NOVEL TECHNIQUES FOR DATA WAREHOUSING AND ONLINE ANALYTICAL PROCESSING IN EMERGING APPLICATIONS

by

Moonjung Cho

September 1, 2006

A dissertation submitted to the

Faculty of the Graduate School of

State University of New York at Buffalo

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science and Engineering

To my eternal mentor Daisaku Ikeda

Acknowledgements

This thesis could not have been written without the invaluable support of my advisor, Dr. Jian Pei. His promptness, sincerity, patience, and challenging spirit are great forces for me to break through every deadlock in the process of researches. It is impossible for me to express how much I appreciate to his unflagging trust and encouragements.

I am very grateful to Dr. Xin He and Dr. Venugopal Govindaraju for review and advices on my thesis. Also, I should not omit to express my thanks to Dr. Kenneth W. Regan for his encouragement, and Dr. Guozhu Dong for a thorough review and comments.

My family has provided support from behind the scenes. This thesis is dedicated to my parents, to whom I owe the most. I wish to thank to my six roommates in Buffalo and many friends in the world. I will not forget our precious memories of hearty laughs, smiles, tears, and joy. I also strongly hope that all of my friends will fulfill their dreams and goals. I especially thank to eternal friends in SGI for unlimited and unconditional encouragements. They have shown me what true friendship is and how wonderful and powerful heart-to-heart connections are.

I am for sure that the only way to pay my gratitude in debt to all is to never stop developing myself so that I am able to contribute something more positive value to this world. I will not let you down.

Abstract

A data warehouse is a collection of data for supporting of decision making process. Data cubes and on-line analytical processing(OLAP) have become very popular techniques to help users analyze data in a warehouse. Even though previous studies on a data warehouse and data cube have been proposed and developed, as new applications emerging, there are still technical challenges which have not been addressed well.

We propose effective and efficient solutions to the challenging problems in the areas of (1) mining iceberg cube from multiple tables, (2) online answering ad-hoc aggregate queries on data streams, and (3) warehousing pattern-based clusters.

Firstly, we argue that the materialized base table assumption in most of the previous studies on computing iceberg cubes is often infeasible in practice. Instead, a data warehouse is often organized with multiple tables in schemas such as star schema, snowflake schema, and constellation schema. We propose a novel approach to compute an iceberg cube from multiple tables in a data warehouse in order to avoid costly materialization of a base table. Secondly, it is infeasible to compute a full data cube for answering ad-hoc aggregate queries on data streams due to a rapid data input and the huge size of data. We develop a new method to answer online ad-hoc aggregate queries on data streams, which is to maintain and index a small subset of aggregate cells on a designed data structure. Last, we extend the data warehousing and OLAP techniques to tackle pattern-based clusters. We propose an efficient method to construct a data warehouse of non-redundant pattern-based clusters.

Table of Contents

List of Figures

Chapter 1

Introduction

Data warehousing and online analytic processing are essential facilities for data analysis tasks supporting a user's decision in a business. This dissertation reports the novel techniques for data warehousing and online analytical processing in emerging applications. It especially focuses on the challenges in data warehouse and online analytical processing and presents new techniques to tackle those challenges. In this chapter, we begin with a general overview of data warehouse and online analytical processing concepts, and then motivations and contributions are presented.

Section 1.1 gives the overview of data warehouse and OLAP concepts and applications. Section 1.2 shows what motivates our researches on data warehouse and OLAP. Our contributions responding to the motivations are described in the Section 1.3. Section 1.4 outlines the rest of the dissertation.

1.1 Data warehouse and OLAP

Data warehousing is an architecture to help business executives to understand and organize data and make a business decision. A data warehouse is a *subject oriented, integrated, time variant, non volatile* collection of data in support of decision making processes(Inmon, 2002). Data warehousing approach is to integrate information and heterogeneous sources in advance, store the historical information in a warehouse and support complex multidimensional queries. On-

Line Analytical Processing(OLAP) manages data warehouses for data analysis and provides calculations such as summarization and aggregation in advance, and manages information at different levels of granularity. OLAP has become very popular techniques to help users analyze data by providing multiple views of the data.

Data warehouses and OLAP tools are based on a multidimensional data model. A data cube is a model for multi-dimensional database and is defined by dimensions and facts(or measures). Dimensions are entities of records which a company want to keep, and facts are numerical measures or quantities. Given a base table consisting of dimensions and measures and an aggregate function, a data cube consists of the complete set of group-bys on any subsets of dimensions and their aggregates using the aggregate function. The number of group-bys(cuboids) is exponential to the number of dimensions. A data cube in practice is often huge due to the very large number of possible dimension value combinations. Even many detailed aggregate cells whose aggregate values are too small may be trivial in data analysis. To overcome the curse of dimensionality, an iceberg cube has been proposed.

An iceberg cube consists of only the set of group-bys whose aggregates are no less than a user-specified aggregate threshold, and does not compute a complete cube. Mining iceberg cubes is an important research problem in both online analytic processing (OLAP) and data mining. It can be to answer group-by queries, mine multidimensional association rules, and identify interesting subsets of the cube for precomputation(Beyer & Ramakrishnan, 1999).

Based on the concept of data warehouses and OLAP methods, we like to present what motivated our researches.

1.2 Motivations

As we mentioned in the previous section, data warehouses and OLAP techniques have developed for users to understand data and make a decision. However, the techniques in a data warehouse have still challenges in situations where a data warehouse may have complicated schemas instead of materialized single base table, and manage a data streams. Among many

of those challenges on data warehouse and data cube, we propose several interesting ones we would like to address.

- **Mining Iceberg Cubes.** A data warehouse is often organized in a schema of multiple tables, such as star schema or snowflake schema, in practice. Several efficient algorithms, such as *BUC* (Beyer & Ramakrishnan, 1999), MultiWay (Y. Zhao et al., 1997), H-Cubing (Han et al., 2001), Star-Cubing (Xin et al., 2003), and Range Cube (Feng et al., 2004), have been proposed to compute iceberg cubes efficiently from a *single* base table, with simple or complex measures. Although mining iceberg cube from a single table becomes more efficient, such algorithms cannot be applied directly to real data warehouses in many applications. Also the cost to materialize single base table from a data warehouse is high in space and time due to the redundancy and multiple scans of dimensional tables. This observations become a motivating question:"*Can we compute iceberg cubes efficiently from multiple tables without materializing a universal base table?*"

- **Aggregate Queries on a Data Stream.** Recently, several important applications see the strong demands of online answering *ad hoc aggregate queries* over fast data streams. In those applications, it is required to *maintain the recent data in a sliding window*, and *provide online answers to ad hoc aggregate queries over the current sliding window*. Unfortunately, a traditional data warehouse often updates in batch periodically and such updates are often conducted offline. Therefore, online aggregate queries about the most recent data cannot be answered by the traditional data warehouses due to the delay of the incremental updates. Moveover, as a data cube, the complete set of aggregate cells on a multidimensional base table over a data stream can be huge. Herein we have a motivating question to address: "*Can we materialize and incrementally maintain a small subset of aggregates by scanning the data stream only once, and still retain the high performance of online answering ad hoc aggregate queries?*"

- **Warehouse Pattern-based Clusters.** Clustering data is a challenging data mining task with many important applications. Clustering methods need similarity measures defined

globally on set of attributes/dimensions. However, in some applications, it is hard or even infeasible to define a good similarity measure on a global subset of attributes to serve the clustering. As indicated by some recent studies (Jiang et al., 2003,0; Liu & Wang, 2003; L. Zhao & Zaki, 2005; H. Wang et al., 2002), pattern-based clustering is introduced and useful in many applications. In general, given a set of data objects, a subset of objects forms a pattern-based clusters if these objects follow a similar pattern in a subset of dimensions. Comparing to the conventional clustering, pattern-based clustering has two distinct features. First, pattern-based clustering does not require a globally defined similarity measure. Instead, it specifies quality constrains on clusters. Different clusters can follow different patterns on different subsets of dimensions. Second, the clusters are not necessarily exclusive. In other words, an object can appear in more than one cluster. Due to the non-exclusiveness of clusters, a lot of redundancy exist in the complete set of pattern-based clusters. Pattern-based clustering problem is proposed and a mining algorithm is developed by H. Wang et al. (2002). Our motivating question is "*What is the effective representation of non-redundant pattern-based clusters? Moreover, how can we construct a data warehouse of non-redundant pattern-based clusters efficiently?*"

1.3 Contributions

Responding to the challenges in the previous section, our contributions are as follows:

- We develop a novel method *CTC* which directly computes iceberg cubes from a star schema in a data warehouse without materializing the universal base table. It removes high cost taken by materialization of the universal base table as an input to the previous methods for computation of iceberg cubes. It computes local iceberg cells in each dimensional table, and then derives the global ones using key relations between a fact table and dimensional tables. Due to the usage of local iceberg cells from dimensional tables, it avoids redundancy of data caused by the materialization of the universal base table. The experimental results of *CTC* clearly indicate that *CTC* is efficient and scalable in comput-

ing iceberg cubes for large data warehouses in a star schema. Also, the idea of *CTC* can be generalized and extended to handle more complicated schemas.

- In response to strong demands of online answering ad hoc aggregate queries over fast data streams, a novel *PAT* data structure is proposed to construct an online data warehouse. Efficient algorithms are also developed to construct and incrementally maintain a *PAT* over a data stream, and answer various ad hoc aggregate queries. The key points of this work are as follows:

  1. A *PAT* maintains a small subset of aggregates from a recent data over a sliding window, not the complete set of aggregate cells.
  2. It scans the data stream only once.
  3. It answers various ad hoc aggregate queries over a recent data over a sliding window exactly instead of approximately.

  This work certainly opens a way to apply mining iceberg cubes to a online data warehousing over a data stream.

- For an effective representation of non-redundant pattern-based clusters, we propose *maximal* pattern-based clusters, which are non-redundant clusters. We also develop *MaPle* and *MaPle+*, two efficient and scalable algorithms, for mining maximal pattern-based clusters in large databases. By removing the redundancy, it reduces the cost to find redundant clusters so that the effectiveness of the mining can be improved substantially.

1.4 Dissertation Outline

The remainder of the dissertation is organized as follows.

Chapter 2 describes the problem of computing iceberg cubes from data warehouses. An algorithm *CTC* avoids materializing the universal base table, instead it facilitates the local iceberg cubes of the dimension tables. We also show by experiments that *CTC* is more efficient than

any other methods. Some ideas to handle more complicated schemas such as snowflake schema and constellation schema are discussed.

Chapter 3 proposes a novel *PAT* data structure to construct an online data warehouse. It maintains the data stream within a sliding window in main memory. We present efficient algorithms to maintain a *PAT* and answer essential aggregate queries, including point and range queries.

Chapter 4 studies the pattern-based clustering and proposes the mining of maximal pattern-based clusters which are non-redundant pattern-based clusters. It also presents two efficient algorithms called *MaPle* and *MaPle+*.

Chapter 5 concludes the works on data warehouse and data cubing in the dissertation. Interesting future research topics are also discussed.

Chapter 2

Computing data cube and iceberg cube from data warehouses

2.1 Preliminaries

Mining iceberg cubes from a data warehouse is a very useful technology since it serves only data cells satisfying specified aggregate threshold. This chapter describes inherent challenges of previous mining iceberg cubes algorithms and our novel method to solve the challenge.

Example 2.1 (Motivating example). The data warehouse in Figure 2.1 records the information about sales of automobiles, and is organized in a star schema. Table *Sales* is the fact table and tables *Customer*, *Brand*, *Model* are dimension tables. Attribute *profit* in the fact table is the measure.

