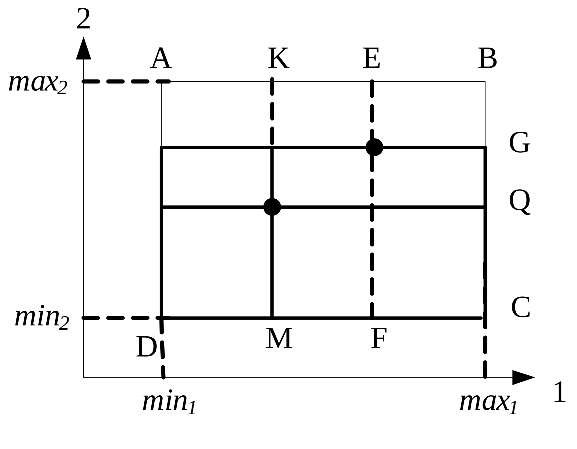


Figure 2.3: Splitting of the rectangles

As shown in Figure 2.3, the new rectangles formed after points P_1 and P_2 are inserted. The rectangles are split along dimension 1 and dimension 2, resulting in new smaller rectangles. Only sufficiently large ones are kept. The proposed algorithm is memory-based and is not optimized for large datasets. As the data is scanned, a data structure is kept storing all maximal hyper-rectangles. The algorithm runs in $O(|D|^{2(d-1)}\, d^3 (log\,|D|)^2)$ where d is the number of dimensions in the dataset. Even in two dimensions ($d = 2$), this algorithm is impractical for large datasets. In an attempt to address both the time and space complexity, the authors proposed to maintain only maximal empty hyper-rectangles that exceed an a user defined minimum size.

The disadvantage of this method is that it is impractical for large dataset. Besides that, results might not be desirable if the dataset is dense. In this case, a lot of small empty regions will be mined. This solution only works for continuous attributes and it does not handle discrete attributes.

Discovering Holes by Decision Trees Classifiers

In [Liu et al., 1998], Liu *et. al*proposed to use decision tree classifiers to (approximately) separate occupied from unoccupied space. They then post-process the discovered regions to determine maximal empty rectangles. It relaxes the definition of a hole, to that of a region with count or density below a certain threshold is considered a hole. However, they do not guarantee that all maximal empty rectangles are found. Their approach transforms the holes-discovery problem into a supervised learning task, and then uses the decision tree induction technique for discovering holes in data. Since decision trees can handle both numeric and categorical values, this method is able to discover empty regions in mixed dimensions. This is an advantage over the other methods as they are not only able to mine empty rectangles in numeric dimensions, but also in mixed dimensions of continuous and discrete attributes.

This method is built upon the method in [Liu et al., 1997] (discussed above under Discovering Holes by Geometry - Splitting hyper-rectangles). Instead of using points as input to split the hyper-rectangle space, this method uses *filled regions*, FR as input to produce maximal holes.

This method consists of the following three steps:

1. **Partitioning the space into cells.**
   Each continuous attribute is first partitioned into a number of equal-length intervals (or units). Values have to be carefully chosen to ensure that the space is partitioned into suitable cell size.

2. **Classifying the empty and filled regions.**
   A modified version of decision tree engine in C4.5 is used to carve the space into filled and empty regions. With the existing points, the number of empty cells are calculated by minusing the filled cells from the total number of cells. With this information, C4.5 will be able to compute the information gain ratio and use it to decide the best split. A decision tree is constructed, with tree leafs labeled *empty*, representing empty regions, and the others the filled region, FR.

3. **Producing all the maximal holes.**
   In this post-processing step, all the maximal holes are produced. Given a k-dimensional continuous space S and n FRs in S, they first start with one MHR which occupies the entire space S. Then each FD is incrementally added to S. At each insertion, the set of MRHs found is updated. When a new FR is added, they identify all the existing NHRs that intersect with this FR. These hyper-rectangles are no longer MHRs since they now contain part of the FR within their interior. They are then removed from the set of existing MHRs. Using the newly added FR as reference, a new lower and upper bound for each dimension are formed to result in 2 new hyper-rectangles along that dimension. If these new hyper-rectangles are found to be MHRs and are sufficiently large, they are inserted into the list of existing MHRs, otherwise they are discarded.

As shown in Figure 2.4, the new rectangles are formed after FRs H_1 and H_2 are inserted. The hyper-rectangles are split, forming smaller MHRs. Only sufficiently large ones are kept. The final set of MHRs are GQNH, RBCS, GBUT, IMND, OPCD, AEFD and ABLK respectively.

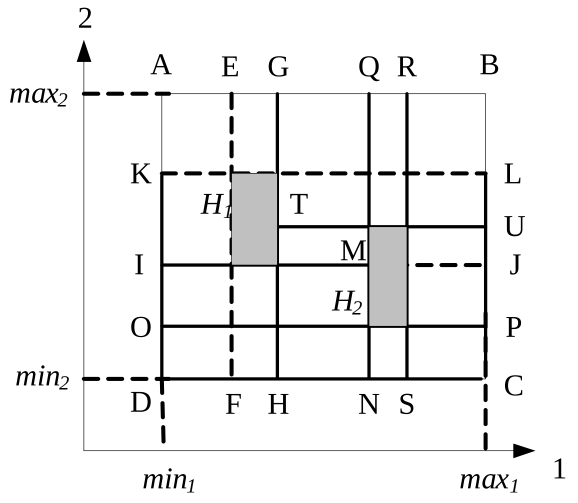


Figure 2.4: Carving out the empty rectangles using decision tree

2.2.3 Analysis

It is true that mining empty regions is not an easy task. As some of the mining methods produce a large number of empty rectangles. To counter this problem, a user input is required and mining is done only on regions with size larger than the user's specification. Another difficulty is the inability of existing methods to mine discrete attributes or the combination of continuous and discrete values. Most of the algorithms scale well only in low-dimensional space, eg: 2 or 3 dimension. They immediately become impractical if mining is needed to be done high dimensional space.

It is stated that not all holes are important in [Liu et al., 1997], however our task is to store as many large discovered empty hyper-rectangles. It is noticed that there are no clustering techniques used. Simple observation is that existing clustering methods find high density data and ignore low density regions. Existing clustering techniques such as CLIQUE [Agrawal et al., 1998] find clusters in subspaces of high dimensional data, but only works for continuous values and only works for high density data and ignore low density ones. In this, we want to explore the usage of clustering in finding empty regions of the data.

3

Proposed Solution

3.1 General Overview

In view of the goal of mining for empty result queries, we propose to mine empty regions in the dataset to achieve that purpose. Empty result queries happens when the required region turns out to be empty, containing no points. Henceforth, we will focus on mining empty region, which is seen as equivalent to mining empty result queries. More specifically, empty regions here mean empty hyper-rectangles. We concentrate on hyper-rectangular regions instead of finding irregular regions described by some of the earlier mining methods, like the staircase-like region [Edmonds et al., 2001] or irregular shaped regions due to overlapping rectangles.

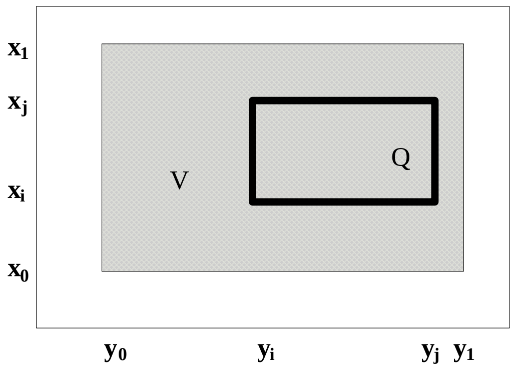


Figure 3.1: Query and empty region

```
SELECT * FROM table
WHERE X BETWEEN x_i AND x_j
AND Y BETWEEN y_i AND y_j;
```

To illustrate the relation between empty result queries and empty regions in database views, consider Figure 3.1 and the SQL query above. The shaded region V is a materialized view depicting an empty region in the dataset and the rectangle box Q with thick lines depicts the query. It is clear the given SQL query is an empty result query since the query falls within the empty region of the view.

To mine these empty regions, the proposed solution uses the clustering method. In a nutshell, all existing points are clustered into clusters, and empty spaces are inferred from them. For this method, no prior domain knowledge is required. With the clustered data, we use the itemset lattice and a levelwise search to permutate and mine for empty regions.

Here are the characteristics of the proposed solution:

- works for both categorical and continuous values.
- allows user can choose the granularity of the mined rules based on their requirement and the available memory.
- characterize regions strictly as rectangles, unlike in [Gryz and Liang, 2006], which has overlapping rectangles and odd-shaped regions.
- implements independently on the size of the dataset, just dependent on the Cartesian product of the size of each attribute domain.
- operates with the running time that is linear in the size of the attributes.

3.2 Terms and definition

Throughout the whole thesis, the following terms and definitions are used.

Database and attributes:

Let D be a database of N tuples, and let A be the set of attributes, $A = \{A_1, ...A_n\}$, with n distinct attributes. Some attributes take continuous values, and other attributes take discrete values. For a continuous attribute A_i, we use min_i and max_i to denote the bounding (minimum and maximum) values. For a discrete attribute A_i, we use $dom(A_i)$ to denote its domain.

Empty result queries

Empty result queries are queries that are processed by a DBMS and returns 0 tuples. The basic assumption is that all relations are non-empty. All tuples in the dataset are complete and contains no missing values. From this, we can assert that all plain projection queries are not empty. Therefore we only consider selection-projection queries, which may potentially evaluates to empty.

1. Projection queries:
   $q = \pi_X(R)$ is not empty if relation R is not empty.
2. Selection-projection queries with one selection:
   $q = \pi_X(\sigma_{A_i=a_i}(R))$ is not empty since $A_i \in A$ and $a_i \in dom(A_i)$.

It is noted that the more specific the query (i.e: a query that has more selection clauses), the more chances that it might evaluate to empty. By using a levelwise method, this solution mines such selection combinations from each possible combination of attributes. In the domain of our testing, we only consider values that are in the domain, as for numeric values, the domain is the set of clusters formed by the continuous values.

Query selection criteria

Query selection criteria can have the following format:

- for discrete attributes: $\{(A_i = a_i)|(A_i \in A) \text{ and } a_i \in dom(A_i))\}$
- for continuous attributes: $\{A_j\ op\ x|x, y \in \mathbb{R},\ op \in \{=, <, >, \leq, \geq\}\}$

Minimal empty selection criteria

We are interested in mining only the minimal empty selection criteria that makes a query empty. We have established that empty result may only appear in selection-projection queries. Therefore we focus our attention to the selection criteria of a query. The property of being empty is monotonic, in which if a set of selection criteria causes a query to be empty, then any superset of it will also be empty.

Definition 3.1. Monotone. Given a set M, E denotes the property of being empty, and E is defined over the powerset of M is *monotone* if

$$\forall S, J : (S \subseteq J \subseteq M \wedge E(S)) \Rightarrow E(J).$$

Monotonicity of empty queries:
Let $q = \pi_X(\sigma_s(R))$ where $X \subseteq A$. The selection criteria $s = \{A_i = a_i, A_j = a_j\}$ where $i \neq j$, $A_i \neq A_j$ and $A_i, A_j \in A$. If we know that $ans(q) = \emptyset$, then any selection $s\prime$, where $s \subseteq s\prime$, will produce an empty result too.

Hence, our task is to mine only **minimal selection criteria** that makes a query empty.

Hyper-rectangle

Based on the above database and query selection criteria definitions, we further define the syntax of empty hyper-rectangles, **EHR**:

1. $(A_{i_j} = v_{i_j})$ with $v_{i_j} \in a_{i_j}$ if A_{i_j} is a discrete attribute, or
2. $(A_{i_j}, l_{i_j}, u_{i_j})$ with $min_{i_j} \leq l_{i_j} \leq max_{i_j}$ if A_{i_j} is a continuous attribute.

EHRs have the following properties:

1. Rectilinear: All sides of **EHR** are parallel to the respective axis, and orthogonal to the others.
2. Empty: **EHR** does not have any points in its interior.

Multidimensional space and subspace:

We have that $A = \{A_1, A_2, \ldots, A_n\}$, let $S = A_1 \times A_2 \times .... \times A_n$ be a n-dimensional space where each dimension A_i is of two possible types: discrete and continuous. Any space whose dimensions are a subset of $A_1, A_2, ..., A_n$ is known as the *subspace* of S. we will refer to $A_1, \ldots, A_n$ as the dimensions of S.

3.2.1 Semantics of an empty region

This solution works on heterogeneous attributes, and here we focuses on two common types of attributes, namely discrete and continuous valued attributes. With this, we have three different combinations of attribute types, forming three different types of subspaces. Below we give the semantic definitions of an empty region of each such subspaces:

Discrete attributes:

We search for the combination of attribute-value pairs that does not exist in the database. Consider the following illustration:
A_1: $dom(A_1) = \{a_{1_1}, a_{1_2}, ..., a_{1_m}\}$, where $|A_1| = m$.
A_2: $dom(A_2) = \{a_{2_1}, a_{2_2}, ..., a_{2_n}\}$, where $|A_2| = n$.
Selection criteria, σ: $A_1 = a_i \wedge A_2 = b_j$
where $a_{1_i} \in dom(A_1)$ and $a_{2_j} \in dom(A_2)$, but this combination, $A_1 = a_i \wedge A_2 = b_j$ does not exist in any tuple in the database.

Continuous attributes:

An empty hyper-rectangle is a subspace that consist of only connected empty cells. Given a k-dimensional continuous space S bounded in each dimension i $(1 \leq i \leq k)$ by a minimum and a maximum value (denoted by min_i and max_i), a hyper-rectangle in S is defined as the region that is bounded on each dimension i by a minimum and a maximum bound. A hyper-rectangle has $2k$ bounding surfaces, 2 on each dimension. The two bounding surfaces on dimension i are parallel to axis i and orthogonal to all others.

Mixture of continuous and discrete attributes:

Without loss of generality, we assume that the first k attributes are discrete attributes, $A_1, ...A_k$, and the rest are continuous attributes, $A_{k+1}, ...A_m$. A cell description is as follows:

$$(discrete - cell, continuous - cell)$$

which $discrete - cell$ is a cell in the discrete subspace, which is $((A_1 = a_1), ...)$ with $a_i \in dom(A_i)$, and $continuous - cell$ is a cell in the continuous subspace for $A_{k+1}, ...A_m$. An hyper-rectangle is represented with

$$(discrete - region, continuous - region)$$

It is a region consisting of a set of empty cells, where $discrete - region$ is a combination of attribute-value pairs and $continuous - region$ is a set of connected empty cells in the continuous subspace.

In theory, an empty region can be of any shape. However, we focus only on hyper-rectangle regions instead of irregular regions as described by a long disjunction of conjunctions (Disjunctive Normal Form used in [Agrawal et al., 1998]), or as x-monotone regions described in [Fukuda et al., 1996].

3.3 Method

3.3.1 Duality concept

We present here the duality concept of the itemset lattice in a levelwise search. Consider the lattice illustrated in Figure 3.2, (with the set of attributes A = {A, B, C, D}). A typical frequent itemset algorithm looks at one level of the lattice at a time, starting from the empty itemset at the top. In this example, we discover that the itemset {D} is not 1-frequent (it is empty). Therefore, when we move on to successive levels of lattice, we do

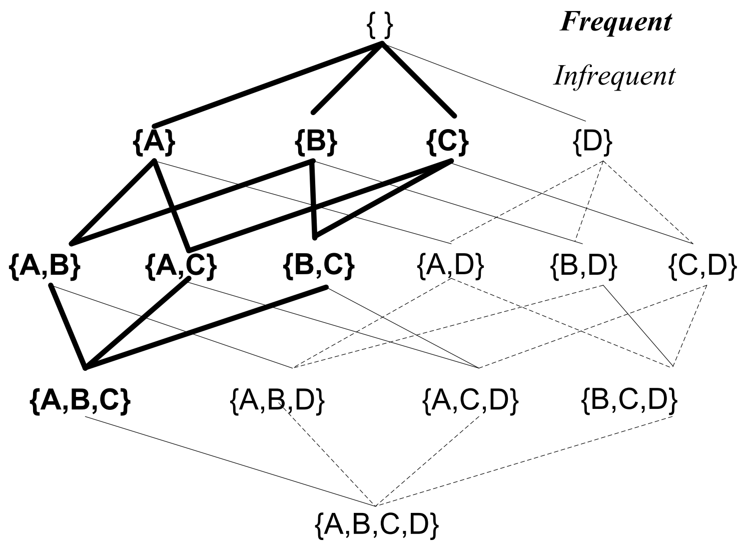


Figure 3.2: The lattice that forms the search space

not have to look at any supersets of {D}. Since half of the lattice is composed of supersets of {D}, this is a dramatic reduction of the search space.

Figure 3.2 illustrates this duality. Supposed we want to find the combinations that are empty. We can again use the level-wise algorithm, this time starting from the maximal combination set = {A, B, C, D} at the bottom. As we move up levels in the lattice by removing elements from the combination set, we can eliminate all subset of a non-empty combination. For example, the itemset {A, B, C} is empty, so we can remove half of the algebra from our search space just by inspecting this one node.

There are two trivial observations here:

1. Apriori can be applied to any constraint P that is antimonotone. We start from the empty set and prune supersets of sets that do not satisfy P.
2. Since itemset lattice can be seen as a boolean algebra, so Apriori also applies to a monotone Q. We start from the set of all items instead of the empty set. Then prune subsets of sets that do not satisfy Q.

The concept of duality is useful in this context, as the proposed solution is formulated based on it. Further discussion of the usage of this duality is found in the next chapter, where the implementation of the solution is also discussed in detail.