A sales manager may want to compute an iceberg cube with condition $avg(profit) \geq \$3,000$, i.e., finding the groups of sales that bring in a profit of $3,000$ dollars or more on average. Such group-bys, such as "*sales to customers of 30s buying 6-cylinder cars with sun-roof have average profit of* $\$3,500$" may be interesting to her, since she can use the information to promote the 6-cylinder cars with sun-roof to the target customer group. □

Example 2.1 shows how mining iceberg cubes can be used in a business and also a data warehouse in practice has multiple tables in a simple schema such as star schema, snowflake

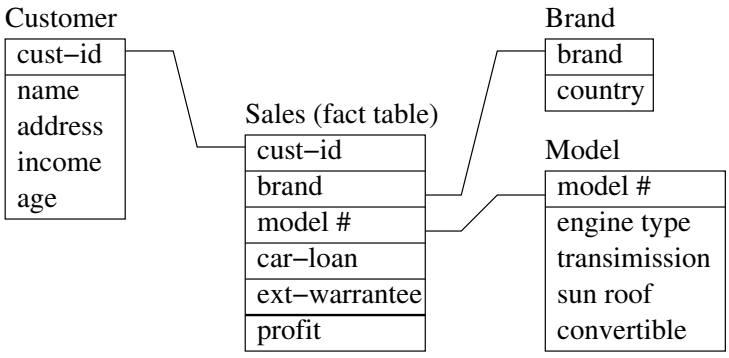


Figure 2.1: The auto data warehouse in star schema.

schema, not a huge universal base table. However, previous efficient algorithms (Beyer & Ramakrishnan, 1999; Y. Zhao et al., 1997; Han et al., 2001; Xin et al., 2003; Feng et al., 2004) have assumed a single universal base table as an input. Let us come up a rudimentary approach to compute iceberg cubes from multiple tables.

Given a data warehouse with multiple tables, one rudimentary approach may have two steps to mining iceberg cubes. First, a universal base table is formed by joining the related tables. Take the auto data warehouse in Figure 2.1 as an example, we can compute a base table $Sales_{base} = Sales \bowtie Customer \bowtie Brand \bowtie Model$. Once the base table is formed, we can apply an existing iceberg cube computation method to derive the iceberg cube.

Although the rudimentary method is simple, it may not be efficient or may not be even feasible in a real application. Usually, a large data warehouse may contain tens of dimensions and millions of tuples. It is often unaffordable in both space and time to join the related tables and form the universal base table.

Then, the purpose is to compute iceberg cubes efficiently from multiple tables without materializing a universal base table. Surprisingly, our investigation indicates that computing iceberg cube from multiple tables directly without materializing a universal base table may be even more efficient in both space and runtime than computing from the universal base table even though we assume that the universal base table is already instantiated. While the systematic study will be presented in Section 2.3, we highlight the intuition in the following example.

Example 2.2 (Intuition). Consider computing the iceberg cube from tables F and D in Figure 2.2. Suppose attribute M is the measure.

K	A_1	$\cdots$	A_n	M
k	$a_{1,1}$	$\cdots$	$a_{1,n}$	m_1
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
k	$a_{l,1}$	$\cdots$	$A_{l,n}$	m_l

Table F

K	B_1	$\cdots$	B_m
k	b_1	$\cdots$	b_m

Table D

A_1	$\cdots$	A_n	K	B_1	$\cdots$	B_m	M
$a_{1,1}$	$\cdots$	$a_{1,n}$	k	b_1	$\cdots$	b_m	m_1
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$a_{l,1}$	$\cdots$	$a_{l,n}$	k	b_1	$\cdots$	b_m	m_l

Universal base table $B = F \bowtie D$

Figure 2.2: A simple case of computing iceberg cube from two tables.

A rudimentary method may first compute a universal base table $B = F \bowtie D$, as also shown in the figure, and then compute the iceberg cube from B. However, such a rudimentary method may suffer from two non-trivial costs.

- *Space cost.* As shown in the figure, the tuple in table D is replicated l times in the universal base table B, where l is the number of tuples in the fact table. Moreover, every attribute in the tables appears in the universal base table. Thus, the universal base table is wider than any table in the original database. In real applications, there can be a large number of tuples in the fact table, and hundreds of attributes in the database. Then, the dimension information may be replicated many times, and the universal base table may be very wide – containing hundreds of attributes.

- *Time cost.* The large base table may have to be scanned many times and many combinations of attributes may have to be checked. As the universal base table can be much wider and larger than the original tables, the computation time can be dramatic.

Can we compute iceberg cubes directly from F and D without materializing the universal base table B? The following two observations help.

First, for any combination of attributes in table D, the aggregate value is $m = aggr(\{m_1, \ldots, m_l\})$. Therefore, if m satisfies the iceberg condition, then every combination of attributes in D is an iceberg cell. Please note that we compute these iceberg cells using table D only, which contains only 1 tuple. In the rudimentary method, we have to use many tuples in table B to compute these iceberg cells. The saving is significant!

Second, for any iceberg cell involving attributes in table F, the aggregate value can be computed from table F only. In order words, if we find an iceberg cell in F, we can enumerate a whole bunch of iceberg cells by inserting more attributes in D and the aggregate value retains. Please note that we only use F, which has only $(n+1)$ attributes, to compute these iceberg cells. In the rudimentary method, we have to compute these iceberg cells using a much wider universal base table B. This is another significant saving. □

In the Section 2.3, we tackle the problem of mining iceberg cubes from data warehouses, and make the following contributions.

First, *we address the problem of mining iceberg cubes from data warehouses of multiple tables*. Particularly, we formulate the problem of computing iceberg cubes from star schema and propose efficient algorithms. Our approach can be easily extended to handle other schemas in data warehouses, such as snowflake schema.

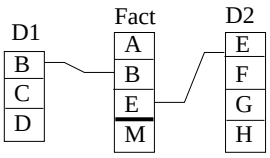
Second, *we develop an efficient algorithm,* CTC *(for Cross Table Cubing), to compute iceberg cubes from star schema*. Our method does not need to materialize the universal base table. Instead, *CTC* works in three steps. First, *CTC* propagates the information of keys and measure to each dimension table. Second, the local iceberg cube in each table is computed. Last, the global iceberg cube is derived from the local ones. We show that *CTC* can be more efficient in both space and runtime than computing iceberg cube from a materialized universal base table.

Last, we conduct an extensive performance study on synthetic data sets to examine the efficiency and the scalability of our approach. The experimental results show that *CTC* is efficient and scalable for large data warehouses.

2.2 Problem Definition and Related Work

2.2.1 Problem Definition

We first consider only data warehouses in star schema. However, the techniques developed can be extended to handle data warehouses in more complicated schemas, such as snowflake



(a) The star schema

A	B	E	M
a_1	b_1	e_1	1
a_2	b_2	e_2	3
a_3	b_3	e_3	2
a_1	b_1	e_2	4
a_2	b_4	e_4	2

(b) Fact table *Fact*

B	C	D
b_1	c_1	d_1
b_2	c_1	d_2
b_3	c_1	d_3
b_4	c_2	d_1

(c) Dimension table D_1

E	F	G	H
e_1	f_1	g_1	h_1
e_2	f_1	g_2	h_2
e_3	f_1	g_1	h_2
e_4	f_2	g_1	h_1

(d) Dimension table D_2

Figure 2.3: Data warehouse *DW* as the running example.

schema. We will discuss the extensions in Section 2.5.

Definition 2.1 (Star schema). *A* **star schema** *is a set of tables $F, D_1, \ldots, D_n$, where*

- *$F = (K_1, \ldots, K_n, M)$ is called the* **fact table**. *$K_1, \ldots, K_n$ are the* **dimensions** *and M is the* **measure**. *K_i $(1 \leq i \leq n)$ is the foreign key referencing to dimension table D_i; and*
- *$D_1, \ldots, D_n$ are called the* **dimension tables**. *K_i is the primary key in D_i $(1 \leq i \leq n)$.*

Table $B = F \bowtie D_1 \bowtie \cdots \bowtie D_n$ is called the **universal base table**.

In Figure 2.1, table *Sales* is the fact table, and tables *Customer*, *Brand*, and *Model* are the dimension tables. Dimensions *car-loan* and *ext-warrantee* do not have further attributes. In other words, the dimension tables for those two dimensions contain only the dimension themselves and thus can be trivially omitted.

Attributes *cust-id*, *brand*, and *model#* serve as the foreign keys in the fact table and reference to the primary keys in the dimension tables, respectively.

As another example, consider the data warehouse *DW* in Figure 2.3. We will use this data warehouse as the running example in the rest of this chapter.

The star schema is shown in Figure 2.3(a). In data warehouse *DW*, the fact table *Fact* has 3 dimensions, namely A, B and E. The measure is M. Dimensions B and E reference

to dimension tables D_1 and D_2, respectively. In data warehouse DW, the universal base table $T_{base} = Fact \bowtie D_1 \bowtie D_2$ is shown in Figure 2.4.

A	B	C	D	E	F	G	H	M
a_1	b_1	c_1	d_1	e_1	f_1	g_1	h_1	1
a_2	b_2	c_1	d_2	e_2	f_1	g_2	h_2	3
a_3	b_3	c_1	d_3	e_3	f_1	g_1	h_2	2
a_1	b_1	c_1	d_1	e_2	f_1	g_2	h_2	4
a_2	b_4	c_2	d_1	e_4	f_2	g_1	h_1	2

Figure 2.4: The universal base table T_{base}.

Definition 2.2 (Iceberg cube). *Let $B = (A_1, \ldots, A_m, M)$ be a universal base table, where $A_1, \ldots, A_m$ are either dimensions or attributes in dimension tables. A cell $c = (a_1, \ldots, a_m)$ is called an* **aggregate cell***, where $a_i \in A_i$ or $a_i = *$ $(1 \leq i \leq m)$. The* cover *of c is the set of tuples in B that match all non-$*$ a_i's, i.e., $cov(c) = \{t \in B | \forall a_i \neq *, t.A_i = a_i\}$.*

For an aggregate function $aggr()$ on the domain of M, $aggr(c) = aggr(cov(c))$.

For an **iceberg condition** *C, where C is defined using some aggregate functions, a cell c is called an* **iceberg cell** *if c satisfies C. An* **iceberg cube** *is the complete set of iceberg cells.*

Example 2.3 (iceberg cube). In base table T_{base} (Figure 2.4), for aggregate cell $c = (*, b_1, *, d_1, *, f_1, *, *)$, $cov(c)$ contains 2 tuples, the first and the fourth tuples in T_{base}, because they match c in dimensions B, D and F. We have $\texttt{COUNT}(cov(c)) = 2$.

Consider iceberg condition $C \equiv (\texttt{COUNT}(c) \geq 2)$. Aggregate cell c satisfies the condition and thus is in the iceberg cube. □

Problem definition. The *problem of computing iceberg cube from a data warehouse* is that, given a data warehouse in star schema and an iceberg condition, compute the iceberg cube. □

In general, an iceberg condition can take any form. As indicated by previous studies, a specific category of iceberg conditions called *monotonic conditions* are often of particular interest.

For aggregate cells $c = (a_1, \ldots, a_m)$ and $c' = (a'_1, \ldots, a'_m)$, c is called an *ancestor* of c' and c' a *descendant* of c if for any $a_i \neq *$, $a'_i = a_i$ $(1 \leq i \leq m)$, denoted by $c' \sqsubseteq c$. Immediately, we have the following result.

Lemma 1. *For aggregate cells c and c′, if* $c' \sqsubseteq c$*, then* $cov(c') \subseteq cov(c)$. □

An iceberg condition C is called *monotonic* if for any aggregate cell c, if C holds for c, then C also holds for every ancestor of c. Some typical examples of monotonic iceberg conditions include $\texttt{COUNT}(c) \geq v$, $\texttt{MAX}(c) \geq v$, $\texttt{MIN}(c) \leq v$, $\texttt{SUM}(c) \geq v$ (if the domain of the measure consists of only non-negative numbers).