4 Algorithm

From the duality concept defined earlier, we could tackle the problem of mining empty hyper-rectangles, known as **EHR** using either the positive pruning or the negative pruning approach. This is further elaborated in the following:

1. Monotonicy of empty itemset
   We have seen that the property of being empty is monotonic, and we could make use of negative pruning in the lattice. If k-combination is empty, then all $k+1$-combination is also empty. Starting from the largest combination, any combination proven not to produce an empty query, is pruned off from the lattice, and subsequently all of its subset are not considered. This is known as negative pruning.

2. Anti-monotonicity of 1-frequent itemset
   Interestingly, this property can be characterized in the form of its duality. A k-combination is empty also means that that particular combination does not have a frequency of at least 1. Therefore, this problem can bee seen as mining all 1-frequent combination. This in turn is anti-monotonic. If k-combination is not 1-frequent, then all $k+1$-combination cannot be 1-frequent. In this case, we can use the positive pruning in the lattice. Candidate itemset that are pruned away are the empty combinations that we are interested in.

A selection criteria will be used to determine which method would evaluate best for the given circumstances. This will be discussed in Section 4.3. In this chapter, we focus mainly on mining for **EHR**, empty regions bounded by continuous values. As we shall see that the same method can be generalized and be applied to mine empty heterogeneous itemsets. All mined results are presented in the form of rules.

4.1 Preliminaries

4.1.1 Input Parameters

We require user to input parameters for this algorithm, however input parameters are limited to the minimum, because the more input parameters we have, the harder it is to provide the right combination for an optimum performance of the algorithm. User can input the coverage parameter, τ. This will determine how fine grain the empty rectangles will be. The possible values of τ ranges between 0 and 1. The detail usage of this parameter is discussed in Section 4.4.3.

4.1.2 Attribute Selection

As mentioned earlier, mining empty result queries are restricted only on attributes that are frequently referenced together in the queries. They can also be set of attributes selected and specified by the user. Statistical techniques for identifying the most influential attributes for a given dataset, such as *factor analysis* and *principal component analysis* stated in [Lent et al., 1997] could also be used. In this algorithm, we consider both discrete and continuous attributes.

1. Discrete Attributes:
   Only discrete attributes with low cardinality are considered. As it makes no sense to mine attributes with large domains like 'address' and 'names'. We limit the maximal cardinality of each domain to be 10.

2. Continuous Attributes:
   Continuous values are unrestricted, they can have an unbounded range. They will be clustered and we limit the maximum number of clusters or bins to be 100.

4.1.3 Maximal set, max_set

The maximal set is the Cartesian product of the values in the domain of each attribute. It makes up the largest possible search space for the algorithm. It represents all the possible permutations of the values. The maximal attribute set, max_set .

$$max\_set = \{dom(A_1) \times dom(A_2) \times \ldots \times dom(A_n)\} \tag{4.1}$$

and the size of max_set is:

$$|max\_set| = |dom(A_1)| \times |dom(A_2)| \times \ldots |dom(A_n)| \tag{4.2}$$

4.1.4 Example

Consider Table 4.1, it contains the the flight information. This toy example will be used throughout this chapter.

Flight			
Flight_No	airline	destination	price
F01	SkyEurope	Athens	0.99
F02	SkyEurope	Athens	1.99
F03	SkyEurope	Athens	0.99
F04	SkyEurope	Vienna	49.99
F05	SkyEurope	London	299.99
F06	SkyEurope	London	48.99
F07	SkyEurope	London	49.99
F08	EasyJet	Dublin	299.99
F09	EasyJet	Dublin	300.00
F10	EasyJet	Dublin	300.99

Table 4.1: Flight information table

Main Algorithm:

Here are the main steps of the algorithm (shown in a form of a flowchart below):

1. Data Preprocessing
2. Encoding of database into a simplified form.
3. Selection of method based on size criteria
   a) method 1: anti-monotone pruning
   b) method 2: monotone pruning

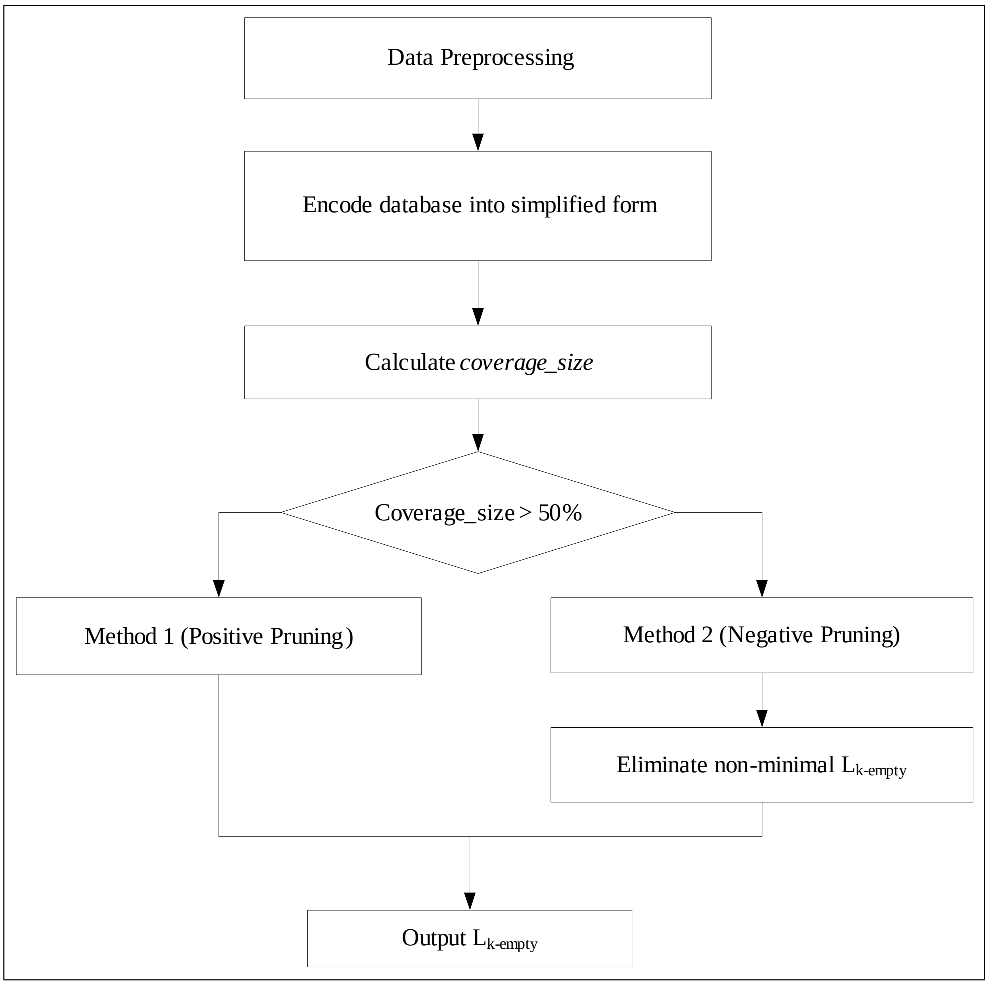

Data Preprocessing
Encode database into simplified form
Calculate *coverage_size*
Coverage_size > 50%
Method 1 (Positive Pruning)
Method 2 (Negative Pruning)
Eliminate non-minimal $L_{k\text{-empty}}$
Output $L_{k\text{-empty}}$

Figure 4.1: Main process flow

4.2 Step 1: Data Preprocessing

In this phase, numeric attributes are clustered, by grouping them according to their distance. Data points that are located close to each other are clustered into the same cluster. Our goal is to find clusters or data groups that are compact (the distance between points within a cluster is small) and isolated (relatively separated from other data groups). In a generalized manner, this can be seen as discretizing the continuous values into meaningful ranges based on the data distribution. Existing methods include *equi-depth*, *equi-width* and *distance-based* that partitions continuous values into intervals. However, both *equi-depth* and *equi-width* are found not to be able to capture the actual data distribution, thus are not suitable to be used in the discovery of empty regions. Here, an approach that is similar to *distance-based* partitioning as described in [Miller and Yang, 1997] is used.

Existing clustering strategies aims at finding densely populated region in a multi-dimensional dataset. Here we only consider finding clusters in one dimension. As explained in [Srikant and Agrawal, 1996], one dimensional cluster is the range of smallest interval containing all closely located points. We employ an agglomerative hierarchical clustering method to cluster points in one dimension. An agglomerative, hierarchical clustering starts by placing each object in its own cluster and then merges these atomic clusters into larger and larger clusters until all objects are in a single cluster.

The purpose of using hierarchical clustering is to identify groups of clustered points without recourse to the individual objects. In our case, it is desirable to identify as large clustered size as possible because these maximal clusters are able to assist in finding larger region of empty hyper-rectangles. We shall see the implementation of this idea in the coming sections.

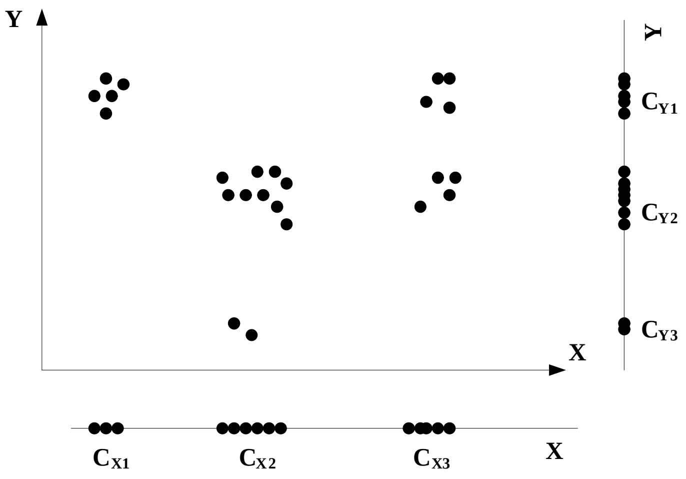


Figure 4.2: Clusters formed on X and Y respectively

Figure 4.2 shows a graph with points plotted with respect to two attributes, X and Y. These points are clustered into two sets of clusters, one in dimension X, and the other in dimension Y. When the dataset is projected on the X-axis, 3 clusters are formed, namely C_{X1}, C_{X2} and C_{X3}. Similarly, C_{Y1}, C_{Y2} and C_{Y3} are 3 clusters formed when the dataset is projected on the Y-axis.

The clustering method used here is an modification to the BIRCH(Balanced Iterative Reducing and Clustering using Hierarchies) clustering algorithm introduced by Zhang *et al.* in [Zhang et al., 1996]. It is an adaptive algorithm that has linear IO cost and is able to handle large datasets.

The clustering technique used here shares some similarities with BIRCH. First they both uses the distance-based approach. Since data space is usually not uniformly occupied, hence, not every data point is equally important for the clustering purpose. A dense region of points is treated collectively as a single cluster. In both techniques, clusters are organised and characterized by the use of an in-memory, height-balance tree structure (similar to a B+-tree). This tree structure is known as CF-tree in BIRCH, while here it is referred as Range Tree. The fundamental idea here is that clusters can be incrementally identified and refined in a single pass over the data. Therefore, the tree structure is dynamically built, and only requires a single scan of the dataset. Generally, these two approaches share the same basic idea, but differs only in two minor portions: First, the CF Tree uses Clustering Feature(CF) for storage of information while the Range tree stores information in the leaf nodes. Second, CF tree is rebuilt when it runs out of memory, Range Tree on the other hand is fixed with a maximum size, hence will not be rebuilt. The difference between this two approach will be highlighted in detail as each task of forming the tree is described.

Storage of information

In BIRCH, each cluster is represented by a Clustering Feature (CF) that holds the summary of the properties of the cluster. The clustering process is guided by a height-balanced tree of CF vectors. A Clustering Feature is s a triple summarizing the information about a cluster. For $C_x = \{t_1, \ldots, t_N\}$, the CF is defined as:

$$CF(C_x) = (N, \sum_{i=1}^{N} t_i[\vec{X}], \sum_{i=1}^{N} t_i[X]^2)$$

where N is the number of points in the cluster, second being the *linear sum* of N data points, and third, the *squared sum* of the N data points. BIRCH is a clustering method for multidimensional and it uses the *linear sum* and *square sum* to calculate the distance between a point to the clusters. Based on the distance, a point is placed in the nearest cluster.

The Range-tree stores information in it's leaf nodes instead of using Clustering Feature. Each leaf node represents a cluster, and they store summary of information of that cluster. Clusters are created incrementally and represented by a compact summary at every level of the tree. Let the root of the hierarchy be at level 1, it's children at level 2 and so on. A node in level i corresponds to the union of range formed by its children at level $i + 1$. For each leaf node in the Range tree, it has the following attributes:

$$LeafNode(min, max, N, sum)$$

- min: the minimum value in the cluster
- max: the maximum value in the cluster
- N: the number of points in the cluster
- sum: $\sum_{i=1}^{N} X_i$

The centroid of a cluster is the mean of the points in the cluster. It is calculated as follows:

$$centroid = \frac{\sum_{i=1}^{N} X_i}{N} \tag{4.3}$$

The distance between clusters can be calculated using the centroid Euclidean distance, $Dist$. It is calculated as follows:

$$Dist = \sqrt{(centroid_1 - centroid_2)^2} \tag{4.4}$$

Together, min and max defines the left and right boundaries of the cluster. The distance of a point to a cluster is the distance of the point from the centroid of that cluster. A new point is placed in the nearest cluster, i.e. the centroid with the smallest distance to the point.

From the leaf nodes, the summary of information can be derived and stored by their respective parent node (non-leaf node). So a non-leaf node represents a cluster made up of all the subcluster represented by its entry. Each non-leaf node can be seen as providing a summary of all the subcluster connected to it. The parent nodes, $node_i$ in the Range tree, contains the summary information of their children node, $node_{i+1}$:

$$Non-leafNode(min_i, max_i, N_i, sum_i)$$

where $min_i = MIN(min_{i+1})$, $max_i = MAX(max_{i+1})$, $N_i = \sum N_{i+1}$ and $sum_i = \sum sum_{i+1}$

Range Tree

The Range-tree is built incrementally by inserting new data points individually. Each data point is inserted by locating the closest node. At each level, the closest node is updated to reflect the insertion of the new point. At the lowest level, the point is added to the closest leaf node. Each node in the Range-Tree is used to guide a new insertion into the correct subcluster for clustering purposes, just the same as a B+-tree is used to guide a new insertion into the correct position for sorting purposes. The range tree is a height-balanced tree with two parameters: branching factor B and size threshold T.

In BIRCH, the threshold T is initially set to 0 and the B is set based on the size of the page. As data values are inserted into the CF-tree, if a cluster's diameter exceeds the threshold, it is split. This split may increase the size of the tree. If the memory is full, a smaller tree is rebuilt by increasing the diameter threshold. The rebuilding is done by re-inserting leaf CF nodes into the tree. Hence, the data or the portion of the data that has already been scanned does not need to be rescanned. With the higher threshold, some clusters are likely to be merged, reducing the space required by the tree.

In our case, the maximum size of the Range-tree is bounded, and therefore will not have the problem of running out of memory. We have placed a limit on the size of the tree by predefining the threshold T with the following calculation:

$$T = \frac{(maximum\_value - minimum\_value)}{max\_cluster} \tag{4.5}$$

where *maximum_value* and *minimum_value* are the largest and the smallest value respectively for an attribute.

The maximum number of cluster, *max_cluster* for each continuous value is set to be 100. We have set the branching factor $B = 4$, where each nonleaf node contains at most 4 children. A leaf node must satisfy a *threshold requirement*, in which the *interval_size* of the cluster has to be less than the threshold T. The *interval_size* of each clustering is the range or interval bounded by between the *min* and the *max*.

$$interval\_size = max - min \tag{4.6}$$

Insertion into a Range Tree

When a new point is inserted, it is added into the Range-tree as follows:

1. *Identifying the appropriate leaf:* Starting from the root, it recursively descends the Range-tree by choosing the **closest** node where the point is located within the *min* and *max* boundary of a node.

2. *Modifying the leaf:* When it reaches a leaf node, and if the new point falls within the *min* and *max* boundary, it is 'absorbed' into the cluster, values of N and *sum* are then updated. If a point does not fall within the *min* and *max* boundary of a leaf node, it first finds the closest leaf entry, say L_i, and then test whether *interval_size* of the augmented cluster satisfy the threshold condition. If so, the new point is inserted into L_i. All values in L_i: (*min*, *max* and N, *sum*) are updated accordingly. Otherwise, a new leaf node will be formed.

3. *Splitting the leaf:* If the number of branch reaches the limit of B, the branching factor, then we have to *split* the leaf node. Node splitting is done by choosing the **farthest** pair of entries as seeds, and redistributing the remaining entries. Distance between the clusters are calculated using the centroid Euclidean distance, *Dist* in Equation 4.4. The levels in the Range tree will be balanced since it is a height-balanced tree, much in the way a B+-tree is adjusted in response to the insertion of a new tuple.

In the Range-tree, the smallest clusters are stored in the leaf nodes, while non-leaf nodes stores the summary of the union of a few leaf nodes. The process of clustering can be seen as a form of discretizing the values into meaningful ranges.

In the last step of data preprocessing, the values in a continuous domain are replaced with their respective clusters. The domain of the continuous attribute now consist of the smallest clusters. Let A_i be an attribute with continuous values, and it is partitioned into m clusters, $c_{i1}, \ldots c_{im}$. Now $dom(A_i) = \{c_{i1}, c_{i2}, \ldots c_{im}\}$.

Example 4.1. In the *Flight Information* example, the attribute *price* is a continuous attribute, with values ranging from 0.99 to 300.99.
minimum_value = 0.99, *maximum_value* = 300.99, and we have set *max_cluster* = 100.
T is calculated as follows:

$$T = \frac{(300.99 - 0.99)}{100} = 3$$

Any points that falls within the distance threshold of T is absorbed into a cluster, while all points with the interval_size larger than T will form a cluster of its own. So all clusters have the maximum *interval_size* of less than T. The values in *price* are clustered in the following clusters:
C1: [0.99, 1.99], C2: [48.99, 49.99], C3: [299.99, 300.99].
Now the domain of *price* is defined in terms of these clusters, *dom*(*price*) = {[0.99, 1.99], [48.99, 49.99], [299.99, 300.99]}. The values in price are replaced with their corresponding clusters, as shown in Table 4.2.

Calculation of the size of the maximal set

After all the continuous attributes are clustered, now the domain of the continuous attributes are represented by the clusters. The cardinality of the domain is simply the number of clusters. With these information, we are able to obtain the count of the maximal set defined in Section 4.1. Recall that maximal set is obtained by doing the permutation of values in the domain. From there, we can obtain the maximal set simply by permutating the values in the domain for each attributes. This information will be used in determining which method to use, discussed in detail in Section 4.3.

Flight			
Flight_No	airline	destination	price
F01	SkyEurope	Athens	[0.99, 1.99]
F02	SkyEurope	Athens	[0.99, 1.99]
F03	SkyEurope	Athens	[0.99, 1.99]
F04	SkyEurope	Vienna	[48.99, 49.99]
F05	SkyEurope	London	[299.99, 300.99]
F06	SkyEurope	London	[48.99, 49.99]
F07	SkyEurope	London	[48.99, 49.99]
F08	EasyJet	Dublin	[299.99, 300.99]
F09	EasyJet	Dublin	[299.99, 300.99]
F10	EasyJet	Dublin	[299.99, 300.99]

Table 4.2: Attribute *price* is labeled with their respective clusters

Example 4.2. Based on the Flight example, the size of the maximal set is calculated as follows:
$dom(airline)$ = {SkyEurope, EasyJet}
$dom(destination)$ = {Athens, Vienna, London, Dublin}
$dom(age)$ = {[0.99, 1.99], [48.99, 49.99], [299.99, 300.99]}

$$\begin{aligned} |max\_set| &= |dom(airline)| \times |dom(destination)| \times |dom(age)| \\ &= 24 \end{aligned}$$

4.3 Step 2: Encode database in a simplified form

In step 1, each continuous values are replaced with their corresponding cluster. The original dataset can then be encoded into a simplified form. The process of simplifying is to get a reduced representation of the original dataset, where only the distinct combination of values are taken into consideration. Then we assign a tuple ID, known as the TID to each distinct tuples. By doing this, we are reducing the size of the dataset, thus improving the runtime performance and the space requirement of the algorithm.

Since in essence, we are only interested in whether or not the combination exist, we do not need other information like the actual frequency of the occurence of those values. It is the same like encoding the values in 1 and 0, with 1 to show that a value exist and 0 otherwise. Unlike the usual data mining task, eg: finding densed region, clusters or frequent itemset, in our case we do not need to know the actual frequency or density of the data. The main goal is to have a reduction in the size of the original database. The resulting *simplified* database D_s will have only distinct values for each tuple.

$$t_i = \langle a_{1_i}, a_{2_i}, \ldots, a_{n_i} \rangle$$

$$t_j = \langle a_{1_j}, a_{2_j}, \ldots, a_{n_j} \rangle$$

such that if $i \neq j$, then $t_i \neq t_j$, where $i, j \in \{1, \ldots, n\}$.
We sequentially assign a unique positive integer to D_s as an identifier for each tuple t.

Example 4.3. Consider the Table 4.2. It can be simplified into Table 4.3, D_s where it is the set of unique tuples with respect to the attributes *airline, destination and price.* These unique tuples are assigned an identifier, called TID.

Flight			
TID	airline	destination	price
1	SkyEurope	Athens	[0.99, 1.99]
2	SkyEurope	Vienna	[48.99, 49.99]
3	SkyEurope	London	[299.99, 300.99]
4	SkyEurope	London	[48.99, 49.99]
5	EasyJet	Dublin	[299.99, 300.99]

Table 4.3: The simplified version of the Flight information table

$|D_s| = 5$. In this example, D_s is 50% smaller than the original dataset.

Correctness

D_s is a smaller representation of the original dataset. Information is preserved in this process of shrinking. Each point in a continuous attribute is grouped and represented by their respective clusters. Therefore D_s is just a coarser representation of the original dataset, where the shape of the original data distribution is preserved. This simplification does not change or affect the mining of **EHR**.

Distribution size

Next we calculate the percentage of the data distribution with the following equation:

$$distribution\_size = \frac{|D_s|}{|max\_set|} \tag{4.7}$$

Example 4.4. In our example, we have $|D_s| = 5$, and $|max\_set| = 24$

$$distribution\_size \quad = \quad \frac{5}{24} = 0.208$$

Trivial observation: only 21% of the value combination exist. The rest of them, with 79% are empty combinations, i.e: value combinations that does not exist in the dataset.

Method selection

$distribution\_size$ is used to determine which of the two methods to use. If $distribution\_size \leq 0.5$, we will choose method 1. On the other hand method 2 will be chosen if $distribution\_size$ is > 0.5. Observation on the value of $distribution\_size$ are as follows:

- $distribution\_size = 1$: For this case, data distribution covers all the possible combination of n-dimensions. In this situation, mining of **EHR** will not be done.
- $0 < distribution\_size \leq 0.5$: Data distribution covers less than half of the combination. Data distribution is characterized as **sparse**.
- $0.5 < distribution\_size < 1$: Data distribution covers more than half of the combination. Data distribution is characterized as **dense**.

Based on the above observation, we justify the selection of each method as follows. In *Levelwise Search and Borders of Theories in Knowledge Discovery*, the concept of negative border discussed is useful in analysing the levelwise algorithm, like the Apriori algorithm. It is stated that in Thoerem 1, the Apriori algorithm uses $|Th(L, r, q) \cup Bd^{-}(Th(L, r, q))|$ evaluations of being *frequent*.

Example 4.5. Consider the discovery 1-frequent sets with attributes R=A,. . . , F. Assume the collection Th of frequent sets is

$$Th = \{\{A\}, \{B\}, \{C\}, \{F\}, \{A, B\}, \{A, C\}, \{A, F\}, \{C, F\}, \{A, C, F\}\}$$

The positive border of this collection contains the maximal 1-frequent sets, i.e

$$Bd^{+} = \{\{A, B\}, \{A, C, F\}\}$$

The negative border, in turn, contains minimal empty sets. The negative border is thus

$$Bd^{-}(Th) = \{\{D\}, \{E\}, \{B, C\}, \{B, F\}\}$$

The simplified dataset, D_s is the maximal 1-frequent sets. If the size of D_s is small, then that would mean a lot of candidate itemset has been pruned off along the lattice, indicating that the negative border of the itemset is low. Since the evaluation is related to the negative border, it is desirable to have a relatively low negative border. Therefore, method 1 works well in this situation.

If the size of D_s is large, the levelwise method that starts the search from small combinations obviously is further from the optimum. In this case, the levelwise search can start from the large combinations. In method 2, the complement is D_s is used instead. The complement set obtained here is the maximal empty sets (the largest or the most specific combination of attributes). A large D_s would generate a small complement set. Since the complement set is not large, therefore method 2 is more suitable. Starting from the bottom of the lattice, we move levelwise upwards.

Notation	**Description**
itemset	Since we are using the We follow the notation used in the Apriori algorithm for mining association rules. In this context, itemset is analogous to dimension.
k-itemset	A set with k-dimensions.
C_k	Set of candidate k-itemset.
TID	The identifier associted with each unique tuple in D_s.
idlist	The list of TIDs
D_s	The *simplified* database.
$L_{k-empty}$	The minimal empty combination of size k
L_{empty}	The union of all minimal empty combination, $\bigcup_k L_{k-empty}$
max_set	The set of possible permutation of values in each domain.
n	Number of dimensions.

Table 4.4: Notations used in this section

4.4 Step 3 - Method 1:

Method 1 uses the 'top-down' approach, both in the itemset lattice and in the Range-tree traversal. We start our search from the top of the lattice with the Apriori algorithm [Agrawal and Srikant, 1994]. The interesting feature of this algorithm is that the database D_s is not used for counting support after the first pass. In the phase of generating 1-itemset, D_s is scanned and *partitions* for each attribute is created. (The idea *partition* is described in Section 4.4.1). At each level of the lattice, candidate itemsets are association with an array called *idlist*, which stores the list of TIDs. In the subsequent phases for generating $k+1$-itemset, we do not need to rescan D_s, as we can mine **EHR** in the 'offline' mode, by just using k-itemset and their *idlist* stored in their partition. Similar to the Apriori algorithm of generating large-itemset, $k+1$-candidate itemset is formed by self-joining k-itemset, and the new $idlist_{k+1}$ is the set intersection of $idlist_k$. Mining **EHR** is done through the set manipulation of the list of TIDs.

Algorithm 1: Method 1

```
input  : $D_s$, the simplified database
output: $L_{k-empty}$,the minimal empty combination set
1  forall tuple $t \in D_s$ do                                /* see Section 4.4.1 */
2      constructPartition()
3  for ($k = 2$; $L_{k-1} \neq \emptyset$; $k$++) do
4      $c_k$ = joinSubspace($L_{k-1}$)
5      $L_k$ = { $c \in c_k$ where ($c.idlist \neq \emptyset$)}
6      $c_{k-empty}$ = { $c \in c_k$ where ($c.idlist = \emptyset$)}
7      if hasEmptySubset($c_{k-empty}$, $L_{k-1}$) then     /* anti-monotonic pruning */
8          delete $c_{k-empty}$
9      else
10         $L_{k-empty} = c_{k-empty}$
11         ouput $L_{k-empty}$
```

Procedure hasEmptySubset(*$c_{k-empty}$: candidate k-itemset, L_{k-1}: $k-1$-itemset:*)

```
input  : $c_{k-empty}$: candidate $k$-itemset, $L_{k-1}$: $k-1$-itemset
output: Boolean: TRUE, FALSE
1  foreach $k-1$-subset $s$ of $c_{k-empty}$ do
2      if $s \notin L_{k-1}$ then
3          return TRUE
4  return FALSE
```

4.4.1 Generating 1-dimension candidate:

In the 1-itemset generating phase, we start off with constructing the *partitions* for each attribute, in the procedure of *constructPartition*(), described as follows:

Set partition

Each object in the domain is associated with their TID. The TID are stored in the form of *set partition*, a set of non-empty subsets of x such that every element $x_i \in X$ is exactly one of the subset. The fundamental idea underlying our approach is to provide a reduced representation of a relation. This can be achieved using the idea of partitions [Spyratos, 1987, Huhtala et al., 1998]. Semantically, this has been explained in [Cosmadakis et al., 1986, Spyratos, 1987].

Definition 4.1. Partition. Two tuples t_i and t_j are *equivalent* with respect to a given attribute set X if $t_i[A] = t_j[A]$, for $\forall\ A \in X$. the *equivalance class* of a tuple $t_i \in r$ with respect to a given set X $\subseteq R$ is defined by $[t_i]_X = \{t_j \in r \ / \ t_i[A] = t_j[A], \forall\ A \in X\}$. The set $\pi_X = \{[t]_X \ / \ t \in r\}$ of equivalance classes is a *partition* of r under X. That is, π_X is a collection of disjoint sets of rows, such that each set has a unique value for the attribute set S, and the union of the sets equals the relation.

Example 4.6. Consider the simplified dataset in Table 4.3.
Attribute airline has value 'SkyEurope' in rows t_1, t_2, t_3 and t_4, so they form an equivalence class $[t_1]_{airline} = [t_2]_{airline} = [t_3]_{airline} = [t_4]_{airline} = \{1, 2, 3, 4\}$. The whole partition with respect to attribute *airline* is $\pi_{airline} = \{\{1, 2, 3, 4\}, \{5\}\}$. The table below summarizes the partitions for L_1, the 1-itemset.

Attribute	**Domain**	**Partition**
airline	dom(airline) = {SkyEurope, EasyJet}	$\pi_{airline} = \{\{1, 2, 3, 4\}, \{5\}\}$
destination	dom(destination) = {Athens, Vienna, London, Dublin}	$\pi_{destination} = \{\{1\}, \{2\}, \{3, 4\}, \{5\}\}$
price	dom(price) = {[0.99, 1.99], [48.99, 49.99], [299.99, 300.99]}	$range\_tree(price).child_1 = \{1\}$, $range\_tree(price).child_2 = \{2, 4\}$, $range\_tree(price).child_3 = \{3, 5\}$

Table 4.5: Partitions for L_1

We store *idlist* with respect to the partitions in the following form:

- discrete values: set partitions are stored in a vector list called *idlist*.
- continuous values: set partitions are stored only in the leaf node of the Range-tree. The leaf nodes in a Range tree represents the smallest clusters. A nonleaf node, or known as a *parent node* represents a cluster made up of all the subclusters of it's children node. The $idlist_i$ of a non-leaf node in level i are the union of all $idlist_{i+1}$ of its child nodes are level i+1. To conserve space, *idlist* are only stored on the leaf nodes. The *idlist* at a parent node are collected dynamically when needed by doing post-order traversal on the Range-tree.

4.4.2 Generating k-dimension candidate

Subspace Clustering

First we start with low dimensionality, eliminating as many empty combinations as possible before continuing with a higher dimension. The concept of candidate generation for each

```
Procedure joinSubspace($L_{k-1}$): $k-1$-itemset
  input  : $L_{k-1}$, $k-1$-itesetm
  output: $c_k$, $k$ candidate itemset
1 foreach (itemset $l_1 \in L_{k-1}$) do
2     foreach (itemset $l_2 \in L_{k-1}$) do
3         if $(l_1[1] = l_2[1]) \wedge \ldots (l_1[k-2] = l_2[k-2])$ then
4             c = $l_1 \bowtie l_2$
5             c.idlist = $l_1.idlist \cap l_2.idlist$
6 return $c_k$
```

level in is similar to that of Apriori algorithm [Agrawal and Srikant, 1994]. Candidate k-dimension are generated by self-joining k-1 dimensions. The new *set partition* for a candidate k-itemset is generated by obtaining the set intersection of the *idlist* of two $(k-1)$-itemset (refer to line 5 in Procedure $joinSubspace()$).

A candidate is said to agree on the new *set partition* if its resulting *idlist* is not empty. Empty combinations on the other hand are the combination of attributes disagree on a partition, where their resulting *idlist* is empty. These empty combinations are pruned away and output as results.

Example 4.7. Continuing from the example in 4.6, we generate candidate-2, C_2 from existing itemset, L_1 that are listed in Table 4.5. Below here is a portion of C_2 generation:

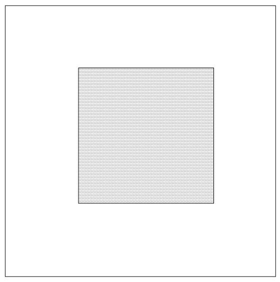
$$\begin{array}{lcll}
\{SkyEurope, Athens\} & : & \{\{1, 2, 3, 4\} \cap \{1\}\} & = & \{1\} \\
\{SkyEurope, Vienna\} & : & \{\{1, 2, 3, 4\} \cap \{2\}\} & = & \{2\} \\
\{SkyEurope, London\} & : & \{\{1, 2, 3, 4\} \cap \{3, 4\}\} & = & \{3, 4\} \\
\{SkyEurope, Dublin\} & : & \{\{1, 2, 3, 4\} \cap \{5\}\} & = & \emptyset \\
\ldots & & & &
\end{array}$$

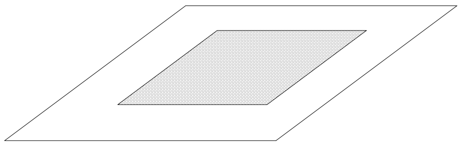
The new partition is $\pi_{\{airline,destination\}}$ $\{\{1\}, \{2\}, \{3, 4\}, \{5\}\}$. The other combinations with empty set intersection are output as results.

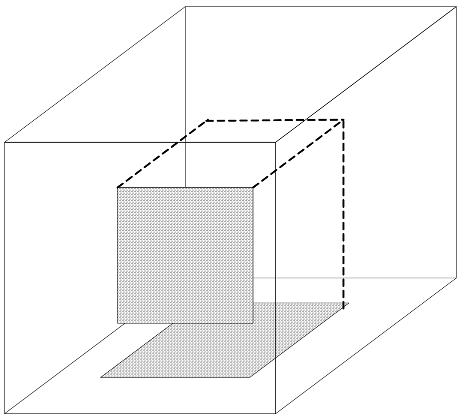
4.4.3 Joining adjacent hyper-rectangles

The idea of joining adjacent hyper-rectangles is illustrated in the procedure $joinSubspace()$. We merge adjacent hyper-rectangles on each k-1 dimension. Joining adjacent hyper-rectangle is seen as finding the connected component in the common face. The Range-tree is traversed in the top-down, depth-first fashion, starting from the big cluster, and then to smaller clusters. Users can specify the granularity of the size of the empty hyper-rectangle. This will determine how to deep to traverse the tree.

We merge the common k-2 dimension by finding the new *set partition*, testing the set intersection of their *idlist*. If the set intersection is not empty, then the hyper-rectangles are formed as a cluster with a higher dimension. On the other hand, if the set intersection is empty, then an **EHR** is formed. This is illustrated in Figure 4.3.