Example 2.4 (Monotonic iceberg condition). Consider iceberg condition $C \equiv (\texttt{COUNT}(c) \geq 2)$. It is monotonic. For example, as shown in Example 2.3, aggregate cell $c = (*, b_1, *, d_1, *, f_1, *, *)$ satisfies the condition. According to Lemma 1, every ancestor of c, such as $c_1 = (*, *, *, d_1, *, f_1, *, *)$ and $c_2 = (*, b_1, *, *, *, *, *, *)$, has a cover as a superset of $cov(c)$, and thus has a $\texttt{COUNT}()$ value greater than or equal to that of c. In other words, every ancestor of c also satisfies the condition and thus is in the iceberg cube.

As another example, the cover of aggregate cell $c' = (a_3, *, *, *, *, *, *, *)$ has only one tuple, the third tuple in T_{base} (Figure 2.4). It fails the condition and thus is not in the iceberg cube. The cover of every descendant of c' must have no more tuples than that of c' and thus cannot honor the condition, either. Thus, we do not even need to compute and search them in the iceberg cube mining. □

Monotonic iceberg conditions enable efficient pruning in iceberg cube mining. As shown by Han et al. (2001) and K. Wang et al. (2003) for many non-monotonic iceberg conditions, the corresponding iceberg cubes can be computed by adopting some weakened but monotonic conditions to approximate the original ones and push deep into the computation.

In this chapter, we focus on monotonic iceberg conditions written in a distributive aggregate function[1], and use condition $\texttt{COUNT}(c) \geq v$ as the illustration. The techniques developed here are general and can be used for any monotonic iceberg conditions. Moreover, the techniques (see Han et al., 2001; K. Wang et al., 2003) can be adopted and integrated into our framework, so that non-monotonic conditions can be handled.

[1] According to (Gray et al., 1997), an aggregate function $aggr()$ is *distributive* if it can be evaluated in a distributed manner as follows. There exists a function f such that, for any set of data D, D can be partitioned into an arbitrary number of exclusive subsets $D_1, \ldots, D_n$ and $\cup_{i=1}^{n} D_i = D$, we have $aggr(D) = f(aggr(D_1), \ldots, aggr(D_n))$. In (Gray et al., 1997), it is shown that $\texttt{COUNT}()$, $\texttt{MIN}()$, $\texttt{MAX}()$ and $\texttt{SUM}()$ are all distributive. In fact, $f = aggr$ for $\texttt{MIN}()$, $\texttt{MAX}()$ and $\texttt{SUM}()$, and $f = \texttt{SUM}$ for $\texttt{COUNT}()$.

2.2.2 Related Work

The data cube operator (Gray et al., 1997) is one of the most influential operators in OLAP. Many approaches (Beyer & Ramakrishnan, 1999; Ross & Srivastava, 1997; Ross & Zaman, 2000; Y. Zhao et al., 1997) have been proposed to compute data cubes efficiently from scratch. In general, they speed up the cube computation by sharing partitions, sorts, or partial sorts for group-bys with common dimensions.

It is well recognized that the space requirements of data cubes in practice are often huge. Some studies investigate partial materialization of data cubes (Beyer & Ramakrishnan, 1999; Han et al., 2001; Harinarayan et al., 1996). Methods to compress data cubes are studied in (Lakshmanann et al., 2002; Lakshmanan et al., 2003; Shanmugasundaram et al., 1999; Sismanis et al., 2002; W. Wang et al., 2002). Moreover, many studies(Barbara & Sullivan, 1997; Barbar & Wu, 2000; Vitter et al., 1998) investigate various approximation methods for data cubes.

In the rest of this Section, we review the major methods on computing (iceberg) cubes.

MultiWay (Y. Zhao et al., 1997) is an array-based top-down approach to computing complete data cube. The basic idea is that a high level aggregate cell can be computed from its descendants instead of the base table. For example, aggregate cell $(a,b,*,*)$ can be derived from aggregate cells $(a,b,c_1,*)$, $(a,b,c_2,*)$, etc.

To compute a data cube on a base table $T(A,B,C,D)$, MultiWay first scans the base table once and computes group-bys (A,B,C,D), $(*,B,C,D)$, $(*,*,C,D)$, $(*,*,*,D)$ and $(*,*,*,*)$. These group-bys can be computed simultaneously without resorting the tuples in the base table. Once these group-bys are computed, we do not need to scan the base table any more. For example, group-bys $(A,B,C,*)$, $(A,B,*,D)$ and $(A,*,C,D)$ can be derived from group-by (A,B,C,D). The computation order is summarized in Figure 2.5.

MultiWay may not be efficient in computing iceberg cubes with monotonic iceberg conditions, because the top-down search cannot use the monotonic iceberg condition to prune.

Fang et al. (1998) proposed the concept of iceberg queries and developed some sampling algorithms to answer such queries. Beyer & Ramakrishnan (1999) introduced the problem of iceberg cube computation in the spirit of the paper by Fang et al. (1998) and developed algorithm

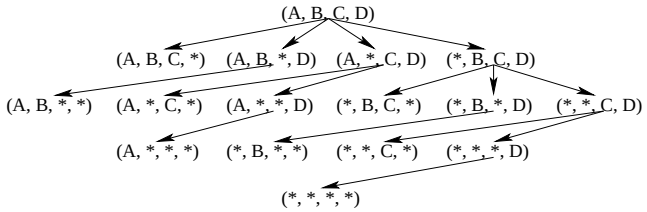


Figure 2.5: Top-down computation in MultiWay.

BUC. *BUC* conducts bottom-up computation and can use the monotonic iceberg conditions to prune. To compute a data cube on a base table $T(A,B,C,D)$, *BUC* first partitions the table according to dimension A, i.e., computing group-bys $(A,*,*,*)$. If an aggregate cell $(a,*,*,*)$ fails the monotonic iceberg condition, any descendant of it, such as $(a,b,*,*)$, $(a,*,c,*)$ must also fail the condition and thus does not need to be computed. Otherwise, *BUC* recursively searches the partition of $cov(a,*,*,*)$, and computes the further aggregates in depth-first search manner. The computation order is summarized in Figure 2.6.

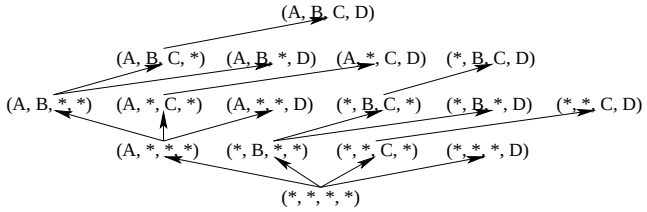


Figure 2.6: Bottom-up computation in *BUC* and H-Cubing.

BUC is efficient in computing iceberg cubes with monotonic iceberg conditions. It also employs counting sort to make partitioning efficient.

H-cubing (Han et al., 2001) uses a hyper-tree data structure called H-tree to compress the base table. Then, the H-tree can be traversed bottom-up to compute iceberg cubes. It also can prune unpromising branches of search using monotonic iceberg conditions. Moreover, a strategy was developed by Han et al. (2001) to use weakened but monotonic conditions to approximate non-monotonic conditions to compute iceberg cubes.

The strategies of pushing non-monotonic conditions into bottom-up iceberg cube computation were further improved by K. Wang et al. (2003). A new strategy, divide-and-approximate, was developed. The general idea is that the weakened but monotonic condition can be made up for each sub-branch search and thus the approximation and pruning power can be stronger.

Xin et al. (2003) developed Star-Cubing by extending H-tree to Star-Tree and integrating the top-down and bottom-up search strategies. Feng et al. (2004) proposed another interesting cubing algorithm, Range Cube, which uses a data structure called range trie to compress data and identify correlation in attribute values.

On the other hand, since iceberg cube computation is often expensive in both time and space, parallel and distributed iceberg cube computation has been investigated. For example, Ng et al. (2001) studied how to compute iceberg cubes efficiently using PC clusters.

All of the previous studies on computing iceberg cubes make an implicit assumption: *a universal base table is materialized*. However, this assumption may not be always true in practice – many data warehouses are stored in tens or hundreds of tables. It is often unaffordable to compute and materialize a universal base table for iceberg cube computation. This observation motivates the study in this section.

2.3 CTC: A Cross Table Cubing Algorithm

In this section, we develop *CTC*, a cross table cubing algorithm. We first present the general ideas and the framework of the algorithm, and then devise the details of the algorithm step by step.

Algorithm *CTC* works in three steps. First, the aggregate information is propagated from the fact table to each dimension tables. Then, the iceberg cubes in the propagated dimension tables as well as in the fact table (i.e., the *local iceberg cubes*) are mined independently using the same iceberg cube condition. Last, the iceberg cells involving attributes in multiple dimension tables are derived from the local iceberg cubes.

The correctness of the above three-step procedure is supported by the following basic observation: *for an iceberg cell c with respect to a monotonic iceberg condition, its projections on the fact table and the dimension tables must also be local iceberg cells.* The observation is formulated as follows.

Lemma 2 (Extended a priori property). *In a base table* $B = (A_1, \ldots, A_n, M)$ *where* M *is the*

Algorithm *CTC*
Input: a data warehouse *DW* in star schema, a monotonic iceberg condition *C*;
Output: an iceberg cube;
Method:

1: propagate the information about aggregates from the fact table to the dimension tables;
2: compute local iceberg cube for each dimension table;
3: compute the iceberg cells on the fact table and join the local iceberg cells to derive the global ones

Figure 2.7: Algorithm *CTC*.

measure, if $c = (a_1, \ldots, a_n)$ *is an iceberg cell with respect to a monotonic iceberg condition C, then, for any subset of attributes* $\{A_{i_1}, \ldots, A_{i_l}\}$ $(1 \leq i_1 < \cdots < i_l \leq n)$, $(a_{i_1}, \ldots, a_{i_l})$ *is an iceberg cell in table* $\sigma_{A_{i_1},\ldots,A_{i_l},M}(B)$ *with respect to C, where* σ *is the multi-set (i.e., bag) projection operator.*

Proof. This lemma can be regarded as the a priori property (Agrawal & Srikant, 1994) on dimensions of iceberg cube. To show the correctness, consider cell c' such that $c'.A_{i_j} = a_{i_j}$ $(1 \leq j \leq l)$ and all other attributes of c' take value $*$. Then, c' is an ancestor of c. Since c satisfies the monotonic iceberg condition C, so does c'. It is easy to see that cell $(a_{i_1}, \ldots, a_{i_l})$ on table $\sigma_{A_{i_1},\ldots,A_{i_l},M}(B)$ has the exactly same aggregate value as c', and thus also satisfies C. □

Based on Lemma 2, instead of directly computing the iceberg cube from a universal base table, we can compute local iceberg cubes from the fact table and the dimension tables, respectively. Then, we can try to derive the global iceberg cube from the local ones.

To enable the computation of local iceberg cubes on dimension tables and the derivation of global iceberg cubes from local ones, before we compute the local iceberg cubes, we have to propagate the aggregate information from the fact table to the dimension tables.

Based on the above idea, the framework of *CTC* is shown in Figure 2.7. The details in algorithm *CTC* are presented step by step in the following subsections.

2.3.1 Propagation Across Tables

Example 2.5 (Propagating aggregate information). To propagate the aggregate information from the fact table *Fact* to the dimension tables D_1 and D_2, we create a new attribute *Count*

in every dimension table. By scanning the fact table once, the number of occurrences of each foreign key value in the fact table can be counted. Such information is registered in the column of *Count* in the dimension tables, as shown in Figure 2.8. Hereafter, the propagated dimension tables are denoted as PD_1 and PD_2, respectively, to distinguish from the original dimension tables.