(a) A filled rectangle in 2-dimension


(b) A filled rectangle in 2-dimension


(c) Joining at a connected component

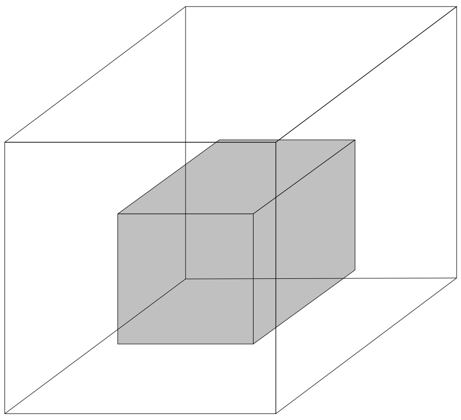

(d) Forming of an **EHR**

Figure 4.3: Joining adjacent hyper-rectangle

Joining of continous attributes:

In this case, Range-trees are compared in a top-down approach. The tree is travered depth first based on the user's threshold input, τ. If the threshold is low, a bigger chunks of the of empty region will be identified. Low threshold has better performance, however accuracy will be sacrificed. On the contrary, if the threshold is high, more empty hyper-rectangle will be mined, but with finer granularity (i.e: size is smaller). More processing time will be needed, as the tree needs to be traversed deeper.

Tree traversal

The Range-tree is traversed top-down, where we start with a bigger range and restrict the range as we move down level of the tree. The tree traversal starts at level 1 and the finding of the suitable node is done by comparing $ratio_{A_i}$ and the user input parameter τ. If $ratio_{A_i} < \tau$, then we traverse down one level of the Range tree of attribute A_i to find the closer set intersection. Now the children of the node will be candidate itemset. On the other hand, if $ratio_{A_i} \geq \tau$, we do not traverse the tree.

$$ratio_{A_i} = \frac{|idlist_{new}|}{|idlist|} \tag{4.8}$$

The concept of tree traversal is illustrated using the example below with reference to Figure 4.4:

Example 4.8. Joining attributes A and B.

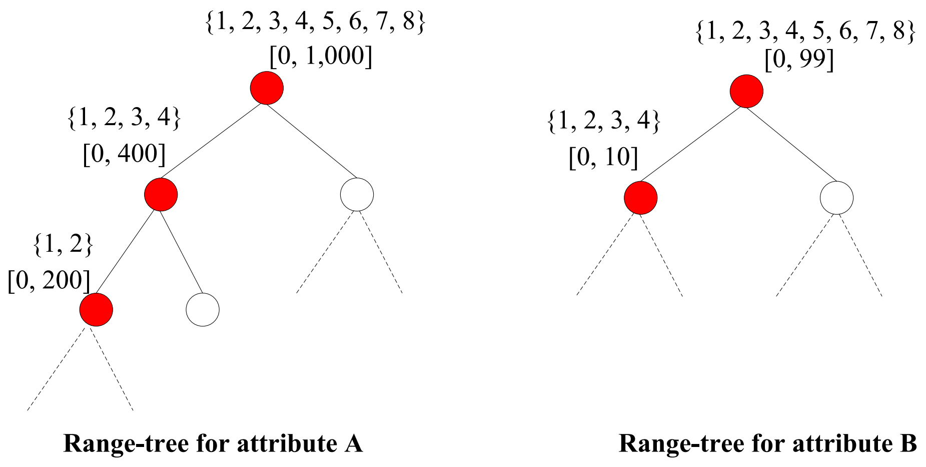


Figure 4.4: Example of top-down tree traversal

C_2: {A:[0, 400], B:[0, 20]}

$$\begin{aligned} idlist_{new} &= \{1,2,3,4\} \cap \{1,2\} \\ & \{1,2\} \\ ratio_A &= 0.5 \end{aligned}$$

1. Case 1: $\tau = 0.5$, $ratio_A = 0.5$.
   Since this satisfies the condition that $ratio_A \geq \tau$, therefore the new itemset is {A:[0,

400], B:[0, 20]}.

2. Case 2: $\tau = 0.75$, $ratio_A = 0.5$.
   $ratio_A < \tau$, therefore we need to traverse down the subtree of salary [0, 400]. The new candidate set is now {A:[0, 200], B:[0, 20]}.

$$\begin{aligned} idlist_{new} &= \{1,2\} \cap \{1,2\} \\ & \quad \{1,2\} \\ ratio_A &= 1.0 \end{aligned}$$

The condition of $ratio_A \geq \tau$ is satisfied, the new itemset is {A:[0, 200], B:[0, 20]}.

Mixture of discrete and continuous attributes:

The method discussed so far focused on mining **EHR**, however the method can be used to mine heterogeneous types of attributes. In the case of combination of a continuous and a discrete attribute, instead of traversing two tree, here we are only traversing only one tree. We use a top-down approach to find the combination of empty results. The tree will be traversed depth first based on user's threshold input, τ. The higher the threshold, the more fine-grain the results. Each branch will traverse in a depth-first search manner until either empty result is reached or count is above threshold. Here the **EHR** are represented as 'strips'.

For cases where it involves combination of only discrete attributes, the process of joining the attribute is straight forward, as it works just the same as itemset generation in Apriori algorithm.

To mine for empty combinations, the same technique still applies. **EHR** or empty combinations are candidates that have an empty set intersection in *idlist*.

4.4.4 Anti-monotonic Pruning

Anti-monotonic pruning, also known as positive pruning is used to reduce the number of combinations we have to consider. Our aim is to mine 1-frequent combination which has the anti-monotonicity property. If k-combination is not 1-frequent, then all $k+1$-combinations cannot be 1-frequent. Using this property, all superset of empty combinations will be pruned from the search space. Although pruned from the search space, these are the results that we are interested in, so they are output and stored on disk as results.

The concept of anti-monotonic pruning is illustrated in Figure 4.5. A_4 is found to be empty, and all supersets of A_4 are pruned away from the search space.

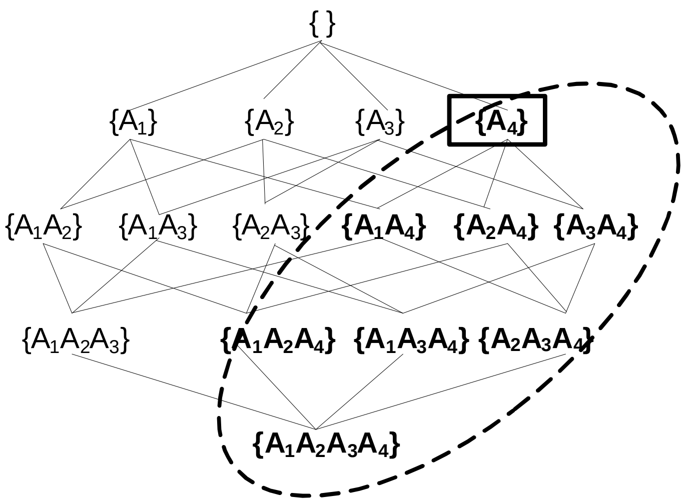


Figure 4.5: Anti-Monotone Pruning

4.5 Step 3 - Method 2:

Algorithm 4: Method 2

```
input  : $D_s$, the simplified dataset
output: $L_{k-empty}$, the minimal empty combination set
1  L$_n$ = $max\_set \setminus D_s$
2  Ł$_k$ = Ł$_n$
3  while ($k \geq 2$ OR Ł$_k \neq \emptyset$) do
4      C$_{k-1}$ = splitHyperspace($L_k$)
5      foreach (transaction $t \in D_s$) do                 /* scan dataset */
6          if for each $c \in c_k \subseteq t$ then          /* monotonic pruning */
7              delete c;
8      Ł$_{k-1}$ = {$c \in c_k$ where $c$ is not found in each transaction of $D_s$}
9      k–
10 $L_{empty} = \bigcup_k L_{k-empty}$
11 foreach $L_{k-empty} \in L_{empty}$ do
12     if isMinimal($L_{k-empty}$), $L_{empty}$ then /* eliminate non-minimal $L_{k-empty}$ */
13         output $L_{k-empty}$
14     else
15         delete $L_{k-empty}$ from $L_{empty}$
```

Contrary to method 1, method 2 uses the 'bottom-up' approach, both in the itemset lattice and in the Range Tree traversal. It starts with the most specific combination, and then find the more general combination as it moves up the lattice. Method 2 is chosen when the *distribution_size* > 0.5, when the data distribution is relatively dense. Therefore, empty hyper-rectangles are not likely to appear in low dimensional subspaces. We start our search at the bottom of the itemset lattice, with the combination of all the attributets. As a duality to method 1, in method 2 we use monotonic pruning. We start from L_n, with the full set of n-dimensions at the bottom. When L_k is found to be

```
Procedure isMinimal($L_{k-empty}$: k-empty itemset, $L_{empty}$: the full set of $L_{k-empty}$)
   input  : $L_{k-empty}$: $k$-empty itemset, $L_{empty}$: the full set of $L_{k-empty}$
   output: Boolean: TRUE, FALSE
 1 foreach $k-1$-subset to 2-subset $s$ of $L_{empty}$ do
 2     if $s \notin L_{empty}$ then
 3         return TRUE
 4 return FALSE
```

non-empty, we prune off this itemset. As we move levelwise upwards in the lattice, we generate candidate $k-1$ from existing L_k. Instead of joining subspaces to form a higher dimension subspace, now we start with high dimension subspace and split the itemset to obtain candidates of lower subspaces.

4.5.1 Generating n-itemset

n-itemset is the biggest or the most specific combination that returns an empty result. In Method 2, we start with n-itemset at the bottom level in the itemset lattice and work our way up levelwise upwards. Recall that in step 2, the simplified database, D_s is obtained. Consequently, with the available information, n-itemset can be obtained by doing the set complement.

$$n - itemset = max\_set \setminus D_s \tag{4.9}$$

4.5.2 Generating k-1 itemset

```
Procedure splitHyperspace(k-itemset: $L_k$)
   input  : $L_k$, k-itemset
   output: $c_{k-1}$, $k-1$-candidate itemset
 1 foreach (itemset $l \in L_k$) do                  /* $findCombination = \binom{k}{k-1}$ */
 2     $c_{subset}$ = findCombination(l, k-1)
 3     if for each $c \in c_{subset} \notin c_{k-1}$ then     /* keep unique candidate itemset */
 4         add c to $c_{k-1}$
 5 return $c_{k-1}$
```

Splitting adjacent rectangles

This is the reverse of joining adjacent rectangles in method 1. At every level of the of itemset lattice, the empty hyper-rectangles are split into smaller subspaces. Candidates of size $k-1$ is generated from k-itemset. At every level, the database will be scanned once to check whether the candidate itemsets are empty. If they are not, they will be pruned off from the search space. Candidate generation and dataset scan to evaluate the candidate itemset alternates until either the size of itemset reaches two, or no more candidates are generated.

Splitting of continuous attributes:

Tree traversal

Here we traverse the tree bottom up, starting with the smallest cluster at the leaf node and then generalize to a bigger one as we move up to the parent node. As we move up the lattice, we generate $k-1$-itemset and 'generalize' the ranges to a larger bounding range. At the bottom, we have the leaves of the tree, and as we generate the subset, we take values from the parents from the range tree. Since the Range-tree is a hierarchical tree, as many as B (branching factor)leaves share the same parent. As shown in Figure 4.6, as the 3-dimensional rectangle is split into 2-dimensional rectangle, we 'generalize' and expand the size of the region. To generate C_{k-1}, we traverse up the Range-tree of F levels. The value F_A for attribute A is calculated as follows:

$$F_A = \frac{tree - depth_A}{n-2} \tag{4.10}$$

where $tree - depth_A$ is the depth of Range-tree for attribute A and n is the number of dimensions. Each time a candidate C_k is generated from L_{k+1}, the range for C_k is obtained by traversing up the Range-tree by F levels. This is a faster option than to traverse the tree level by level, and this will correspond to the level of itemset lattice we have. Starting with bottom of the lattice, we start off with the leaf nodes in the Range-tree, which are the most specific ranges. As we traverse up the lattice by one level, correspondingly the Range-tree is traversed up F levels, providing more general ranges. When the lattice reaches candidate of size 2, we will also reach the general range in the tree. Consider Example 4.9 for the illustration of generating C_2 from L_3.

Example 4.9. Figure 4.7 show the respective Range-tree for attributes A, B and C. Candidate C_2 is generated by obtaining their respective ancestor's value, which is a more general range. Each range-tree is traversed up F-levels (calculated individually using equation 4.10).
As shown in the figure,

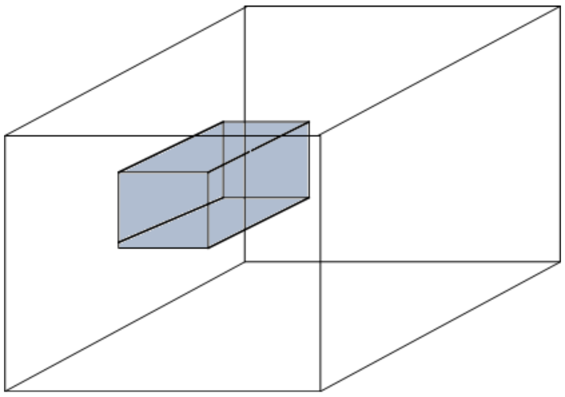
$$\begin{aligned} L_3 &= \{A:[0,5], B:[0.5,0.7], C:[10,15]\} \\ C_2 &= \{\{A:[0,10], B:[0.33,0.90]\}, \\ &\quad \{A:[0,10], C:[0,50]\}, \\ &\quad \{B:[0.33,0.90], C:[0,50]\}\} \end{aligned}$$

Mixture of discrete and continuous attributes:

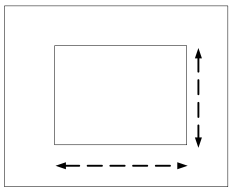
The method discussed so far focused on mining **EHR**, however the method can be used to mine heterogeneous types of attributes. In the case of combination of a continuous and a discrete attribute, instead of traversing two tree, here we are only traversing only one tree. We use a bottom-up approach to generate candidate sets. Here the **EHR** are represented as 'strips'.

For cases where it involves combination of only discrete attributes, the process of generating the candidate sets is straight forward, as it involves only getting the subset of the current itemset.

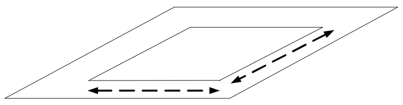
To mine for empty combinations, the same technique still applies, at each level the D_s will be scanned to eliminate non-empty candidates.



(a) Empty rectangle in 3 dimensional space



(b) Increasing the size of the region



(c) Increasing the size of the region

Figure 4.6: Splitting of adjacent rectangles

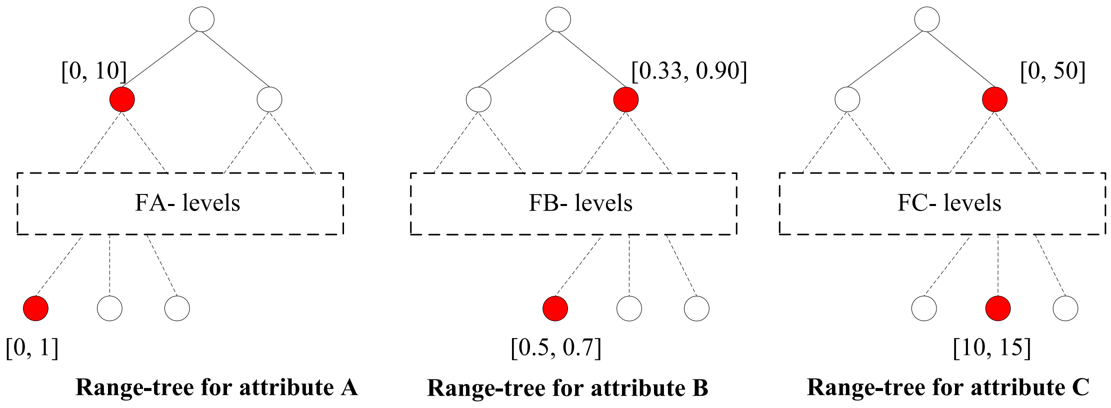


Figure 4.7: Example of generating C_2

4.5.3 Monotonic Pruning

Monotonic pruning is also known as the negative pruning, using it to prune off itemset that does not satisfy the property of being empty. Once the itemset reaches size 2, searching will be terminated. All mined **EHR** will be accumulated and tested for minimality. As our goal is to mine only keep the mininal empty combination, all non-minimal empty combinations are deleted. We know that if k-itemset is not empty, then all $k-1$-itemset is not empty either. As shown in Figure 4.8, $A_2A_3A_4$ is found to be non-empty, and all subsets of $A_2A_3A_4$ are pruned away.

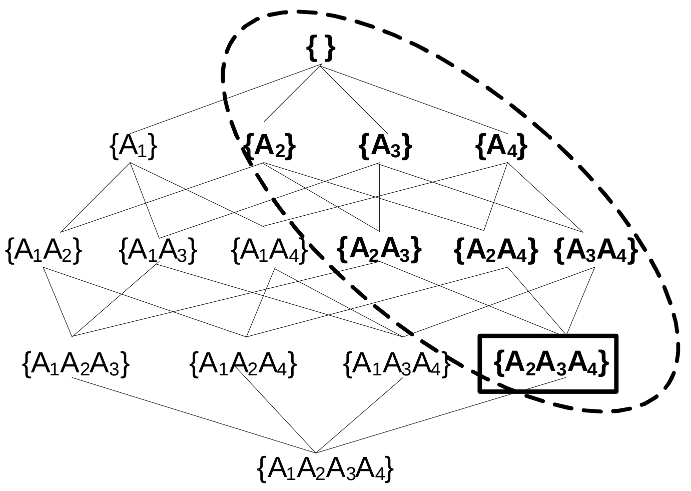


Figure 4.8: Monotone Pruning

4.6 Comparison between Method 1 & Method 2

Method 1 works well if the data distribution is sparse. It offers fast detection of minimal empty combinations. Since it uses the idea of storing and comparing the *idlist* associated to each element in the domain, a substantial amount of storage is needed. However, the advantage of this method is there is no need to scan the database to check whether the k-itemset exist in the database. Instead this can be done efficiently by using set manipulation on the TID, as it uses the concept of partitions, and to test whether k-itemset is empty, it is tested for set intersection. The main drawback of this method is that a big storage space is needed. Storages space is needed not only to store k-itemset, but we also store up the empty combinations that were pruned away from the search space.

Method 2 performs better if data is relatively dense. We start from the bottom of the itemset lattice and proceed levelwise upwards. Since the size of D_s covers more than half of max_set, we only work on the complement set of D_s, which is less than half the size of max_set. This will ensure that the size of the itemset is managable in terms of memory space. However, the disadvantage of this method is that it needs n number of database scans for n-dimensions. The main drawback of this method is that mining for minimal empty combinations is not easy. A lot of comparisons need to be made to prune of the non-minimal combinations.

In view of the strength and weaknesses of these two methods, it is proposed to use both method 1 and method 2 as a more robust method for mining **EHR** in different data distribution.

4.7 Data Structure

In this section, we describe the data structures used in the implementation of the algorithm.

1. Range tree: This tree structure is used to store the hierarchical clustering of the continuous attributes

2. Minimal empty combination: To efficiently mine these minimal combination, they are stored in hash table for fast comparison to trim off non-minimal empty combinations

3. Range tree traversal: Itemsets are stored in a stack. The range tree is traversed in a Depth-First Search manner to generate the candidate itemset. The traversal of the tree is determined by the user specified threshold τ. Once a subtree is traverse, the parent node is poped from the stack and children nodes are pushed into the stack.

4. An array to store bit fields: In order to speed up the comparison of two *idlist* to get the corresponding set intersection, an alternative implementation with fixed length bit fields can be used in place of the TID. The bits are set by hashing on the TID of the transaction. Intersection is accomplish by the logical AND of the corresponding bit fields.

5

Mining in Multiple Database Relations

In previous chapters, we have focused only in mining empty regions in a single table. In this chapter, we will explore generalizing the methods in the previous chapters for mining over a multi-relation database. Maximal empty joins represent the ranges of the two attributes for which the join is empty and such that they cannot be extended without making the join non-empty.

Relational databases are the most popular format for structured data, and is thus one of the riches sources of knowledge in the world. As most of the user queries are issued for multi-relational database, therefore it is only useful if we can mine empty result queries in multi-relational databases. However, data mining in multi-relation database is a long standing problem and is highly intractable. Therefore, currently data mining are mostly done in data represented in single 'flat' relations. It is counterproductive to simply convert multirelational data into a single flat table because such conversion may lead to the generation of a huge universal relation, making mining data difficult.

5.1 Holes in Joins

Currently no work has been done in mining empty region in multi-relational database. The closest is the work of Gryz and Liang, in 'Holes in Joins' [Gryz and Liang, 2006]. They consider mining empty regions in a join of two relations, more specifically the non-joining portions of two relations. In a join of two relations $R \bowtie S$, they are interested in finding empty region in the projection of $\Pi_{R.A,S.B}(R \bowtie S)$, where attributes A and B are *not* the join attributes. This method however, still does not fully achieve the purpose of mining empty regions in a multirelational database.

We identify that mining queries over multi-relation database is highly intractable. Nevertheless, we propose a method that can mine a group of low cardinality attributes from different relations. Method 1 described over a single relation can be slightly modified to be used in the context of a multi-relational database. Recall that in method 1, each attribute is partitioned and grouped into respective categories and a list of TIDs are assigned to each such group. By using tuple ID linkage as suggested in [Yin et al., 2004], we can virtually join relations efficiently and flexibly. Unlike in [Yin et al., 2004] where tuple ID linkage is used Classification problem, here it is used to conveniently to test for empty combinations. Multiple relations in a database are connected via entity-relationship links of ER models in a database design. Among the many relations, we identify a target relation R_t, whose tuples are called *target tuples.* It is usually the fact table or the transaction table. With the use of TID, empty result queries formed by attributes of different relations can be mined independently, as in a single table. Without joining the tables, we are able to do mining by using the idea of *partitions* formed for each different relations.

We focus only on one-to-many relationships, specifically the star schema. An *N-dimensional star schema* consist of a fact table and N dimensional tables, D_N. Every dimension table D_i has a primary key, K_i for $i = \{1, 2, \dots N\}$. Each dimension has a reference key in the fact table where each K_i forms a *foreign key* in the fact table. The main idea here is to assign each tuple in D_N with a TID, and then these TID are propagated back to the respective dimension table, D_i.

The algorithm of mining in a multi-relation database is as follows::

1. Simplify and encode the fact table with respect to the distinct foreign keys involed.
2. Create partitions on each individual foreign key.
3. Propagate the TID back to the dimension table.
4. Proceed as method 1.

The algorithm is presented and illustrated in the following example:

Example 5.1. Figure 5.1 shows a simple star schema, with one fact table and two dimension tables. In step 1, the fact table *simplified* and is encoded into a smaller table, as

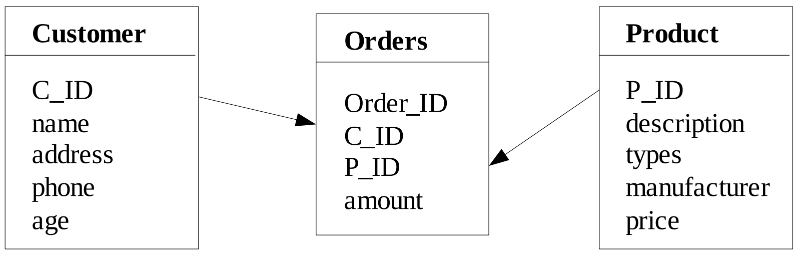


Figure 5.1: Star schema

shown in the tables below. Table 5.1(a) shows the original transactions for table **Orders**, while table 5.1(b) is a simplified version with respect to C_ID and P_ID. C_ID and P_ID are the primary key for **Customer** and **Product** table respectively. Each tuple in the simplified Orders are now distinct in terms of C_ID and P_ID. A distinct TID is assigned to each such tuples.

In step 3, TIDs are then propagated to the dimension tables, as shown in Table 5.2(a) and 5.2(b). Mining for empty queries between these tables can be done using method 1 described in Section 4.4 in Chapter 4.

Orders			
Order_ID	C_ID	P_ID	amount
O1	0100	p01	500
O2	0200	p01	1000
O3	0300	p02	2000
O4	0100	p01	800
O5	0300	p02	2000
O6	0200	p02	1500

(a) Original table

Orders			
TID	Order_ID	C_ID	P_ID
1	O1, O4	0100	p01
2	O2	0200	p01
3	O3, O5	0300	p02
4	O6	0200	p02

(b) Simplified table

Table 5.1: Fact table: **Orders**

Customer				
C_ID	name	...	age	*TID*
0100	Petersen		20	1
0200	Schmitt		25	2,4
0300	Victorino		35	3

(a) Dimension table: **Customer**

Product				
P_ID	description	type	...	*TID*
0100	Chevy	Trucks		1,2
0200	XREO	Motorcycles		3,4

(b) Dimension table: **Product**

Table 5.2: Tuple ID propagation

5.2 Unjoinable parts

In the previous section, we searched for holes that appear in joins, in that two relations are joinable, but their non-joinable attributes are empty. In this section, we focus our attention on non-joinable part of two relations. Another reason that causes a multi-relation query to return an empty result is due the fact that given the selection criteria, the relations are simply non-joinable. In the context of a star schema, there may be parts where the values from a dimension that does not exist in the fact table. After tuple ID is being propagated back into the dimension tables, a tuple that does not exist in the fact table will have an empty value for the TID column. Consider the example below:

Example 5.2. As shown in Table 5.3, TID value is blank for the tuple: $\langle 0300, \ldots, Car \rangle$. This indicates that this particular tuple does not appear in the fact table *Orders* and is not joinable. These *non-joinable* parts can be seen as tuples that appear only in an outer join between the *Product* and *Orders* but not in a natural join between between the two tables.

Product				
P_ID	description	type	...	*TID*
0100	Chevy	Trucks		1,2
0200	XREO	Motorcycles		3,4
0300	XDKEJ	Car		-

Table 5.3: dimension table: **Product**

Based on the above illustration, we now introduce the definition of the target domain.

Definition 5.1. Target domain. Target domain, *target_domain* is the domain of an attribute based on the values that appear in the target table.

$$target\_dom(A_i) = t_{[i]}, t \in target\ table.$$

In a star schema, the target table is the fact table. The extended definition of an empty value in the context of a star schema is:

$$Empty\ values : a_i \in dom(A_i), a_i \notin target\_dom(A_i).$$

Example 5.3. Based on Example 5.2, the empty value for attribute *type* with respect to the fact table *Orders* is 'Car'. The value 'Car' belongs to *dom*(*type*) for dimension table *Product*, but does not exist in $target\_dom$(*type*). Any query to the fact table that has the selection criteria $type =' Car'$ evaluates to empty.

6 Test and Evaluation

In this section, we describe the experiments and the evaluation of the performance of the proposed solution. To get a more accurate evaluation, we evaluated the algorithm using both synthetic and real life datasets. They were obtained from WEKA machine learning homepage [Witten and Frank, 2005]. Larger datasets were chosen to demonstrate the scalability of the proposed solution. A subset of all the attributes in each dataset were selected, and the selection were based on the following criteria:

- discrete attributes: low cardinality (with cardinality below 10)
- continuous attributes: unrestricted range but different value distribution were chosen to demonstrate the mining of **EHR**.

More description about each individual test can be found in the following sections.

Besides that, we have conducted tests based on other factors, such as using different values for threshold τ, varying distribution size and using the different combination of data types. In Chapter 4, we have focused mainly on mining **EHR**, but it is also shown that the method can be used mine heterogeneous itemsets. All mined results are presented in the form of rules. For rules that involves continuous attributes, they can be presented in the form of empty rectangles. The visualization of some of the results in the form of empty rectangles in two dimensional space will be presented in the result section.

Experimental Setup

The tests were ran on a personal computer with 1.6GHz processor, 1GB main memory and Pentium 4 PC with Windows 2000. The proposed algorithm is implemented in Java. It is assumed that for all test cases, the simplified database D_s fits into the main memory.

6.1 Performance on Synthetic Datasets

TPCH

The TPC Benchmark H (TPC-H) is a decision support benchmark and it is a publicly available workload, which allows for replication and for experiments. The attributes were chosen from two tables, namely *orders* and *lineitems*. This testset is stored in a DBMS. The join of these two tables has the size of 6,000,000 tuples and the initial dataset does not fit into the main memory. For the simplicity of the testing, the simplified database D_s

with respect to the chosen attributes were generated directly from the DBMS. Table 6.1(b) shows the result based on the different sets of attributes. Since we started the program execution directly with D_s, the processing time taken is much lower.

Table 6.1(a) shows the set of attributes selected while Table 6.1(b) and Figure 6.1 show the results with respect to different sets of attributes.

Description	Type	Domain Cardinality	Min	Max
l_shipmode	Low-card.	7		
l_returnflag	Low-card.	3		
l_linestatue	low-card.	2		
o_orderstatus	low-card.	3		
o_priority	low-card.	5		
l_discount	Numeric	11 clusters	0	0.1

(a) Selected attributes

Dimension	*distribution_size*	# rules	time (in seconds)
first 2 attributes	1	-	-
first 3 attributes	0.67	2	20
first 4 attributes	0.44	5	45
first 5 attributes	0.44	5	77
all attributes	0.44	9	704

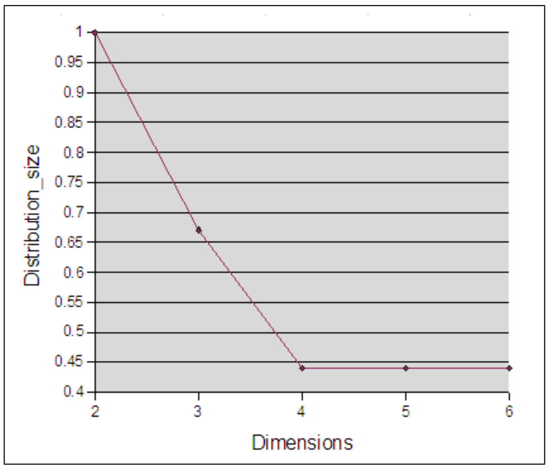
(b) Results

Table 6.1: TPCH

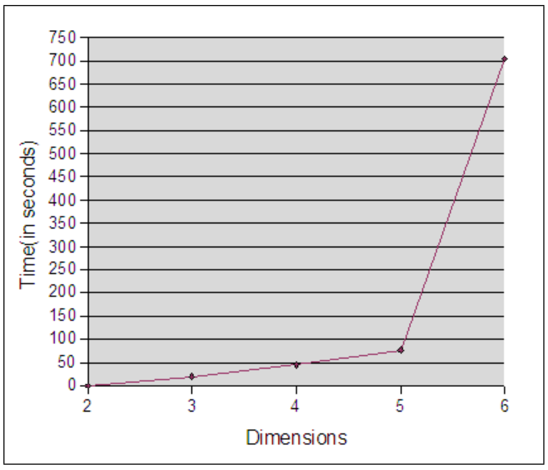
Other synthetic datasets

Detailed description about each test can be found in Appendix A.

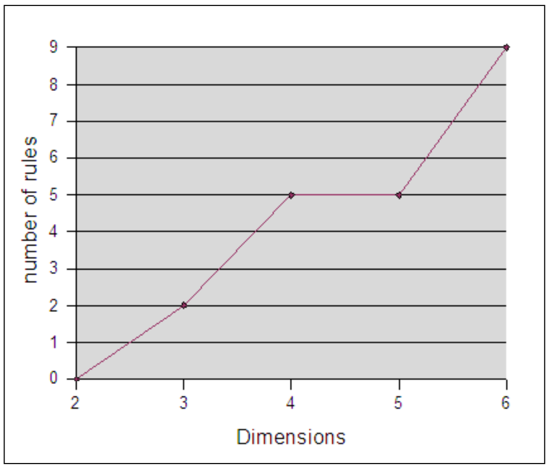
- Testset #1
- Testset #2



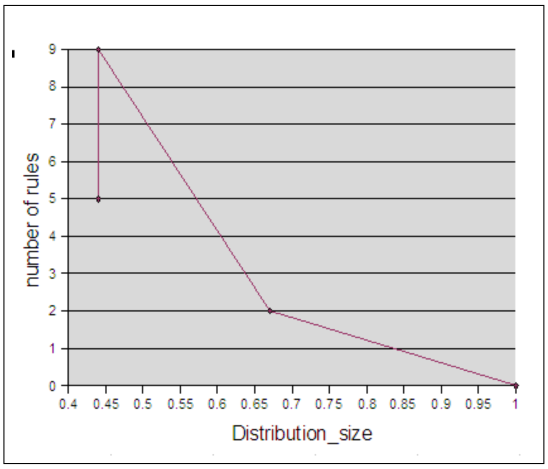
(a) Distribution size versus Number of Dimensions



(b) Time versus Number of Dimensions



(c) Number of Rules versus Number of Dimensions



(d) Number of Rules versus Distribution size

Figure 6.1: TPCH: Result charts

6.2 Performance on Real Life Datasets

Table 6.2 below is the testset for California Housing Survey, with the size of 20,640 tuples, obtained from the StatLib repository (http://lib.stat.cmu.edu/datasets/). It contains information on housing survey on all the block groups in California from the 1990 Census.

Description	Type	Domain Cardinality	min	max
Median Age	Numeric	52 clusters	1	52
Total # rooms	Numeric	56 clusters	2	39320
total # bedrooms	Numeric	59 clusters	1	6445
Household	Numeric	65 clusters	1	6,082
Median Income	Numeric	76 clusters	0.5	15
Median House Value	Numeric	86 clusters	14,999	500,001

(a) Selected attributes

Dimension	*distribution_size*	# rules	time (in seconds)
first 2 attributes	0.53	106	218
first 3 attributes	0.38	389	532
first 4 attributes	0.2	1,825	645
first 5 attributes	0.17	2,180	3268
all attributes	0.08	6,686	4,842

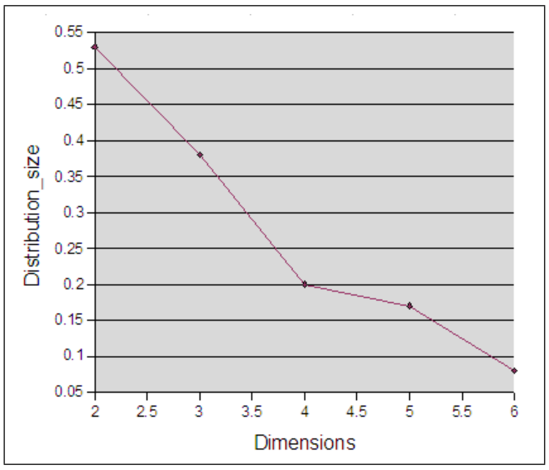
(b) Results

Table 6.2: California House Survey

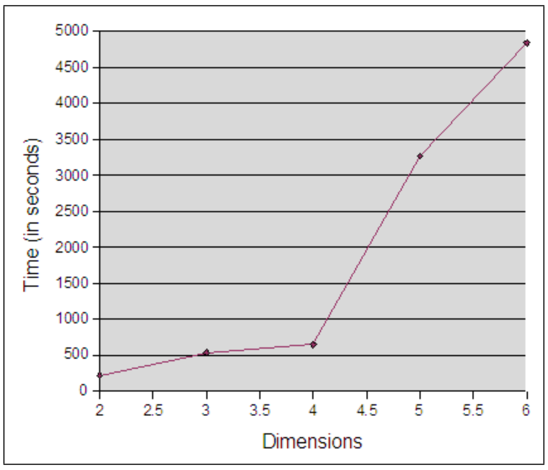
Other real life datasets

Detailed description about each test can be found in Appendix A.