B	C	D	*Count*
b_1	c_1	d_1	2
b_2	c_1	d_2	1
b_3	c_1	d_3	1
b_4	c_2	d_1	1

(a) Propagated PD_1

E	F	G	H	*Count*
e_1	f_1	g_1	h_1	1
e_2	f_1	g_2	h_2	2
e_3	f_1	g_1	h_2	1
e_4	f_2	g_1	h_1	1

(b) Propagated PD_2

Figure 2.8: The propagated dimension tables.

In the rest of the computation, we only use the fact table and the propagated dimension tables PD_1 and PD_2. We will show that the iceberg cube computed from these three tables is the same as the one computed from the universal base table. □

This computation of the aggregates on the keys is implemented as group-by aggregate queries on the key attributes in the fact table. Only the fact table is needed to conduct such queries. The aggregate information is appended to the records in the dimension tables after the aggregates are computed. In general, we extend every dimension table to include a measure column.

The difference between aggregate propagation and joining the fact table and the dimension tables to materializing the universal base table is as follows: *CTC* never really joins multiple tables. Instead, it only conducts group-by queries on each key attribute and propagates the aggregates to the corresponding dimension table. When there are multiple dimension tables, propagating the aggregates is much cheaper than joining multiple tables and materializing a universal base table.

2.3.2 Computation of Local Iceberg Cubes

Local iceberg cubes on propagated dimension tables can be computed using any algorithms for iceberg cube computation, as reviewed in Section 2.2.2. For each iceberg cell c, we maintain the histogram of primary key values that the tuples in $cov(c)$ carry as the signature.

Definition 2.3. (Signature of iceberg cells in propagated dimension tables) *Let D be a dimension table and K be a primary key attribute such that K is used in the fact table as the foreign key referencing to D. For an iceberg cell c in D, the* **signature** *of c, denoted as $c.sig$, is the set of primary key values (i.e., values in K) that appear in the tuples in $cov(c)$ in D.*

Example 2.6 (Computing local iceberg cube). Let us compute the iceberg cube on propagated dimension table PD_2 (Figure 2.8(b)) with respect to condition $C \equiv \texttt{COUNT}(c) \geq 2$. Here, we use an adaption of algorithm *BUC* (Beyer & Ramakrishnan, 1999).

First, we sort the tuples in PD_2 in attribute E using counting sort by at most 2 scans of the table. The sum of counts of tuples having e_1 is 1. Thus, $(e_1,*,*,*)$ and any of its descendants cannot satisfy the condition and cannot be an iceberg cell.

Since the sum of counts of tuples having e_2 is 2, cell $(e2,*,*,*)$:2 is an iceberg cell. Moreover, since there is only tuple (e_2, f_1, g_2, h_2):2 in PD_2 has value e_2, we can enumerate all descendants of $(e2,*,*,*)$, namely, $(e_2, f_1,*,*)$, $(e_2, f_1, g_2,*)$, (e_2, f_1, g_2, h_2), $(e_2, f_1,*, h_2)$, $(e_2,*, g_2,*)$, $(e_2,*, g_2, h_2)$ and $(e_2,*,*, h_2)$. Their counts must also be 2 and thus are iceberg cells.

As can be observed here, one advantage of computing iceberg cubes on propagated dimension tables is that one tuple in the propagated dimension table may summarize multiple tuples in the corresponding projection of the universal base table. Thus, we reduce the number of tuples in the computation.

Similar to the case of e_1, we can find that $(e_3,*,*,*)$ and $(e_4,*,*,*)$, as well as their descendants, cannot be iceberg cells.

Then, we move to the next attribute, F. By sorting PD_2 in F, we find iceberg cells $(*, f_1,*,*)$ with count 4. We also record the set of primary key values $\{e_1, e_2, e_3\}$ that the tuples having f_1

carry. This is called the *signature* of the iceberg cell. It will be used in the future to derive global iceberg cells. To maintain the signature, we can use a vector of m bits for every iceberg cell, where m is the number of distinct values appearing in attribute E (the primary key attribute) in table PD_2.

To find iceberg cells among descendants of $(*, f_1, *, *)$, we sort the tuples in $cov(*, f_1, *, *)$ on attribute G. Recursively, we find iceberg cells $(*, f_1, g_1, *)$:2 with signature $\{e_1, e_3\}$, $(*, f_1, g_2, *)$:2 and $(*, f_1, g_2, h_2)$:2 both with signature $\{e_2\}$. By sorting $cov(*, f_1, *, *)$ on attribute H, we find iceberg cell $(*, f_1, *, h_2)$:3 with signature $\{e_2, e_3\}$.

The remaining local iceberg cells can be computed similarly. □

Based on Example 2.6, we formulate the notation of signature, which will be used to derive global iceberg cells from local ones.

Clearly, to maintain the signatures in D, we only need m bits, where m is the number of distinct values in K that appear in the fact table. m is at most the number of tuples in D, and no more than the cardinality of K.

The algorithm of computing local iceberg cubes on propagated dimension tables is largely the same as the existing iceberg cube computation algorithms. The only adaption is that the signatures of the iceberg cells should be maintained by a bitmap vector attached to every iceberg cell.

Lemma 3 (Computing local iceberg cubes). *In a star schema $(F, D_1, \ldots, D_n)$, where M is the measure attribute in F, let $B = F \bowtie D_1 \bowtie \cdots \bowtie D_n$. For any dimension table D_i $(1 \leq i \leq n)$, the iceberg cube from the propagated dimension table PD_i is identical to the iceberg cube from $\sigma_{D_i \cup \{M\}}(B)$.*

Proof. Proof Sketch The propagated dimension table PD_i can be viewed as a summarization of the projection $\sigma_{D_i \cup \{M\}}(B)$, where the tuples sharing the same key value in K_i are summarized by the extended measure attribute value in PD_i. □

An advantage of computing iceberg cubes from the propagated dimension table PD_i is that *it often has (many) fewer tuples* than the universal base table. In real applications, a fact table

may easily have millions of tuples, but the cardinality of a dimension may be in tens.

2.3.3 Computation of Global Iceberg Cubes

The set of global iceberg cells can be divided into two exclusive subsets: the ones having some non-$*$ values on the dimension attributes in the fact table, and the ones whose projections on the fact table are $(*,\ldots,*)$. We handle them separately.

Example 2.7 (Iceberg cell in fact table and beyond). Let us compute the iceberg cells from our running example data warehouse *DW* (Figure 2.3) with respect to condition $C \equiv \mathsf{COUNT}(c) \geq 2$. In this example, we consider the iceberg cells that contain some non-$*$ values in the dimension attributes in fact table *Fact*.

To find such iceberg cells, we start from applying an iceberg cube computing algorithm, such as *BUC* (Beyer & Ramakrishnan, 1999), to the fact table.

For example, we find $(a_1,*,*) : 2$ is an iceberg cell in the fact table. In the cover of $(a_1,*,*)$ (i.e., the first and the fourth tuples in Figure 2.3(b)), b_1 appears in attribute *B*, which references to dimension table D_1. Thus, for any local iceberg cell c in PD_1 whose signature contains b_1, such as $(b_1,*,*)$, $(*,c_1,*)$, and $(*,c_1,d_1)$, the "join"[2] of $(a_1,*,*)$ and c, such as $(a_1,b_1,*,*,*,*,*)$, $(a_1,*,c_1,*,*,*,*,*)$ and $(a_1,*,c_1,d_1,*,*,*,*)$, must be a global iceberg cell of count 2 (yielding to the measure of the iceberg cell in the fact table).

For iceberg cell $(a_1,*,c_1,*,*,*,*,*)$, e_1 and e_2 appear in attribute E, which reference to dimension table D_2. Thus, for any local iceberg cell c in PD_2 whose signature contains e_1 or e_2, such as $(*,f_1,*,*)$, can be a global iceberg cell, if the overlap of the signatures can lead to an aggregate value satisfying the iceberg condition. Then, we can further join them to get iceberg cell $(a_1,*,c_1,*,*,f_1,*,*)$.

It can be verified that, in such a recursive way, we can find all the global iceberg cells that contain some values in the attributes in fact table *Fact*. □

We formulate the operation of joining two aggregate cells.

[2] We will define the operation precisely soon.

1: apply an iceberg cube computation algorithm to compute the iceberg cells in the fact table
2: once an iceberg cell in the fact table is found search iceberg cells c' in the dimension tables by signatures such that $c \bowtie c'$ is a global iceberg cell
3: use $c = c \bowtie c'$ to conduct recursive search in other dimension tables until no iceberg cells in the dimension tables can be joined

Figure 2.9: Computing global iceberg cells containing some non-$*$ values in the attributes in fact table.

Definition 2.4 (Join of aggregate cells). *Let c_1 and c_2 be aggregate cells on tables T_1 and T_2, respectively, such that if T_1 and T_2 have any common attribute then c_1 and c_2 have the same value in every such common attribute. The* **join** *of c_1 and c_2, denoted as $c_1 \bowtie c_2$, is the tuple c such that (1) for any attribute A that c_1 has a non-$*$ value, c has the same value as c_1 on A; (2) for any attribute B that c_2 has a non-$*$ value, c has the same value as c_2 on B; (3) c has value $*$ in all other attributes.*

It is easy to show that the join operation has the commutative and associative properties.

Lemma 4 (Properties of join). *Let c_1, c_2 and c_3 are aggregate cells on T_1, T_2 and T_3, respectively. Then, $c_1 \bowtie c_2 = c_2 \bowtie c_1$ and $(c_1 \bowtie c_2) \bowtie c_3 = c_1 \bowtie (c_2 \bowtie c_3)$.* □

In general, once an iceberg cell c is found from the fact table, we can extract $c.sig$, the signature of c, and search the iceberg cells in the dimension tables whose signatures have overlap with $c.sig$. Suppose iceberg cell c' in a dimension table is found and $c.sig \cap c'.sig = \{k_1, \ldots, k_l\}$, i.e., k_1, ..., k_l are the keys in both signatures. Then, whether $c \bowtie c'$ is a global iceberg cell is determined by the aggregate on the tuples in $cov(c)$ having values k_1, ..., or k_l. This can be easily derived from the fact table.

The algorithm of computing global iceberg cells involving attributes in the fact table is summarized in Figure 2.9. As can be seen, we never need to join the fact table with any dimension tables to generate a global iceberg cell. Instead, we join the local iceberg cells based on the signatures. Recall that since we maintain the signatures using bitmap vectors, the matching of signatures is efficient. To facilitate matching, we also index the iceberg cells in the dimension tables by their signatures.

Another advantage of the algorithm is that, a local iceberg cell is found only once but is

used many times to join with other local iceberg cells to form global ones. If we compute the global iceberg cells from the universal base table, we may have to search the same portion of the universal base table for the (local) iceberg cell many times for different global iceberg cells. The cross table algorithm eliminates the redundancy in the computation.

Now, let us consider how to compute the global iceberg cells that have no non-$*$ value in attributes in the fact table.

Example 2.8 (Joining local iceberg cells). We consider how to compute the global iceberg cells in data warehouse *DW* (Figure 2.3) that do not contain any non-$*$ value in attributes in the fact table. Those global iceberg cells can be divided into two subsets: (1) the descendants of some local iceberg cells in PD_1, and (2) the descendants of some local iceberg cells in PD_2 but not descendant of any local iceberg cells in PD_1. In both cases, we only consider the cells that do not contain any non-$*$ value in the key attributes, otherwise, they are taken care by Example 2.7 and the algorithm in Figure 2.9.

To find the first subset, we consider the local iceberg cells in PD_1 one by one. For example, $(*, c_1, *)$ is a local iceberg cell in PD_1 with signature $\{b_1, b_2, b_3\}$. To find the local iceberg cells in PD_2 that can be joined with $(*, c_1, *)$ to form a global iceberg cell, we should collect all the tuples in the fact table that contain either b_1, b_2 or b_3, and find their signature on attribute E.

Clearly, to derive the signature on attribute E for a local iceberg cell in table PD_1 by collecting the tuples in the fact table is inefficient, since we have to scan the fact table once for each local iceberg cell. To tackle the problem, we build an H-tree (Han et al., 2001) using only the foreign key attributes in the fact table, as shown in Figure 2.10.

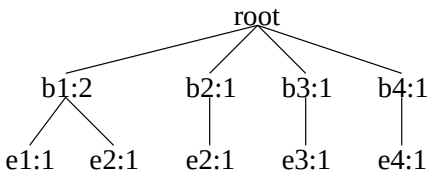


Figure 2.10: The H-tree for foreign key attribute values.

With the H-tree, for a given signature on attribute *B*, it is efficient to retrieve the corresponding signature on attribute E. For example, for $(*, c_1, *)$, its signature (on *B*) is $\{b_1, b_2, b_3\}$. From

the H-tree, we can retrieve its signature on E is $\{e_1, e_2, e_3\}$, i.e., the union of the nodes at level E that are descendants of b_1, b_2 or b_3.

Then, we can search the iceberg cells in dimension table PD_2. For example, iceberg cell $(*, *, g_1*)$ in dimension table PD_2 has signature $\{e_1, e_3, e_4\}$. The intersection of the two signatures is $\{e_1, e_3\}$. From the H-tree, we know that the total aggregate of tuples having e_1 or e_3 and b_1, b_2 or b_3 is 2 (the sum of the first and the fourth leaf nodes in the H-tree). Thus, the two iceberg cells can be joined and $(*, *, c_1, *, *, *, g_1, *)$ is a global iceberg cell.

Moreover, if we have more than 2 foreign key attributes, once all the global iceberg cells that are descendants of local iceberg cells in dimension table PD_1 are computed, the level of attribute B in the H-tree can be removed and the remaining sub-trees can be collapsed according to the next attribute, E. That will further reduce the tree size and search cost.

The second subset of global iceberg cells, i.e., the ones that are descendants of some local iceberg cells in PD_2, but not of PD_1, are exactly $(*, *, *,) \bowtie c$, where c is a local iceberg cell in PD_2. □

Please note that, in general, the space complexity of the H-tree in *CTC* is $O(kn)$, where k is the number of dimension tables and n is the number of tuples in the fact table. In many cases, the H-tree is smaller than the fact table and much smaller than the universal base table. The signatures of local iceberg cells can be stored on disk and do not have to be maintained in main memory.

The algorithm is summarized in Figure 2.11. Again, we use the local iceberg cells to generate the global ones. The matching is based on the signatures and an H-tree. This avoids the redundant searches on some common parts of the universal base table many times.

2.4 Experimental Results

In this section, we report an extensive performance study on computing iceberg cubes from data warehouses in star schema, using synthetic data sets. All the experiments are conducted on a Dell Latitude C640 laptop computer with a 2.0 GHz Pentium 4 processor, 20 G hard drive, and

1: build an H-tree on the foreign key attributes in the fact table;
optimization: only the key attributes a_i should be considered if $(*,\ldots,a_i,\ldots,*)$ is an iceberg cell
2: let $D_1, \ldots, D_n$ be the n dimension tables;
3: **for** $i = 1$ to n **do**
4: **for** each local iceberg cell c in dimension table D_i **do**
5: recursive search local iceberg cells in $D_{i+1}, \ldots, D_n$ that can be joined to form descendants of c and are global iceberg cells;
6: **end for**
7: remove the level of D_i in the H-tree
8: **end for**

Figure 2.11: Joining local iceberg cells in dimension tables to form global ones.

512 MB main memory, running the Microsoft Windows XP operating system.

We compare two algorithms: *BUC* (Beyer & Ramakrishnan, 1999) and *CTC*. Both algorithms are implemented in C++.

2.4.1 The Synthetic Data

Data Generator and Settings

We generate the synthetic data sets following the Zipf distribution. Our data generator takes the following parameters to generate the data sets.

- To generate the fact table, the data generator needs the number of dimensions, the number of tuples, and the cardinality in each dimension in the fact table. By default, the fact table has 5 dimensions, 1 million tuples and the cardinality of each dimension is set to 10.

- To generate dimension tables, the data generator needs the number of dimension tables and the number of attributes in each dimension table. Please note that the number of tuples in a dimension table is equal to the cardinality in the corresponding dimension in the fact table. By default, we set 3 dimension tables, and each dimension table has 3 attributes.

- The Zipf factor. We assume that all the data in the data warehouse follows the same distribution. By default, the Zipf factor is set to 1.0.

In a data warehouse generated by the above data generator, if there are n dimensions in the fact table and k dimension tables $(n \geq k)$, and there are l attributes in each dimension table, then the universal base table has $(l \cdot k + (n-k))$ dimensions. Thus, by default, the data warehouse generated by the data generator has $3 \times 3 + (5-3) = 11$ dimensions.

In all our experiments, we use aggregate function `count()`. Therefore, the domain, cardinality and distribution on the measure attribute have no effect on the experimental results. By default, we set the iceberg condition to "`COUNT`$(*) \geq$ number of tuples in fact table $\times 5\%$".

In all our experiments, the runtime of *CTC* is the elapsing time that *CTC* computes iceberg cube from multiple tables, including the CPU time and I/O time. However, the runtime of *BUC* is only the time that *BUC computes iceberg cube from the universal base table*, including the CPU time and I/O time. That is, *the time of deriving the universal table is not counted in the BUC runtime*. The rationale is that a universal base table can be computed once and used multiple times. To achieve this, *BUC* is always fed with a universal base table. We believe that such a setting is fair for both algorithms and does not bias towards *CTC*.

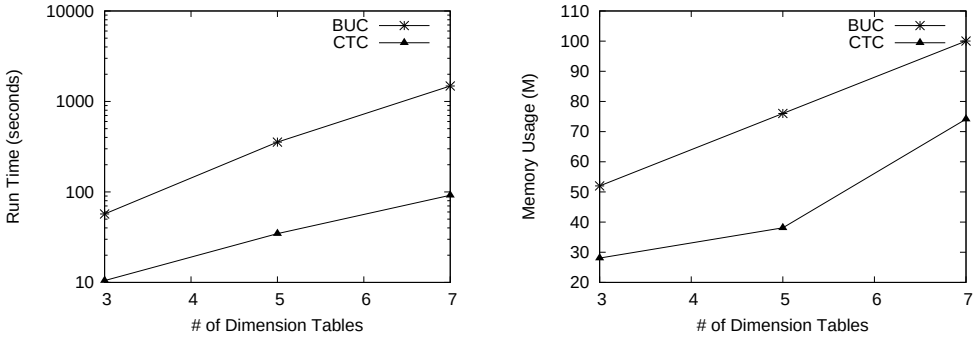
To simplify the comparison, we assume that the universal base table can be held into main memory in our experiments. When the universal base table cannot be held into main memory, the performance of *BUC* will be degraded substantially. *CTC* does not need to store all the tables in main memory. Instead, it loads tables one by one. The local iceberg cells can be indexed and stored on disk. One major consumption of main memory in *CTC* is to store the H-tree for the fact table. As shown before, the H-tree is often smaller than the fact table and much smaller than the universal base table. When the H-tree is too large to fit into main memory, the disk management techniques as discussed in (Han et al., 2001) and also the techniques for disk-based *BUC* can be applied.

Scalability w.r.t. Factors of Dimension Tables

The major feature distinguishing *CTC* from other iceberg cube algorithms is that it can compute iceberg cubes from multiple tables without joining them. In this subsection, we test the performance of *CTC* on mining data warehouses in star schema with different number of dimension

tables and different number of attributes in each dimension table.

To test the scalability with respect to the number of dimension tables, we vary the number of dimension tables from 3 to 7, set the number of dimensions in the fact table to the number of dimension tables plus three (i.e., we put 3 dimensions in the fact table that do not have dimension tables), and adopt the default values for other parameters. We test the runtime and main memory usage of *BUC* and *CTC*, which are shown in Figures 2.12(a) and (b), respectively. Please note that in Figure 2.12(a), the runtime (Y-axis) is plotted in logarithmic scale.



(a) Runtime (b) Main memory usage

Figure 2.12: Scalability with respect to number of dimension tables.

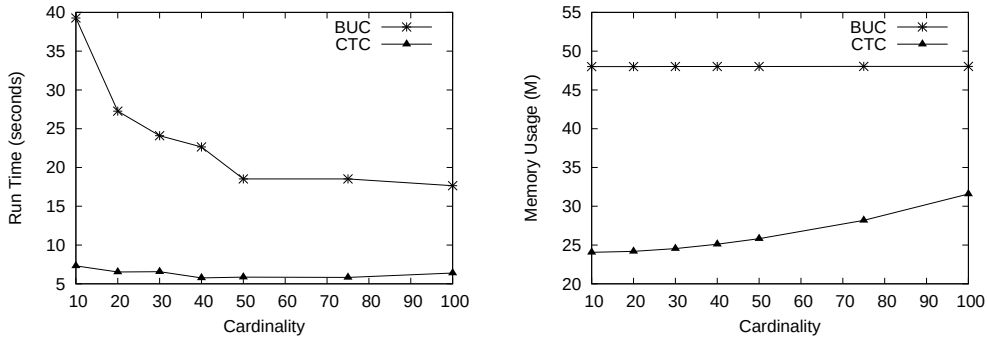
When the number of dimension tables goes up from 3 to 7, the total number of dimensions in the universal base table goes up from 12 to 24. Clearly, *CTC* is more efficient in both runtime and memory usage than *BUC*. As the number of dimension tables goes up, the number of dimensions in the universal table goes up, too. The runtime of *BUC* goes up dramatically. The runtime of *CTC* also goes up, but in a much more moderate trend. That is because using the local iceberg cells in dimension tables to derive the global ones is more efficient than computing from the universal base table, and thus is less sensitive to the increase of the number of total dimensions.

In terms of main memory usage, both algorithms use more memory as the number of dimensions goes up. When there are more dimension tables and dimensions, *CTC* has a taller H-tree and needs more space to store the signatures for the local iceberg cells.

We also test the scalability with respect to the number attributes in each dimension table. The trends are similar to the cases on the number of dimension tables (Figure 2.12).

Scalability w.r.t. Global Factors

There are several global factors that affect the data distribution in a data warehouse and the performance of iceberg cube computation, including (1) the cardinality of each dimension; (2) the Zipf factor; (3) the number of non-foreign key dimensions in the fact table; (4) the iceberg condition threshold; and (5) the number of tuples in the fact table. We test the effects of these factors on the scalability of both *CTC* and *BUC*.

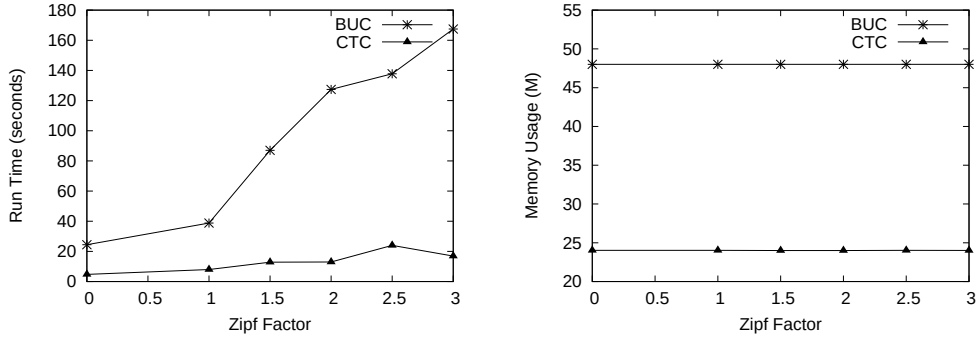


(a) Runtime (b) Main memory usage

Figure 2.13: Scalability with respect to cardinality in each dimension.