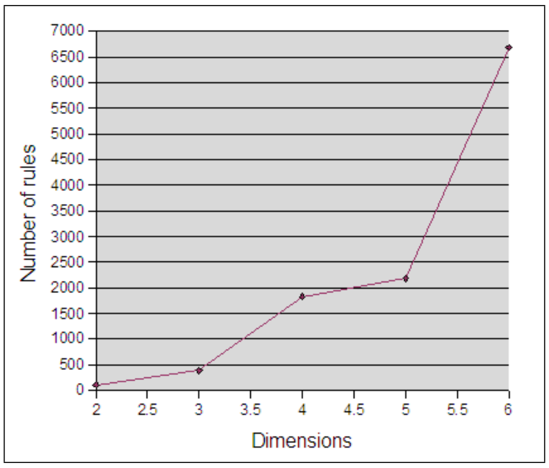
- Ailerons
- Credit Card Approval
- KDD Internet usage survey



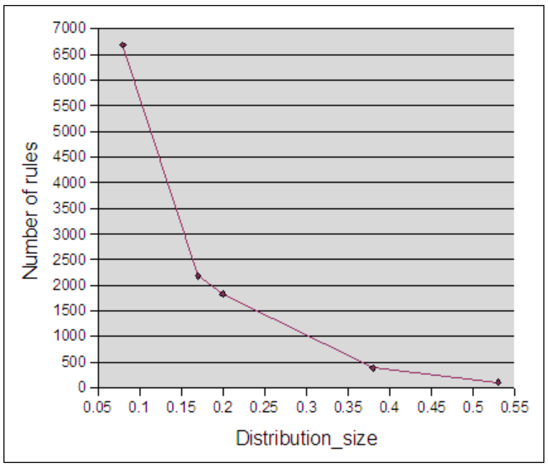
(a) Distribution size versus Number of Dimensions



(b) Time versus Number of Dimensions



(c) Number of Rules versus Number of Dimensions

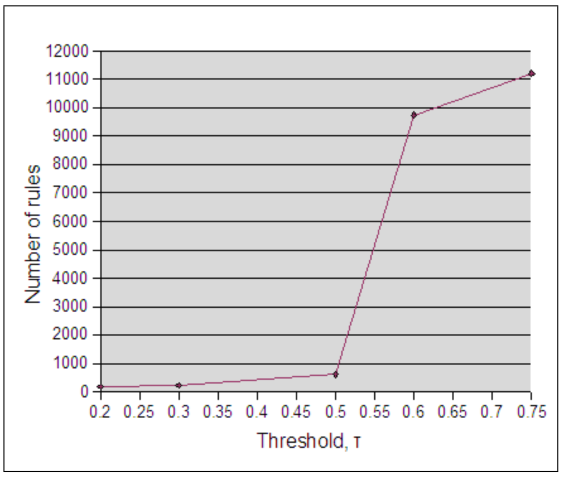


(d) Number of Rules versus Distribution size

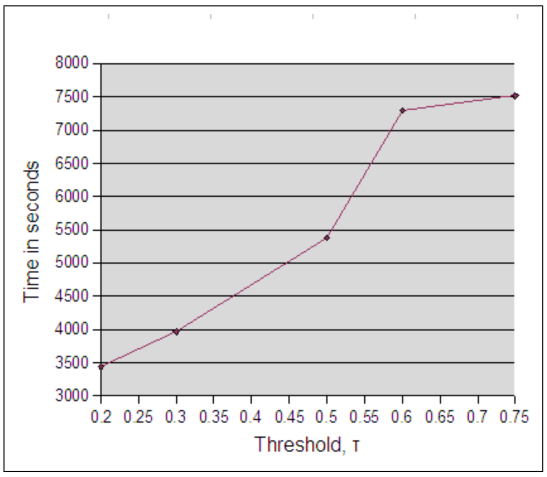
Figure 6.2: California House Survey: Result charts

6.3 Granularity

As described in Section 4.1.1 in Chapter 4, users can determine the granularity of the result, by assigning an appropriate value to the threshold parameter τ. Tests have been conducted based on varying threshold value over Testset #1 (refer to Appendix A). If τ is small, then bigger 'chunks' of empty regions are considered at one time. Bigger τ, on the other hand, splits the empty spaces into small 'chunks' for processing. Figure 6.3(a) shows the number of mined rules with respect of the different threshold value, while Figure 6.3(b) shows the processing time needed for each different threshold. Figure 6.4 shows the visualization the different size and the different number of the rectangles based on the value of τ.



(a) Number of rules versus Threshold τ

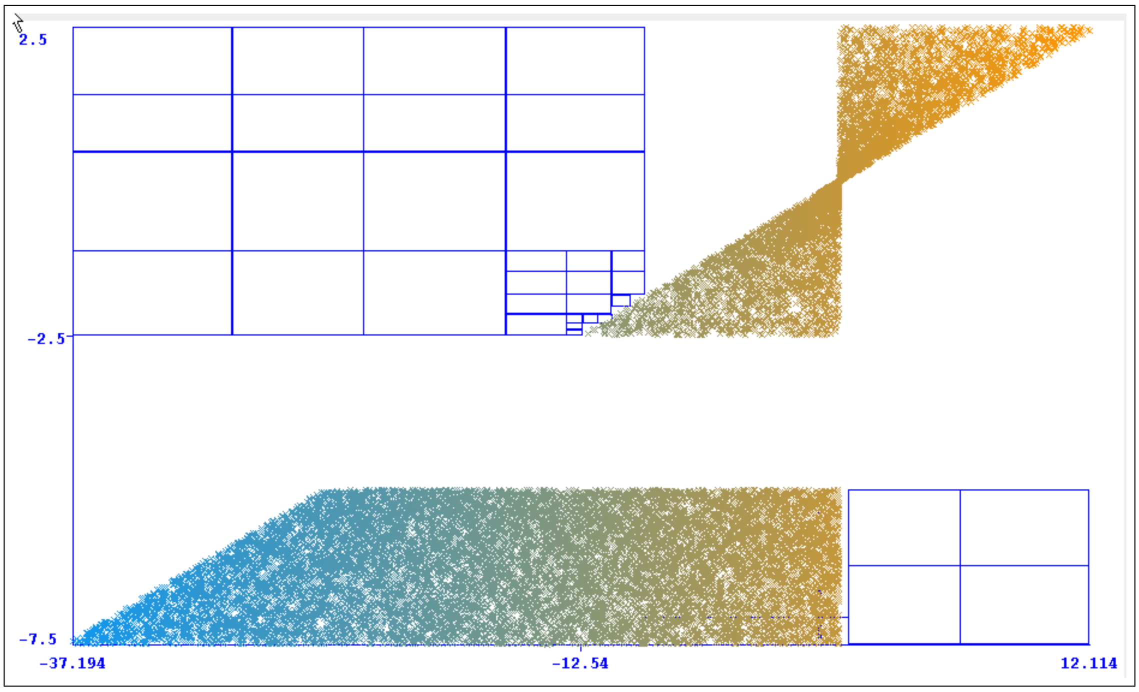


(b) Time versus Threshold τ

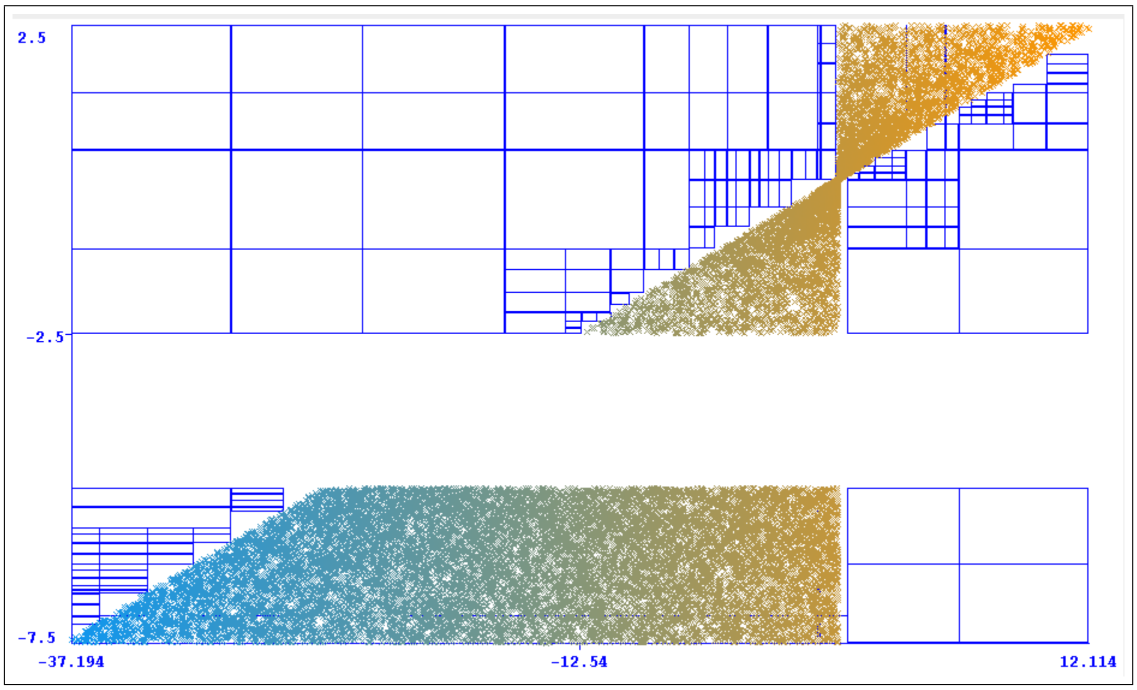
Figure 6.3: Results for varying threshold, τ

Analysis

Higher value of τ produces higher number of finer grain (smaller size) of rectangles. Based on the charts in Figure 6.3, processing time increases proportionally with the value of τ. This is because with a higher τ, Range-trees are traversed deeper and more rules are compared and generated.



(a) τ = 0.5, x-axis: X6, y-axis: X4



(b) τ = 0.75, x-axis: X6, y-axis: X4

Figure 6.4: Mined **EHR** based on different threshold, τ

6.4 Varying Distribution Size

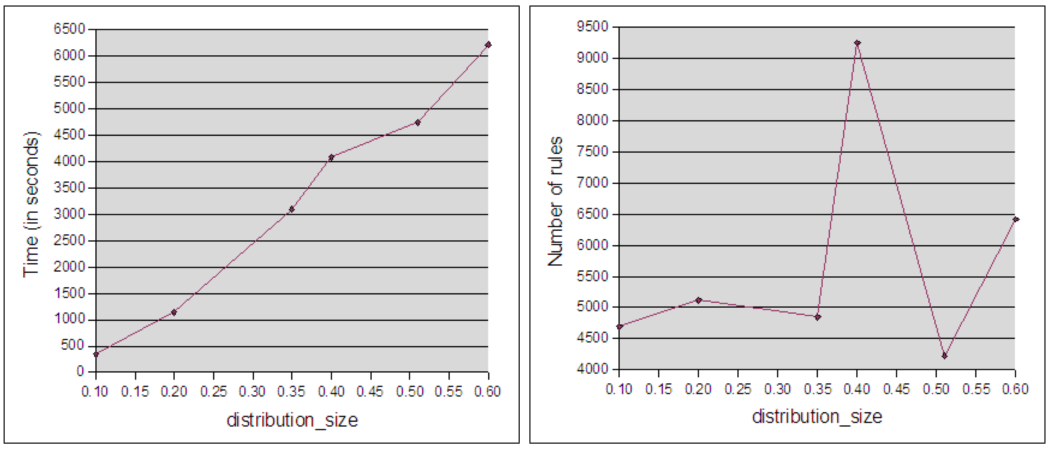
Recall that in Section 4.3 in Chapter 4, data distribution is measured in terms of *distribution_size*. The choice of running method 1 or method 2 is determined by the *distribution_size*. A synthetic dataset was generated based on a fixed set of attributes, but with varying distribution size. The list of attributes are listed in Table 6.3.

Description	Type	Domain Cardinality
att1	low-card.	5
att2	low-card.	5
att3	numeric	25 clusters
att4	numeric	15 clusters
att5	low-card.	5

Table 6.3: Synthetic datasets

Analysis

The result for different *distribution_size* is displayed in Figure 6.5. From the results shown in Figure 6.5(a), we are able to draw the conclusion that the time is proportional to *distribution_size*. As for the number of mined rules, it does not depend on data distribution, as displayed in Figure 6.5(b).



(a) Time versus Distribution_Size

(b) Number of Rules versus Distribution_Size

Figure 6.5: Results for varying *distribution_size* over a fixed set of attributes

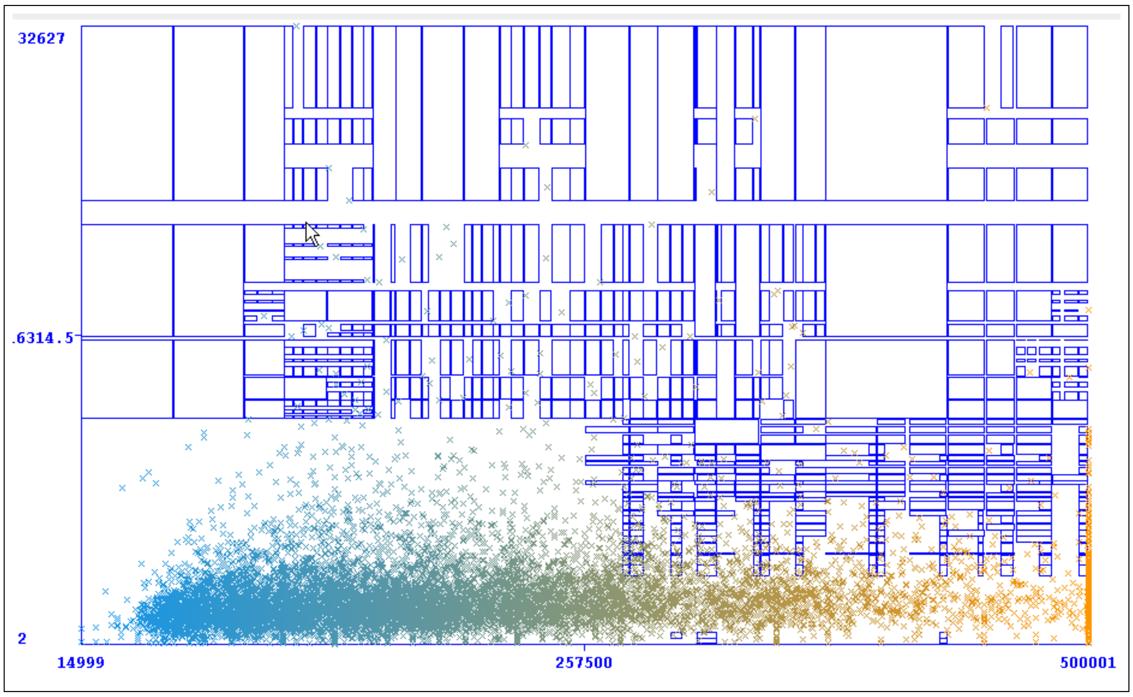
6.5 Different Data Types

These test datasets are chosen to show the working for the algorithm in the case of homogeneous and heterogeneous data types (a mixture of categorical and numerical values). For categorical values, they are of low cardinality domain, while for numeric values, they have different ranges and different size of ranges. The results show that the proposed solution is able to mine efficiently on different data types. The testsets chosen can be categorized into the following groups:

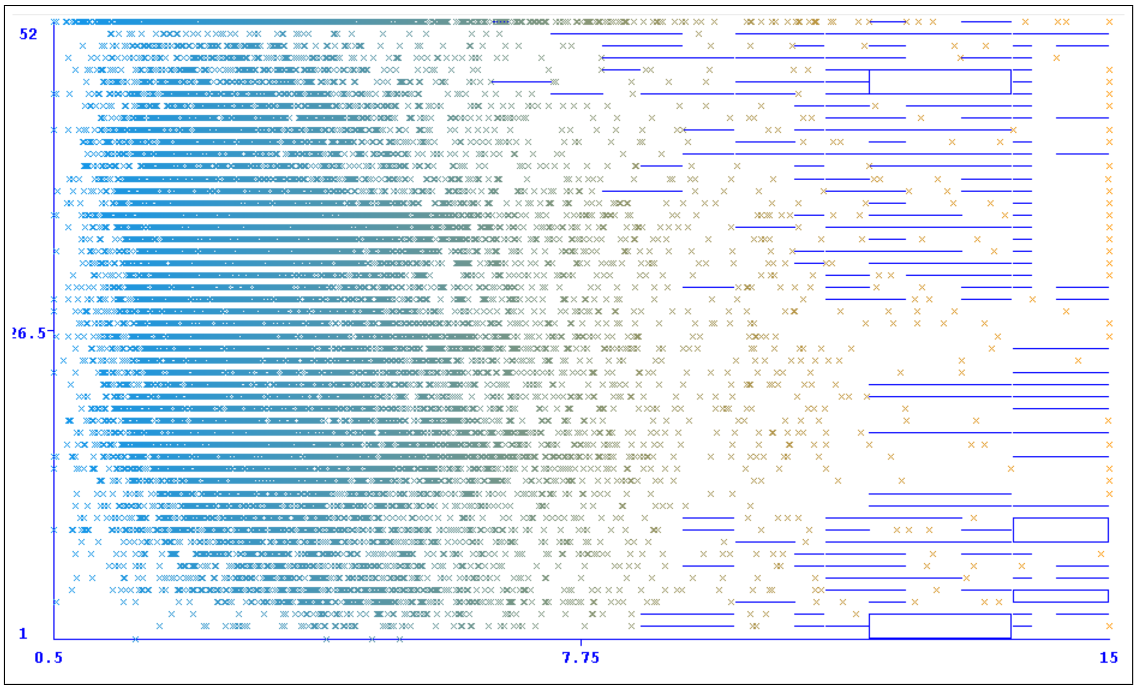
- Low cardinality discrete attributes:
  KDD Internet Usage Survey.
- Unbounded continuous attributes:
  California Housing Survey, Ailerons, Testset #2 .
- Mixed attributes:
  TPCH, Testset #1, Credit Card Scoring.