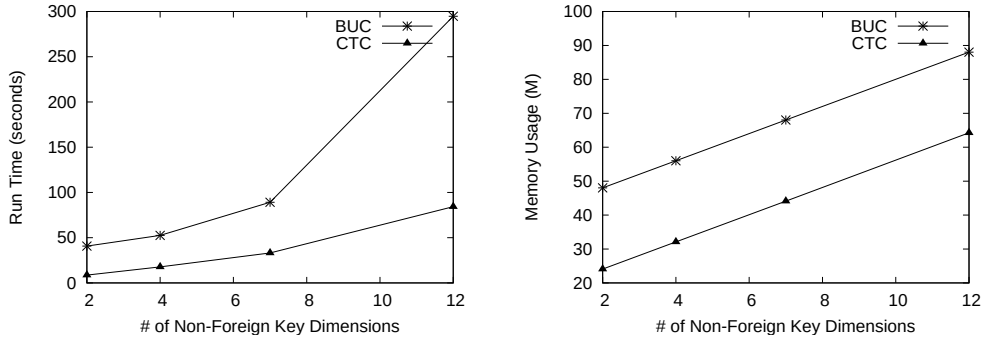
Figures 2.13(a) and (b) show the runtime and main memory usage of *CTC* and *BUC* with respect to the cardinality in each dimension, respectively. The other parameters are set to default. When the cardinality increases, the data set becomes sparser, that is, the number of iceberg cells decreases. Therefore, the runtime of *BUC* decreases. The runtime of *CTC* also decreases slightly. In terms of main memory usage, *BUC* is stable since it mainly holds the complete universal base table in main memory. *CTC* needs more memory to hold a larger H-tree as the cardinality goes up, since the average fan-out of each node in the tree increases. However, in a sparse data set, the number of one dimensional iceberg cells (i.e., the iceberg cells having only one non-$*$ value) also decrease. That slows down the increase of the size of H-tree.

In Figure 2.14, we set the Zipf factor in the range of 0 to 3.0 and adopt the default values for other parameters to test the runtime and main memory usage of *BUC* and *CTC* with respect to Zipf factor. With a larger Zipf factor, the data is more skewed and thus there are more iceberg cells. In sequel, the runtime of both *BUC* and *CTC* increase in general, but *CTC* is clearly more scalable. *BUC* and *CTC* are stable in main memory usage.



(a) Runtime (b) Main memory usage

Figure 2.14: Scalability with respect to Zipf factor.

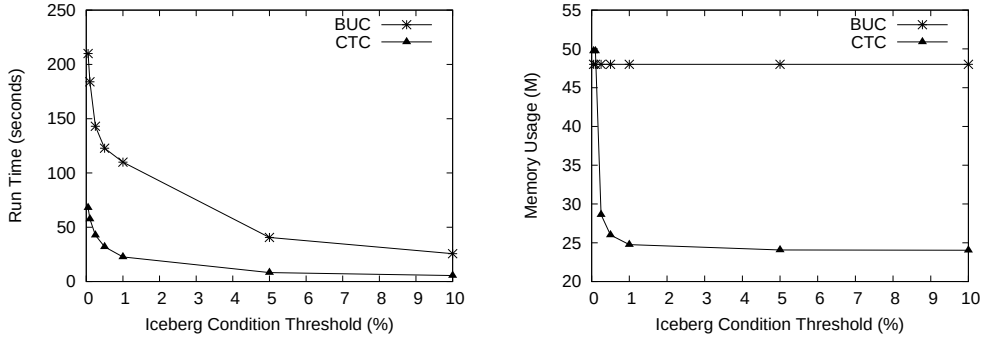


(a) Runtime (b) Main memory usage

Figure 2.15: Scalability with respect to number of non-foreign key dimensions.

In real applications, some dimensions may not have dimension tables. We call them *non-foreign key dimensions*. To test the effect of the number of non-foreign key dimensions in the fact table, we set the number of dimensions in the fact table from 5 to 15 and adopt the default values for other parameters. In such a setting, the number of non-foreign key dimensions in the fact table varies from 2 to 12. The experimental results are shown in Figure 2.15. As can be seen, the runtime of *BUC* goes up dramatically as the number of non-foreign key dimensions goes up, and the runtime of *CTC* is more scalable. The main memory usages of both *BUC* and *CTC* are linear to the increase of number of non-foreign key dimensions.

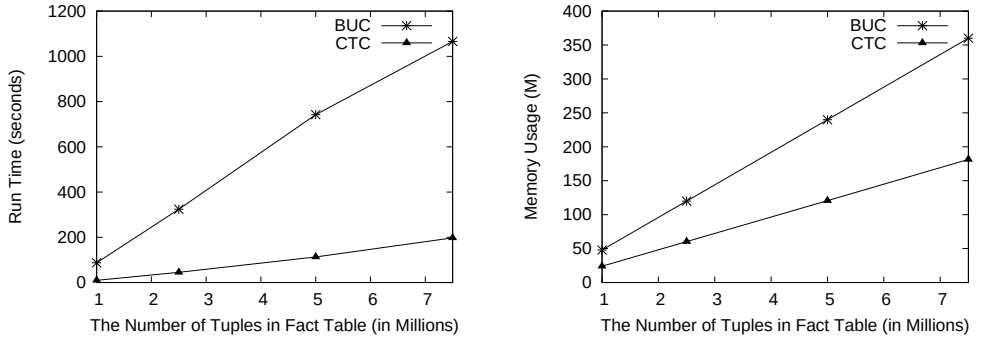
In order to test the scalability of *BUC* and *CTC* with respect to the iceberg condition, we set the parameters of data warehouse to default values, and vary the threshold v in the iceberg condition "$\texttt{COUNT}(*) \geq v$" from $100,000$ (i.e., 10%) to 500 (i.e., 0.05%). The runtime and main memory usage of *BUC* and *CTC* are shown in Figures 2.16(a) and (b), respectively.



(a) Runtime (b) Main memory usage

Figure 2.16: Scalability with respect to iceberg condition threshold.

When the threshold goes down, the number of iceberg cells increases dramatically. The runtime of both *BUC* and *CTC* increase accordingly. *CTC* is consistently 3 to 5 times faster. The main memory usage of *BUC* is stable, since it only keeps the universal base table in main memory. When the threshold is not very low, *CTC* uses much less main memory than *BUC*. When the threshold is very low, the main memory usage of *CTC* goes up. The reason is that, in our current implementation, we load the local iceberg cells and indexes into main memory before they are joined. When the threshold is very low, the number of local iceberg cells can be many times larger than the number of tuples in the universal base table.



(a) Runtime (b) Main memory usage

Figure 2.17: Scalability with respect to number of tuples in the fact table.

Last, we test the scalability of *BUC* and *CTC* on the number of tuples in the fact table. We vary the number of tuples in the fact table from 1 million to 7.5 million, and set the other parameters to default. The runtime and the main memory usage of *BUC* and *CTC* are shown in Figures 2.17(a) and (b), respectively. Clearly, both algorithms are linearly scalable in both

runtime and main memory usage, and *CTC* is faster and more efficient in main memory usage.

2.4.2 The Real Data and Setting

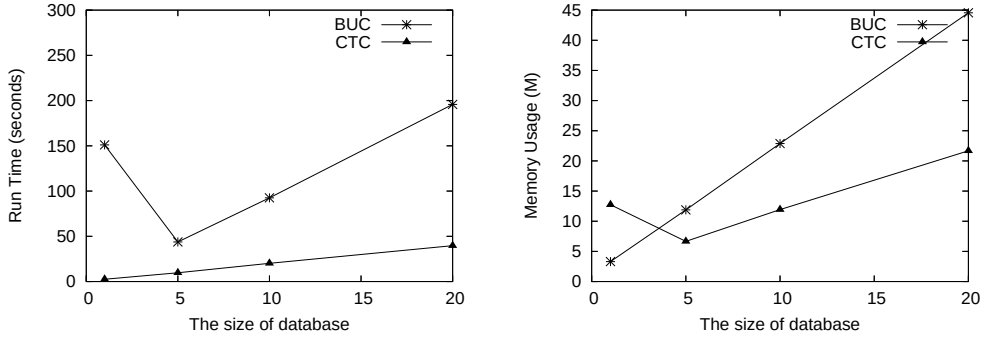
In this section, we report the experimental results on TPC-H (TPC, 1998). TPC-H is an ad-hoc, decision support benchmark. The schema of datasets of TPC-H is the constellation schema. In order to experiment with *CTC* on TPC-H, we changed the schema into a star schema by joining dimension tables with relationships. The initially created tables from TPC-H consist of one fact table and seven dimension tables in the constellation schema. The dimension tables are joined and then we have one fact table and four dimension tables in the star schema.

We use 10 MB size of total database population in TPC-H as default size of data set. The number of dimensions of fact table is 16 and the sum of the number of dimensions of four dimension tables is 46. The size of fact table in the database is around 68 % of the total database population by default setting of TPC-H. We set the iceberg condition v to 10 % of $COUNT(*)$ by default.

Scalability w.r.t. Size of Data Sets

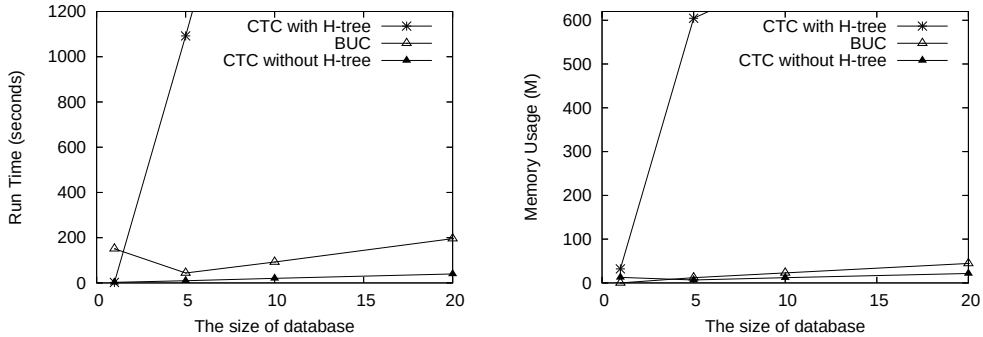
In this section, we test the performance of *CTC* in the different total size of data sets. We change the total size of data sets from 1 MB to 20 MB, whose the size of fact table is 68 % of the total size of data sets. The runtime and main memory usage of *CTC* and *BUC* are shown in Figure 2.18(a) and (b), respectively. Please note that we uses *CTC* not using Htree and we will compare *CTC* using Htree with *CTC* not using Htree soon.

Since *CTC* computes local iceberg cubes in each dimension tables avoiding duplicate computation in the fact table, the runtime of *CTC* is more efficient than *BUC*. We note that the number of iceberg cells is approximately 30 K at the 1 MB data set and 1K at the 5 MB, 10 MB, and 20 MB data sets. Since the number of iceberg cells at 1 MB data set is extraordinarily huge, *BUC* suffers in the runtime at the size of 1 MB of data set. *CTC* is more efficient in the runtime than *BUC* in the situation where the number of iceberg cells are high, whereas *CTC* suffers in



(a) Runtime (b) Main memory usage

Figure 2.18: Scalability with respect to size of data sets.



(a) Runtime (b) Main memory usage

Figure 2.19: Scalability with respect to size of data sets.