6.6 Accuracy

In this section, we investigate the accuracy of the mined rules. Figures 6.6, 6.7 and 6.8 shows empty rectangles identified in 2 dimensional space. For simple illustration purpose, we only plot rectangles in 2 dimensional space. Higher dimension results are also generated, but visualizing them is difficult.



(a) (x-axis: median house value, y-axis: total # rooms)



(b) (x-axis: Median Age, y-axis: Median Income)

Figure 6.6: Point distribution and EHR [California Housing Survey]

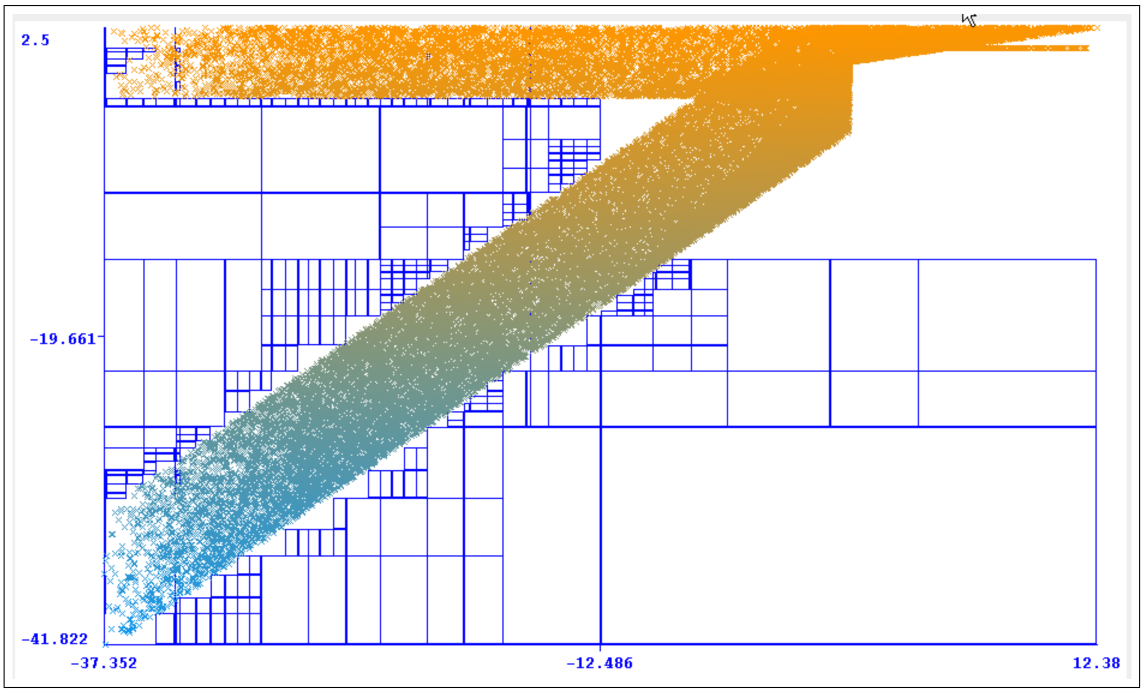


Figure 6.7: Point distribution and EHR [Testset #1 (x-axis: X6, y-axis: Y)]

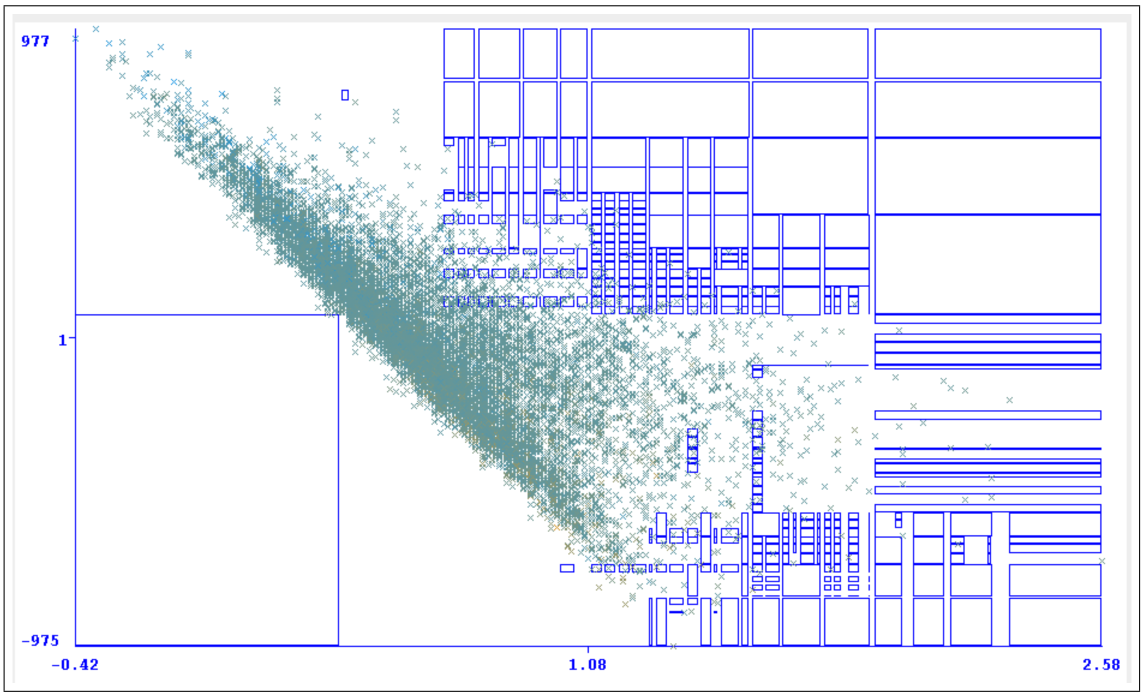


Figure 6.8: Point distribution and EHR [Ailerons (x-axis: curPitch, y-axis: climbRate)]

6.7 Performance Analysis and Summary

In this method, false positives are not allowed while false negatives are tolerable. That is, all mined rules are guaranteed to be empty, but there may be empty regions in the dataset which are not mined. This is due to two factors:

1. User defined coverage, τ. This parameter determines the granularity of the mined hyper-rectangles. The bigger the value of τ, the more accurate the results will be. If the value of τ is small, lesser rules are obtained and the accuracy of the mined result will be sacrificed.

2. Clusters are defined to be as small as possible, but since it is not feasible to have a cluster for each and every individual point, there will be empty region within the cluster itself. In this case, we treat a cluster as a totally filled region, and therefore misses the empty spaces within the cluster. However, since the points in a cluster falls within the interval threshold T, the empty spaces within the cluster will have the size of less than T. Therefore these regions are small enough to be ignored.

6.7.1 Comparison to existing methods

Since the approach of this method is different from any existing method, there is no direct and clear cut comparison to any of the existing methods. Nevertheless, comparisons can still be made in terms of strengths and limitations. The existing methods mentioned here are with reference to ones discussed in detail in Chapter 2.

In the method outlined in [Edmonds et al., 2001], uses the concept of growing empty rectangles. They depict the dataset as an $|X| \times |Y|$ matrix of 0's and 1's. To obtain the largest possible rectangles or maximal rectangles, the data has to be sorted first, so that all 0's are in the adjacent position. Many smaller isolated holes are not taken into consideration. In our case, isolated empty regions can be mined without sorting the data points first. With just one pass of the database, the Range-trees can be constructed. As shown in Section 6.6, it is evident that the algorithm is able to detect empty spaces of different shapes and sizes.

From the test results, we demonstrated that the algorithm works well for both discrete and continuous attributes. Both methods that discover holes by geometry ([Edmonds et al., 2001] and [Liu et al., 1997])has the limitation of working only on continuous values. The method using Decision Tree induction [Liu et al., 1998] is able to cater for both discrete and continuous attributes, but focuses only on the latter.

All existing methods (except [Luo, 2006]) focus only finding empty rectangles in two dimensional spaces, and they do not perform well in higher dimensional spaces because of high time and space complexity. In this case, we have shown that this method scales well in higher dimension spaces. Here tests and experiments of 4 to 7 dimensions were conducted. The method can be accommodated to mine higher dimensional by storing and writing data on disk as suggested in Section 6.7.3.

6.7.2 Storage and Scalability

This method scales well to large database as the algorithm runs independently on the size of the original dataset, but dependent only on *max_set*, which is determined by the domains of the set of chosen attributes. The description and calculation of *max_set* is found in 4.1.3 in Chapter 4. First we do a simplification encoding of the database, to obtain D_s. The size of the simplified database D_s has the upper bound of the size of *max_set*. In most cases, $|D_s|$ is much smaller than $|max\_set|$ unless the dataset is densely populated and evenly spread out.

Both method 1 an 2 uses the levelwise search, they can be analaysed as follows: In the worst case analysis, the time and space complexity depends on the number of sets in the levels L_l, called the size of a level. Let n be the number of dimensions and s be the sum of the sizes of the levels. In the worst case, $s = O(2^n)$.

During the generation of C_k, we need storage for L_{k-1} and the candidates C_k (In the case of method 2, for the generation of C_k, we need storage for L_{k+1}). Memory needs to be managed carefully as the candidates generation phase may occupy and take up all the available buffer. When the *distribution_size* is large, candidate generation may run into the problem of insufficient main memory. To overcome this problem, method 2 is proposed as an alternative to deal with datasets with large *distribution_size*.

The two main differences between method 1 and 2 are outlined as follows:

- Disk Access: In method 1, it needs $O(|D_s|)$ access, while for method 2, it needs $O(s)$ accesses of size $O(|D_s|)$.
- Main Memory Requirement: Method 1 requires higher storage space, because besides storing the L_k and C_{k+1}, extra storage space are needed for *idlist*. In the worst case, the space complexity is $O(2^n\,|idlist|)$. As for method 2, it needs storage space of L_k and C_{k-1}. In the worst case, the space complexity is $O(2^n)$.

6.7.3 Buffer Management

The method employed here is independent of the dataset size, but with a upper bound of the maximal set, *max_set* (the cross product of the attribute domains.) Therefore this algorithm works well for attributes with low cardinality domains.

Method 1:

In the candidate generation phase of pass k, we need storage for itemset L_{k-1} (including their TID)and the candidate itemset C_k. If L_{k-1} does not fit into the memory, we externally sort L_{k-1}. We bring into memory a block of L_{k-1} in which the first $k-2$ items are the same. We now generate candidates using this block. We are interested in the pruned off itemsets, those itemset where their combination of L_{k-1} does not have any common TID and produces an empty set intersection. They are paged out onto disk as results.

Method 2:

We start off with L_n, with n dimensions. In the candidate generation phase of pass k, we need storage for itemsets L_k and the candidate C_{k-1}. In the dataset scanning phase, we need storage for C_k and at least one page to buffer the database transaction. L_k are stored and paged out onto disk if memory is insufficient. Testing for minimality and pruning will be done once the levelwise algorithm is finished. An efficient test for minimality can be done using a hashtable.

6.7.4 Time Performance

Time performance in each step of the methods are analyzed as follows:

Step 1: Data Preprocessing

In the data preprocessing phase, values in continuous attributes are clustered. Let m be the number of continuous attributes. In the data preparation phase, the algorithm needs to scan through the original dataset, D with $|D|$ tuples. Each data tuple may travel at most $log_B\,|D|$ steps down a Range-tree, where B is the branching factor of an internal node in the Range tree. All the above operations are duplicated for each of the m dimensions giving a time complexity of $O(m\,|D|\,(log_B\,|D|))$.

Step 2: Simplify Database

In this step, we simplify the database into a smaller dataset D_s. We need to access the original dataset, the time complexity is linear to the number of tuples in the original dataset, which is $O(|D|)$. The process of simplifying the database is done using the hash table. Each unique tuples is stored in a hashtable, while non-unique tuples are ignored. The process of checking whether a tuple is unique can be done in constant time by checking for the entries in the hashtable.

Step 3: Method 1

In method 1, candidates C_k are created by testing for non-empty set intersection *idlist* of L_{k-1}. The checking of set intersection between two *idlist* is polynomial. Let l_1 be $|idlist_1|$ and l_2 be $|idlist_2|$, the time complexity is $O(l_1 l_2)$. Since we store the TID in main memory, there is no need to scan the database for each level, thus it is faster. However the downside of this method is that it incurs a higher storage space.

Step 3: Method 2

For this, we need to scan the database for every level in the lattice, hence the access time is slow. One of the main advantages is the linear dependency of the algorithm on the number of rows in the relation and linear to the size of the number of attributes. After that, all non-minimal empty combinations are eliminated. Method 2 does not incur high storage space, but requires a longer run time.

6.7.5 Summary

The method is at its best when the number of attribute in an $Ł_{empty}$ are relatively small. When the size of the empty rules is roughly one half of the number of attributes, the number of itemset is exponential in the number of attributes. When the attribute combinations are larger than that, the levelwise method that starts the search from small dependencies obviously is further from the optimum. In this situation, we apply method 2, which starts from the largest combination and work our way to smaller combinations.

Here is the observation made, the *distribution_size* is inversely proportional to the number of dimension. It is found that *distribution_size* decreases as more attributes are being considered. This is consistent with the observation made in [Agrawal et al., 1998], where it is stated that the average density of points in a high dimensional space is likely to be quite low. Therefore, they propose first to mine dense region in lower subspace, before proceeding higher dimensional spaces. From all the expriments and the corresponding results, it is found that *distribution_size* is generally quite small when it involves continuous attributes. Consequently in most cases, method 1 is used.

In view of the time and space complexity of both method 1 and 2, it is therefore justifiable to use both methods as a more robust method for mining **EHR** in different data distribution. By taking into consideration of the data distribution, we can find a suitable method that will ensure a method that has an efficient runtime and storage space.

6.7.6 Practical Analysis

Due to pruning, s can be significantly smaller than the worst case analysis. In the candidate generation phase, L_k are stored in hashtable, making pruning of C_{k+1} at constant time. For the elimination of non-minimal empty combination, mined rules are stored in hashtables. Therefore, in practice, the retrieval and pruning of rules takes constant time.

6.7.7 Drawbacks and Limitations

One of the drawbacks of the algorithm is that for some cases, a large number of rules are produced. In cases like these, it will simply be counter-productive if the number of rules are almost equal or close to the size of the dataset. This is because a large amount of time has to be devoted to check through the rules to determine whether any given query is empty. Besides that, a large storage space is needed to store the rules. In such situation, mining and checking for empty result queries will incur a high computational cost.

The size of the rectangles depends on the types of data distribution. Clusters for each continuous attributes are formed according to the positions of points, where points located in close proximity to each other forms a cluster. For points that are highly skewed, bigger sized rectangles can be identified (refer Figure 6.8, 6.7), whereas for points that are evenly distributed, clusters of equal size and are located close to each other are produced. In fact in such cases, no real clusters are formed since the data points are evenly scattered. (eg: Figure 6.6(b)). In this case, rectangles are that small or insignificant are produced. This can be explained using the concept used in [Liu et al., 1997]. A maximal hyper-rectangle is split when a data point is added. The existing rectangles are repeatedly being split into smaller rectangles whenever a new data point is added. If the point distribution is evenly scattered, then many small rectangles will be produced. Analogous in our situation, mined rules produced have a very fine granularity and are not useful in empty query detection. To overcome this problem, [Liu et al., 1997] proposed to mine only sufficiently large empty regions.

7

Integration of the algorithms into query processing

In this chapter, we explore the potential usage of mined **EHR**. First and foremost, we can use it to detect empty result queries. This is done by matching the selection criteria with the mined rules. Once any selection criteria is found to be an exact match or a superset of any empty rules, then that particular query is categorized as 'empty' and thus will not be evaluated. All **EHR** can be materialized as views, as described in [Gryz and Liang, 2006]. Another alternative is to store them in the form of rules.

7.1 Different forms of EHR

As explained earlier, mining of empty result queries can be done in the form of finding empty spaces in datasets or selection rules. The goal is to mine the minimum empty query combination $L_{k-empty}$, and any superset of an empty selection will not be stored.

Mined rules.

Mined rules are stored in a Disjunctive Normal Form (DNF). Queries that matches one of the clauses will be categorized as empty. The rules are stored in the following form:

$$L_{empty} : \{rule_1 \vee rule_2 \vee \dots rule_m\}$$

where $rule_i$ for $i \in \{1, \dots m\}$ is in the form of Conjunctive Normal Form(CNF), consisting of

- point based comparison (for discrete and continuous attributes): $A_x = a_x$
- interval comparison (for continuous attributes): $min_x \leq A_x \geq max_x$

where $A_x \in A$, and $x \in \{1, \dots, n\}$.