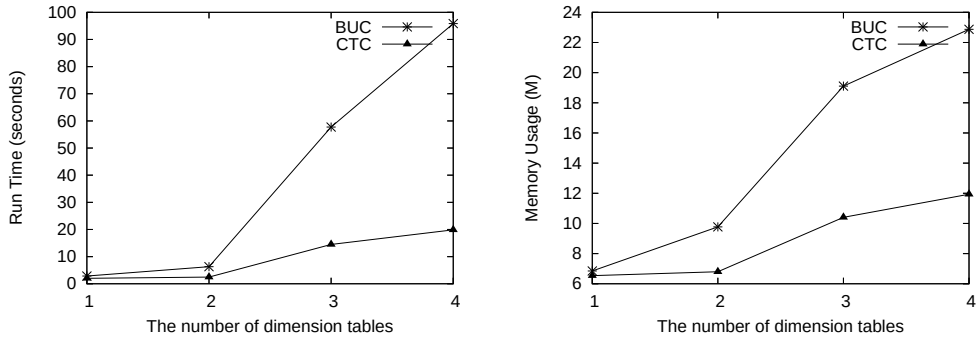
the memory usage more than *BUC* since *CTC* needs to store all local iceberg cubes with the signatures.

CTC uses H-tree to join the global iceberg cells efficiently. However, we find out that to use H-tree in *CTC* is not always efficient in all data sets. Figure 2.19 shows the performance of *CTC* with H-tree, *CTC* without H-tree, and *BUC* in 10 MB data set and default iceberg condition. We can see that *CTC* with H-tree is slower than *CTC* without H-tree and even *BUC*. Since the cardinality of the dimensions referencing to dimension tables in the fact table is much larger than the synthetic data set whose cardinality is 10 as default in the Section 2.4.1, H-tree for dimensions referencing to dimension tables in the fact table is bush so that the cost of run time and memory usage to handle H-tree is more expensive than to deal with the dimensions of foreign keys in the fact table itself.

Scalability w.r.t. Number of Dimension Tables

We vary the number of dimension tables from 1 to 4 to show the performance of computing the local iceberg cells in each dimension table. Each dimension table in the data has different numbers of dimension which are 9, 12, 21, and 4, respectively. The Figure 2.20(a) and (b) show the runtime and memory usage of *BUC* and *CTC*.

In the both of runtime and memory usage, *CTC* is more efficient than *BUC*. Since the difference between 2 and 3 on the number of dimensions is the highest one, the runtime and memory usage go up more sharply than others. The runtime of *BUC* increases faster than *CTC* since *BUC* suffers more with the increasing number of dimensions.



(a) Runtime (b) Main memory usage

Figure 2.20: Scalability with respect to number of dimension tables.

2.4.3 Summary

By the extensive performance study using synthetic and real data sets, we show that *CTC* is consistently more efficient and more scalable than *BUC*. The performance of *BUC* in our experiments is consistent in trend with the results reported in (Beyer & Ramakrishnan, 1999).

CTC is a general framework for computing iceberg cube from data warehouses directly, without joining the tables. It can use any iceberg cube computation algorithm to compute local iceberg cells on dimension tables and fact table. In our experiments, we use *BUC*. We believe that the observations from the performance study are general for any other iceberg cube computation methods. In order words, the advantage of computing global iceberg cube by local

iceberg cells still retains, no matter which iceberg cube computation approach is adopted.

2.5 Discussion

In the previous sections, we develop *CTC* that computes iceberg cube from star schema without computing the universal base table. The central idea in *CTC* is that it uses the information of foreign key dimension values in the fact table to join local iceberg cells in various dimension tables. This idea can be generalized and extended to handle other kinds of schemas for data warehouses. We use snowflake schema as an example.

The snowflake schema is a variant of the star schema model, where some dimension tables are further normalized, and thus further splitting the data into additional tables. The major advantage of the snowflake schema is that the dimension tables of the snowflake model are stored in normalized form to reduce redundancies. An example is shown in Figure 2.21. As can be seen, the dimension tables are in more than 1 level.

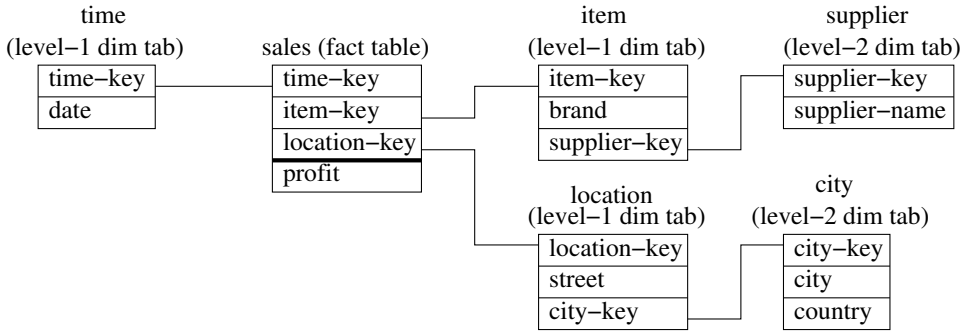


Figure 2.21: Snowflake schema: an example.

To extend *CTC* to handle snowflake schema, we can first propagate the measure information from the fact table to the level-1 dimension tables, and then from the level-1 dimension tables to the level-2 dimension tables, and so on.

As the second step, the local iceberg cells can be computed for each table. To join the local iceberg cells and derive the global ones, we build an H-tree for the fact table and each intermediate dimension table (i.e., a level-i dimension table containing some foreign key attribute referencing to level-$(i+1)$ dimension table, such as tables `item` and `location` in

Figure 2.21). As the last step, the H-trees can be combined as a global H-tree and the local iceberg cells can be joined.

Similarly, we can also handle some other complicated cases, such as fact constellation schema.

Chapter 3

Online answering ad-hoc aggregate queries on data streams

3.1 Preliminaries

A data warehouse materializes a large set of aggregates from a given base table. Various aggregate queries(OLAP queries) can be answered online by proper indexes in a data warehouse.

In general, the complete set of aggregate cells on a multidimensional base table can be huge. For example, if a base table has 20 dimensions and the cardinality of each dimension is 10, then the total number of aggregate cells is $11^{20} \approx 6.7 \times 10^{20}$. Even if only on average one out of 10^{10} aggregate cells is non-empty (i.e., covering some tuple(s) in the base table), the total number of non-empty aggregate cells still can be up to 6.7×10^{10}! Thus, computing and/or materializing a complete data cube is often expensive in both time and space, and hard to be online.

Recently, several important applications see the strong demands of online answering *ad hoc aggregate queries* over fast data streams. In this chapter, we are particularly interested in the applications where the *accurate instead of approximate answers* to the queries are mandatory.

For example, trading in futures market is often a high-risk and high-return business in many financial institutions. Transactional data and market data are collected timely. Dealers often raise various ad hoc aggregate queries about the data in recent periods, such as "*list the total*

transaction amounts and positions in the last 4 *hours, by financial products, counter parties, time-stamp (rounded to hour), mature date and their combinations.*" In those applications, it is required to *maintain the recent data in a sliding window,* and *provide accurate and online answers to ad hoc aggregate queries over the current sliding window*. As another example, a large sensor network is often deployed to monitor the hydraulic, hydrologic and water quality data from a large-scale river network. To dynamically monitor and model the transport of toxic contaminants or sediment, it is important to online answer various ad hoc aggregate queries about the data in recent periods, such as "*to see the effect of the heavy rain last night, list the density of toxic contaminants in the last* 24 *hours, by areas, branches, reservoirs, categories of toxic contaminants, time (rounded to hour) and their possible combinations*".

Many previous studies proposed approximate methods to monitor aggregates over very fast and high cardinality data streams, such as network traffic data streams where the speed of a data stream can be of gigabytes per second and the cardinality of the IP addresses is 2^{32}. In such situations, it is impossible to obtain accurate answers. Approximate answers usually provide sufficiently good insights. However, the target applications investigated in this chapter, such as the transactional data streams in business, are substantially different. First, accurate answers are mandatory in many business applications. This is particularly important for some business applications such as those in the financial industry. Second, the data streams studied here often are not extremely fast, and the cardinality of the data is not very huge. Instead, they have a manageable speed. For example, since the modern computers easily have gigabytes of main memory and typically the transactions in those applications will be in the scale of millions per day, it is reasonable to assume that the current sliding window of transactions can be held into main memory. Thus, it is possible to obtain accurate answers to ad hoc aggregate queries, even though the task is still challenging.

Given a data stream, what we want to do is that (1) to maintain the current sliding window of transactions in main memory, (2) to provide accurate answers to OLAP queries. Several approaches may be proposed to this task.

First, *Can traditional data warehousing techniques meet the requirement?* A traditional data

warehouse often updates in batch periodically, such as daily maintenance at nights or weekly maintenance during weekends. Such updates are often conducted offline. Online aggregate queries about the most recent data cannot be answered by the traditional data warehouses due to the delay of the incremental updates.

Second, *Then, can we maintain a materialized data cube over the sliding window?* Unfortunately, the size of a data cube is likely exponential to the dimensionality and much larger than the sliding window. Moreover, the cubing runtime is also exponential to the dimensionality and often requires multiple scans of the tuples in the sliding window or the intermediate results. However, in a typical data stream, each tuple can be seen only once, and the call-back operations can be very expensive. Thus, a data cube resulted from a reasonably large sliding window is usually too large in space and costly in time to be materialized and incrementally maintained online.

Unfortunately, the above two approaches have shortcomings to meet the requirements of the task. In order to meet the requirement of this take, *we materialize and incrementally maintain only a small subset of aggregates online by scanning the data stream only once, and still retain the high performance of online answering ad hoc aggregate queries.* In other words, we get a good tradeoff between efficiency of the query answering and the efficiency of indexing and maintenance (i.e., the size of index and the time of building and maintaining the index). In online answering aggregate queries, there are three factors needed to be considered, namely the space to store the aggregates, the time to create and maintain the aggregates, and the query answering time. While the existing static data cubing methods focus on reducing the last of them, *the goal of this chapter is to trade off the query answering time a little bit against the space and time of incremental maintenance*. Particularly, due to the inherent requirements of data stream processing, the data stream can be scanned only once, and the space to store the aggregates is highly desirable to be linear in the stream. In this chapter, we address the following challenges.
Challenge 1: *Can we avoid computing the complete cube but still retain the capability of answering various aggregate queries efficiently?*
Our contribution: We propose a solution that the transient segment (i.e., a sliding window)

of a data stream is maintained in an online data warehouse, which is enabled by the idea of materializing only the *prefix aggregate cells* and the *infix aggregate cells*. We show that they form just a small subset of the complete data cube, and the total number of prefix aggregate cells is *linear* to the number of tuples in the sliding window and the dimensionality. With such a small set of aggregates cells, many aggregate queries, including both point queries and range queries, still can be answered efficiently.

Challenge 2: *How can we compute, maintain and index the selected aggregates from a data stream?*

Our contribution: We devise a novel data structure, *prefix aggregate tree* (*PAT*), to store and index the prefix aggregate cells and the infix aggregate cells. The size of *PAT* is bounded. Algorithms are developed to construct and incrementally maintain *PAT*. Our experimental results indicate that *PAT* is efficient and scalable for fast and large data streams.

Challenge 3: *How can we answer various aggregate queries efficiently?*

Our contribution: We develop efficient algorithms to answer essential aggregate queries, including point queries and range queries. Infix links and the locality property of side-links of *PAT* enable various aggregate queries to be answered efficiently. An extensive performance study shows that the query answering is efficacious over large and fast data streams.

The remainder of this study is organized as follows. In the remainder of this Section 3.1, we describe the framework and review related work. The prefix aggregate tree structure as well as its construction and incremental maintenance are presented in Section 3.3. The query answering algorithms are developed in Section 3.4. The extensive experimental results are reported in Section 3.5.