Example 7.1. Let D be a database with attributes A = $\{A_1, A_2, A_3\}$, where A_1, A_2 are continuous valued attributes and A_3 is a discrete valued attribute, with the domain of $\{a_{3_1}, a_{3_2}, a_{3_3}\}$.
rule1: $5 < A_1 < 10$
rule2: $2 < A_2 < 14 \wedge A_3 = a_{3_3}$
rule3: $13 < A_1 > 20 \wedge 15 < A_2 < 20 \wedge A_3 = a_{3_1}$

The mined rules are presented in this form:

$$\begin{aligned} L_{empty} &= (5 < A_1 < 10) \\ &\vee\ (2 < A_2 < 14 \wedge A_3 = a_{3_3}) \\ &\vee\ (13 < A_1 > 20 \wedge 15 < A_2 < 20 \wedge A_3 = a_{3_1}) \end{aligned}$$

Materialized Views

As for finding empty regions in the data, it is analogous to finding empty hyper-rectangles. The mined results can be materialized as views. When a materialized view for a set of attributes X is constructed, all empty rules of set X and all rules involving subset of X are combined to form the view. In the visualized form, all **EHR** formed by the subset of X, which are in lower dimension are projected to X dimensions.

Both rules and materialized views can be used in detecting empty result queries. Mined rules can be used in the form of checking against the selection criteria of a query. If the query selection matches any of the conjunctive clauses, then it is categorized as 'empty'. **EHR** can be materialized as views. Any query that falls within the empty region defined by the **EHR** is categorized as 'empty'.

7.2 Avoid execution of empty result queries

Our main focus here is to detect these empty result queries and to avoid executing them. A possible implementation of empty query detection can be done for 'canned queries'. 'Canned queries' are predefined queries, with predefined selection predicates. In most instances, canned queries contain prompts that allows user to select the query parameters from a list. In these cases, only the selection values in the query changes. Typically, these kind of queries are common for end users in the Internet domain. Online users often submit canned queries by filling parameter values into a predefined template. This usually involves choosing a few predefined lists, each having a low cardinality domain. An example would be the Advanced Online Search, that has more selection criteria than a general or basic search.

We propose to use mined **EHR** in this context. Since the method that we propose efficiently mines low cardinality domains, we can use the result in detecting empty canned queries. We output **EHR** in the form of rules and populate them in the form of an XML file. The XML file can then be placed in the intermediate stage to check for empty result queries before queries are being sent to DBMS to be evaluated. Figure 7.1 shows the architecture.

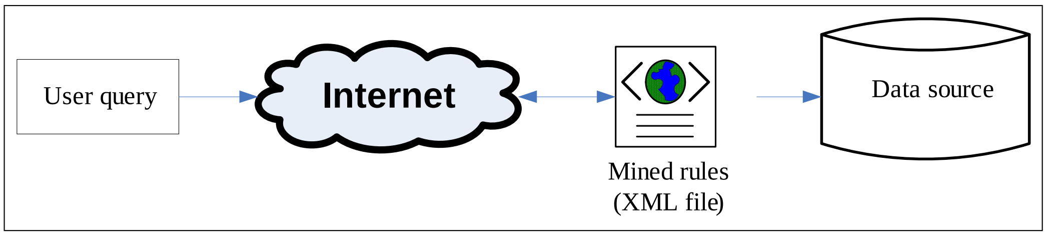


Figure 7.1: Implementing mined empty queries for online queries

These XML files are generated according to the following DTD, shown in Table 7.1. An example of such an XML file is shown in Table 7.2.

```
<!DOCTYPE emptySets [
      <!ELEMENT emptySets (emptySet*)>
      <!ELEMENT emptySet (rule_no, att1?, att2?,....attn?)>
      <!ELEMENT att1    (#PCDATA)>
      <!ELEMENT att2    (#PCDATA)>
       .
       .
      <!ELEMENT attn    (#PCDATA)>
]>
```

Table 7.1: DTD

```
<?xml version="1.0"?>
  <emptySets">
      <emptySet>
       <rule_no>1</rule_no>
       <att1>A</att1>
    </emptySet>
    <emptySet>
       <rule_no>2</rule_no>
       <att2>
              <min>5</min>
              <max>10</max>
       </att2>
    </emptySet>
    <emptySet>
         <rule_no>3</rule_no>
        <att1>B</att1>
         <att2>
                <min>15</min>
                <max>20</max>
        </att2>
    </emptySet>
  </emptySets>
```

Table 7.2: An example of an XML file

Query criteria matching can be done using XQuery FLOWR expression. Table 7.3 shows an example of XQuery used in respect to the XML file in Table 7.2:

If the XQuery evaluates to true and returns an XML node, then we know that the selection criteria of the query matches the empty rules. If no match is found, then query will be sent to the DBMS.

```
for $x in doc("empty.xml")/emptySets/emptySet
where $x/att1='A'
  or ($x/att2/min=5 and $x/att2/max<15)
return $x/number
```

Table 7.3: XQuery

Two applications were proposed in [Gryz and Liang, 2006]. They are:

1. Query optimization by query rewriting
   Empty hyper-rectangles can be materialized in views and used for query optimization. By reducing the range of one or more of the attributes or by adding a range predicate (hence reducing an attributes range), the number of tuples that participate in the join execution are reduced thus providing optimization.

2. Improve Cardinality estimates in query processing
   Currently, a join cardinality estimate is based on the assumption of uniform data distribution or by using histogram. By identifying and taking into consideration the empty regions in a join, the corresponding join cardinality estimate is more accurate.

7.3 Interesting Data Discovery

Clearly, the knowledge of empty regions may be valuable in and of itself as it may reveal unknown correlations between data values which can be exploited in applications. As described earlier, empty regions in data can be seen as a type of data skew. From the mined empty regions, it can lead to interesting data discovery, for example attribute correlations and functional dependencies.

In [Liu et al., 1997], the authors proposed to use the knowledge of empty regions in medical domains. For example, in a disease database, we may find that certain symtoms and/or test values do not occur together, or when a certain medicine is used, some test values never go beyond certain range. Discovery of such information can be of great importance in medical domains because it could mean the discovery of a cure to a disease or even some biological laws.

7.4 Other applications

1. Database administration
   The knowledge of empty regions can help database administrator in adding integrity constraint or check constraint in the database.

2. Detection of anomalies in updates
   From the mined rules, we are able to identify which ranges of values do not occur together. They can be used in checking for anomalies in updates.

8 Conclusion

The results have shown that given a set of attributes, the proposed solution is able to find the set of empty combination in the query selection that causes a particular query to return an empty result. The mined results are presented as disjunction of rules. In the special case of numeric attributes, these rules can be defined in terms of empty hyper-rectangles. The proposed solution has a few strong points over existing methods. Unlike previous methods, they focus only in finding empty regions formed by numeric values, this methods explores the combinations of category and numeric values. Besides being able to mine mixed type of attributes, it can also mine hyper-rectangles in higher dimensions. It is shown that the algorithm is independent of the size of the actual dataset but dependent only on the size of the attribute domains. It is efficient for low cardinality domains, but performance degrades if the cardinality of the attribute domains becomes too large. In this method, false positive are allowed, but no false negatives are reported. This means that all mined rules are guaranteed to be 'empty', however there may be some empty regions that go undetected. Since this solution has good performance for low cardinality domains, therefore it can be implemented to detect empty query results for 'canned queries'.

8.1 Future Work

We expect this feature of mining empty result queries to be most helpful in read-mostly environments. Extending this to a more update-intensive environment is an interesting area for future work. The continuation of this work would be to use the mined results to efficiently detect empty result queries. As stated, there exist a few previous works that check for empty regions in the dataset, however only [Luo, 2006] focused on empty result queries detection.

Section 6.7.7 in Chapter 6 contains the list of drawbacks and limitation of the proposed method. One of them is that the mined rules are too fragmented to be useful in empty query detection. Thus, it is worth investigating on how to optimize the current rules, by combining these 'fragments', making them large enough to be used in empty query detection. By combining these 'fragmented' rules, we can minimize the number of rules used to define empty spaces. This optimization problem of finding the minimal number of rules to fully identify the empty regions is known to be NP-Hard. Just like it is stated in [Agrawal et al., 1998], computing the optimal cover is also NP-hard. Nonetheless, it will be interesting to see the performance of empty query detection in such optimized condition.

Bibliography

R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 12–15 1994. ISBN 1-55860-153-8. URL `citeseer.ist.psu.edu/agrawal94fast.html`. 4.4, 4.4.2

R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. pages 94–105, 1998. URL `citeseer.ist.psu.edu/agrawal98automatic.html`. 2.2.3, 3.2.1, 6.7.5, 8.1

Q. Cheng, J. Gryz, F. Koo, T. Y. C. Leung, L. Liu, X. Qian, and K. B. Schiefer. Implementation of two semantic query optimization techniques in db2 universal database. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 687–698, 1999. 1

S. S. Cosmadakis, P. C. Kanellakis, and N. Spyratos. Partition semantics for relations. In *Journal Computer System Science*, pages 203–233, 1986. 4.4.1

J. Edmonds, J. Gryz, D. Liang, and R. J. Miller. Mining for empty rectangles in large data sets. In *ICDT*, pages 174–188, 2001. 2.1, 2.2.2, 3.1, 6.7.1

T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Data mining using two-dimensional optimized association rules: scheme, algorithms, and visualization. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 13–23, New York, NY, USA, 1996. ACM. ISBN 0-89791-794-4. doi: http://doi.acm.org/10.1145/233269.233313. 3.2.1

J. Gryz and D. Liang. Holes in joins. In *J. Intell. Inf. Syst.*, volume 26, pages 247–268, Hingham, MA, USA, 2006. Kluwer Academic Publishers. doi: http://dx.doi.org/10.1007/s10844-006-0368-2. 1, 1.1, 2.2.2, 3.1, 5.1, 7, 7.2

Y. Huhtala, J. Kinen, P. Porkka, and H. Toivonen. Efficient discovery of functional and approximate dependencies using partitions. In *ICDE*, pages 392–401, 1998. URL `citeseer.ist.psu.edu/huhtala97efficient.html`. 4.4.1

W. Kießling and G. Köstler. Preference sql - design, implementation, experiences. In *VLDB*, pages 990–1001, 2002. 1.1

B. Lent, A. Swami, and J. Widom. Clustering association rules. In *ICDE*, 1997. URL `mack.ittc.ku.edu/lent97clustering.html`. 4.1.2

B. Liu, L.-P. Ku, and W. Hsu. Discovering interesting holes in data. In *IJCAI (2)*, pages 930–935, 1997. 2.1, 2.2.2, 2.2.2, 2.2.3, 6.7.1, 6.7.7, 7.3

B. Liu, K. Wang, L.-F. Mun, and X.-Z. Qi. Using decision tree induction for discovering holes in data. In *Pacific Rim International Conference on Artificial Intelligence*, pages 182–193, 1998. URL `citeseer.ist.psu.edu/liu98using.html`. 2.1, 2.2.2, 6.7.1

G. Luo. Efficient detection of empty-result queries. In *VLDB*, pages 1015–1025, 2006. 1.1, 2.2.1, 6.7.1, 8.1

R. J. Miller and Y. Yang. Association rules over interval data. In J. Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 452–461. ACM Press, 1997. 4.2

D. J. Newman, S. Hettich, C. L. Blake, and C. J. Merz. UCI repository of machine learning databases, 1998. URL `http://www.ics.uci.edu/~mlearn/MLRepository.html`. A.2, 2

N. Spyratos. The partition model: a deductive database model. In *ACM Trans. Database Syst.*, pages 1–37, 1987. 4.4.1

R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 1996. URL `citeseer.ist.psu.edu/srikant96mining.html`. 4.2

I. H. Witten and E. Frank. WEKA data mining: Practical machine learning tools and techniques, 2005. URL `http://www.cs.waikato.ac.nz/~ml/weka/`. 6, 1, 2, 1, 3

X. Yin, J. Han, J. Yang, and P. Yu. Crossmine: Efficient classification across multiple database relations. 2004. URL `citeseer.ist.psu.edu/yin04crossmine.html`. 5.1

T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: an efficient data clustering method for very large databases. In *ACM SIGMOD International Conference on Management of Data*, pages 103–114, Montreal, Canada, June 1996. URL `citeseer.ist.psu.edu/article/zhang96birch.html`. 4.2

A

More Test Results

A.1 Synthetic Dataset

1. Testset #1

- Description: artificial dataset with 6 attributes (3 discrete and 3 continuous). Data are generated with dependencies between the attribute values.
- Size of dataset: 40,768 rows of records
- Source: collection of regression datasets at WEKA dataset repository [Witten and Frank, 2005].

Description	**Type**	**Values**	**Domain Cardinality**	**min**	**max**
X3	Low cardinality	brown, red, green	3		
X4	Numeric	integer	61 clusters	-7.5	2.5
X6	Numeric	integer	78 clusters	-37.194	12.114
X7	Boolean	yes/no	2		
X8	Low cardinality	normal, large	2		
Y	Numeric	integer	79 clusters	-41.822	2.5

(a) Attributes of testset #1

Dimension	*distribution_size*	**# rules**	**time (in seconds)**
first 2 attributes	0.34	44	910
first 3 attributes	0.22	423	2,895
first 4 attributes	0.19	1,133	6,592
first 5 attributes	0.19	1,216	14,140
all attributes	0.08	34,772	31,322

(b) Results

Table A.1: Testset #1

2. Testset #2 [fried]

- Description: artificial dataset with 6 attributes (all continuous). All values are generated independently each other, of which uniformly distributed over [0,1].
- Size of dataset: 40,768 rows of records
- Source: collection of regression datasets at WEKA dataset repository [Witten and Frank, 2005].

Description	**Type**	**Values**	**Domain Cardinality**	**min**	**max**
Att1	Numeric	integer	81 clusters	0	1
Att2	Numeric	integer	78 clusters	0	1
Att3	Numeric	integer	82 clusters	0	1
Att4	Numeric	integer	87 clusters	0	1

(a) Attributes of Testset #2

Dimension	*distribution_size*	**# rules**	**time (in seconds)**
first 2 attributes	0.42	151	797
first 3 attributes	0.23	227	2,809
All attributes	0.12	302	7,889

(b) Results

Table A.2: Testset #2

A.2 Real Life Dataset

For evaluation of real datasets, we have chosen a few datasets from UCI KDD (Knowledge Discovery in Databases) [Newman et al., 1998] and from statlib of CMU. Datasets from both these sources are provided in WEKA machine learning homepage.

1. Ailerons

   - Description: This data set addresses a control problem, namely flying a F16 aircraft. The attributes describe the status of the aeroplane.
   - Size of dataset: 13,750 rows of records
   - Source: collection of regression datasets at WEKA dataset repository [Witten and Frank, 2005].

Description	**Type**	**Values**	**Domain Cardinality**	**min**	**max**
climbRate	Numeric	integer	76 clusters	-975	977
q	Numeric	integer	58 clusters	-0.54	0.62
curPitch	Numeric	integer	71 clusters	-0.42	2.58
curRoll	Numeric	integer	61 clusters	-3.1	2.9
absRoll	Numeric	integer	21 clusters	-23	-3
diffDiffClb	Numeric	integer	53 clusters	-62	46

(a) Attributes of Ailerons

Dimension	*distribution_size*	**# rules**	**time (in seconds)**
first 2 attributes	0.24	5	101
first 3 attributes	0.18	683	420
first 4 attributes	0.12	1,222	916
first 5 attributes	0.09	3,404	1,907
all attributes		0.04 11,268	4,016

(b) Results

Table A.3: Ailerons

2. KDD internet usage

   - Description: This data comes from a survey conducted by the Graphics and Visualization Unit at Georgia Tech in 1997. The particular subset of the survey provided here is the "general demographics" of Internet users.
   - Size of dataset: 10,108 rows of record
   - Source: UCI KDD [Newman et al., 1998]

Description	**Type**	**Values**	**Domain Cardinality**
Community Building	Low-card.	{D, E, L, M}	4
Gender	Boolean	{Female, Male}	2
Household Income	Low-card.	{[10-19], [20-29], [30-39], [40-49], [50-59], [75-99], Nil, Over100}	9
Occupation	Low-card.	{Computer, Education, Management, Other, Professional}	5
Marital Status	Low-card.	{Divorced, Married, Not_Say, Other, Separated, Single, Widowed}	7
Race	Low-card.	{Asian, Black, Hispanic, Indegenous, Latino, Not_Say, Others}	8
Years on internet	Low-card.	{1-3Yr, 4-6Yr, 6-12Yr, Over7Yr, Under6Mth}	5

(a) Attributes of KDD Internet Usage

Dimension	*distribution_size*	**# rules**	**time (in seconds)**
first 2 attributes	1	-	-
first 3 attributes	0.97	1	33
first 4 attributes	0.95	32	77
first 5 attributes	0.53	592	181
first 6 attributes	0.11	2,524	755
all attributes	0.04	6,987	7,800

(b) Results

Table A.4: KDD Internet Usage Survey

3. Credit Card Scoring
    - Description: Credit card application and approval based on application scoring.
    - Size of dataset: 5,000 rows of records
    - Source: collection of regression datasets at WEKA dataset repository [Witten and Frank, 2005].

Description	**Type**	**Values**	**Domain Cardinality**	**min**	**max**
Age	Numeric	integer	22 clusters	20	50
Income	Numeric	integer	28 clusters	1.51	10
Monthly CC expenditure	Numeric	integer	15 clusters	0	1898.03
Own a home	Boolean	{yes / no}	2		
Category scoring	low-card.	{0,1,2,3,4,7}	6		

(a) Attributes of Credit Card Scoring

Dimension	*distribution_size*	**# rules**	**time (in seconds)**
first 2 attributes	0.34	16	48
first 3 attributes	0.12	30	193
first 4 attributes	0.10	30	283
All attributes	0.08	51	384

(b) Results

Table A.5: Credit Card Scoring