3.1.1 The Framework

In this study, we model a *data stream* as an (endless) *base table* $S(T, D_1, \ldots, D_n, M)$, where T is an attribute of *time-stamps*, $D_1, \ldots, D_n$ are *n dimensions* in discrete domains, and M is the *measure*. For the sake of simplicity, we use positive integers starting from 1 as time-stamps.

In data stream processing, records are often collected in temporal order. Thus, it is reason-

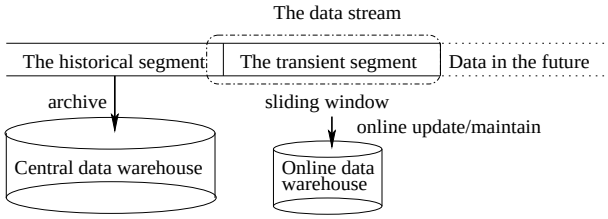


Figure 3.1: The framework of warehousing data streams.

able to assume that the tuples having time-stamp τ arrive before the ones having time-stamp $(\tau+1)$. Tuples having the same time-stamp may be in arbitrary order.

Traditional data warehouses can answer various aggregate queries efficiently. However, those data warehouses have to be incrementally maintained periodically and the maintenance is often offline. It is difficult to answer aggregate queries on the recent data that has not been loaded into the data warehouse in the last update.

To tackle this problem, it is natural to divide a data stream into two segments: the *historical segment* and the *transient segment*, as illustrated in Figure 3.1. Conceptually, the historical segment is the data arrived before the last update of the central data warehouse and thus has been archived. The transient segment, in turn, is the data that has not been archived in the central data warehouse, and should be updated and maintained online in an online data warehouse.

Such a framework of historical and transient segments appears in multiple applications and some prototype implementations of commercial databases. However, to the best of our knowledge, there exists no previous study on how to construct and maintain an online data warehouse for the transient segment.

Technically, should the online data warehouse store only the data in the transient segment? Consider the scenario that the central data warehouse is just updated. Then, the online data warehouse contains very little data and many aggregate queries about the recent data cannot be answered using the online data warehouse. To avoid this problem, *the online data warehouse should maintain the tuples whose time-stamps are in a sliding window of size* ω*, where* ω *is the length of the periodicity that the central data warehouse conducts an regular update.*

In other words, at instant t $(t \geq \omega)$, we assume that all the queries in the online data warehouse are about multi-dimensional aggregates of tuples falling in the sliding window of

$[t-\omega+1, t]$.

Aggregate functions can be used in the queries, such as `SUM`, `MIN`, `MAX`, `COUNT` and `AVG` in SQL. We consider the following two kinds of queries in this study.

- *Point queries*. At instant t, a point query is in the form of a *query cell* $(\tau, d_1, \ldots, d_n)$, where $t-\omega+1 \leq \tau \leq t$ or $\tau = *$, and $d_i \in D_i \cup \{*\}$. For example, consider a data stream of transaction records in an endless table *transaction (Time-stamp, Branch-id, Prod-id, Counter-party-id, Amount)* where *Branch-id*, *Prod-id* and *Counter-party-id* are the dimensions, and *Amount* is the measure. Suppose the sliding window is of size 24 hours. A point query may ask for "*the total amount of 'gold' at 10 am*", where the query cell is (*10am, *, gold, **). Here, the symbols "$*$" in dimensions *Branch-id* and *Counter-party-id* mean every transaction in any branch and with any counter-party counts.

  Particularly, when $\tau = *$, the aggregate over the whole sliding window is returned. As another example, query cell (**, Paris, *, **) stands for the total trading amount in Paris in the current sliding window, including all products and all customers.

- *Range queries*. A range query specifies ranges instead of a specific value in some dimensions. Thus, a range query may cover multiple query cells. For example, a range query may ask for "*the total amount of 'gold' and 'oil' in Paris and London in the current sliding window*", denoted as $(*, \{\text{Paris, London}\}, \{\text{gold, oil}\}, *)$.

  The answer to a range query is one aggregate over all the tuples falling in the range. For example, the above range query is answered by one total amount that covers all the transactions in the two cities and about the two products, in the last two hours.[1]

The above two kinds of OLAP queries are essential, though more complex queries can be raised. Many complex OLAP queries can be decomposed into a set of queries in the above two categories.

[1] Alternatively, a list of the aggregates of the query cells falling in the range can be returned. For example, the above range query may be answered by a list of 4 aggregates corresponding to the combinations of values in the ranges of dimension *Branch-id* and *Product*. The two forms of answers can be derived by similar techniques.

Now, the problem becomes *how to construct and incrementally maintain an online data warehouse and answer ad hoc aggregate queries.*

Problem statement. Given a data stream S and a size of sliding window ω. We want to construct and maintain an online data structure $W(t)$ so that, at any instant t in the sliding window ω, any point queries and range queries can be answered precisely and efficiently based on $W(t)$.

To be feasible for streaming data processing, $W(t)$ should satisfy the following two conditions.

1. The size of $W(t)$ is linear to the number of tuples in the current sliding window and the dimensionality; and

2. $W(t)$ can be constructed and maintained by scanning the tuples in the stream only once, and unnecessarily holding the sliding window in main memory.

W is called the *online data warehouse* of stream S. □

3.2 Related Work

(Chaudhuri & Dayal, 1997; Widom, 1995) are excellent overviews of the major technical progresses and research problems in data warehousing and OLAP. It has been well recognized that OLAP is more efficient if a data warehouse is used.

The data cube operator (Gray et al., 1997) is one of the most influential operators in OLAP. Many approaches have been proposed to compute data cubes efficiently from scratch (Beyer & Ramakrishnan, 1999; Ross & Srivastava, 1997; Ross & Zaman, 2000; Y. Zhao et al., 1997). In general, they speed up the cube computation by sharing partitions, sorts, or partial sorts for group-bys with common dimensions.

It is well recognized that the space requirements of data cubes in practice are often huge. Some studies investigate partial materialization of data cubes (Beyer & Ramakrishnan, 1999; Han et al., 2001; Harinarayan et al., 1996). Methods to compress data cubes are studied

in (Shanmugasundaram et al., 1999; Sismanis et al., 2002; W. Wang et al., 2002; Lakshmanann et al., 2002; Lakshmanan et al., 2003). Moreover, many studies (Barbara & Sullivan, 1997; Barbar & Wu, 2000; Vitter et al., 1998) investigate various approximation methods for data cubes.

How to implement and index data cubes efficiently is a critical problem. Cubetree (Roussopoulos et al., 1997) and Dwarf (Sismanis et al., 2002) are proposed by exploring the prefix and suffix sharing among dimension values of aggregate cells. Quotient cube (Lakshmanann et al., 2002) is a non-redundant compression of data cube by exploring the semantics of aggregate cells, and QC-tree (Lakshmanan et al., 2003) is an effective data structure to store and index quotient cube. As compressions of a data cube, they can be used to answer queries directly, and quotient cube can further support some advanced semantic OLAP operations, such as roll-up/drill-down browsing.

Many studies have been conducted on how to answer various queries effectively and efficiently using fully or partially materialized data cubes (Cohen et al., 1999; Johnson & Shasha, 1997; Levy et al., 1995; Mendelzon & Vaisman, 2000; Srivastava et al., 1996). To facilitate the query answering, various indexes have been proposed. Sarawagi (1997) provides an excellent survey on related methods. A data warehouse may need to be updated timely to reflect the changes of data. In 1990's, the maintenance of views in data warehouses was actively studied (Gupta et al., 1993; Mumick et al., 1994; Quass et al., 1996; Quass & Widom, 1997).

Recently, intensive research efforts have been invested in data stream processing, such as monitoring statistics over streams and query answering (Babu & Widom, 2001; Dobra et al., 2002; Datar et al., 2002; Gehrke et al., 2001) and multi-dimensional analysis (Chen et al., 2002). Please see (Babcock et al., 2002) for a comprehensive overview. While many of them focus on answering *continuous queries*, few of them consider answering *ad hoc queries* by warehousing data streams. Probably the work most related to this paper is done by Chen et al. (2002), where a linear-regression based approach is proposed to accumulate the multi-dimensional aggregates from a data stream, and a variation of the H-tree structure (Han et al., 2001) is used to materialize some selected roll-up/drill-down paths for OLAP. However, their method assumes that

the streaming data can be summarized effectively by linear regression and can only provide approximate answers to aggregate queries, and no efficient method is presented to answer various ad hoc aggregate queries in general. Moreover, the selected roll-up/drill-down paths are hard to determine. It is unclear how the H-tree structure can be stored and incrementally maintained effectively for data streams.

Particularly, this work is related to the research on mining frequent itemsets from data streams (Arasu & Manku, 2004; Chang & Lee, 2003; Cormode et al., 2003; Cormode & Muthukrishnan, 2003; Giannella et al., 2004; Karp et al., 2003; Teng et al., 2003; Yu et al., 2004). Basically, for a given stream of transactions, where a transaction is a set of items, the frequent itemset mining problem for data streams is to maintain the set of itemsets that appear at least $\Delta \cdot n$ times in the transactions seen so far, where Δ is a minimum support threshold, and n is the number of transactions seen so far. Some methods put weights to transactions, and the more recent transactions have heavier weights. Frequency can be viewed as a type of aggregates. However, all of the previous methods are approximate approaches. They cannot provide the exact answers, though some methods can provide different types of quality guarantees.

To the best of our knowledge, this is the first study on warehousing and indexing data in a sliding window over data stream and answering ad hoc aggregate queries accurately.

On the other hand, tree and prefix-tree structures have been frequently used in data mining and data warehousing indices, including cube forest (Johnson & Shasha, 1997), FP-tree (Han et al., 2004), H-tree (Han et al., 2001), Dwarf structure (Sismanis et al., 2002) and QC-tree (Lakshmanan et al., 2003). *PAT* is also a prefix tree data structure. We will further compare our approaches to several important previous studies in Section 3.3.2, after the major technical ideas of *PAT* are brought up.

3.3 Prefix Aggregate Tree (PAT)

In this section, we devise the prefix aggregate tree (*PAT*) data structure. We also develop algorithms to construct and incrementally maintain prefix aggregate tree. Hereafter, all aggregate

queries are ad hoc ones.

3.3.1 Data Structure

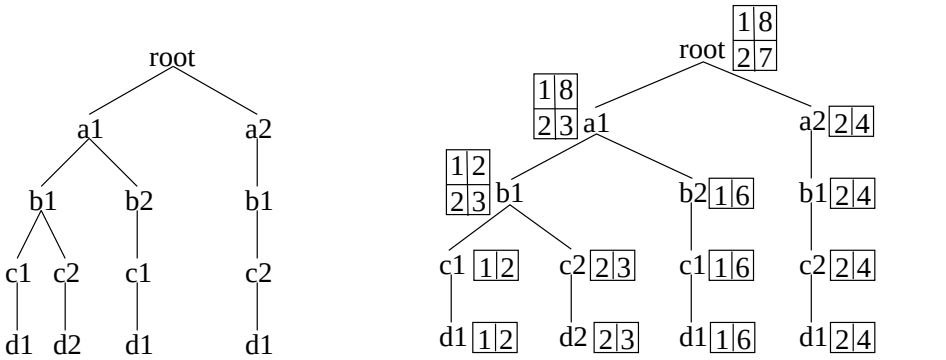
Consider a data stream $S(T, D_1, \ldots, D_n, M)$. Let the size of sliding window be ω. In order to answer any aggregate query about the data in the sliding window, we have to store the tuples in the sliding window. An intuitive way to store the tuples compactly is to use a prefix tree, as shown in the following example.

Example 3.1. Let the data stream as our running example be $S(T, A, B, C, D, M)$ where T and M are the attributes of time-stamps and the measure, respectively. The tuples at instants 1 and 2 are shown in Figure 3.2. Suppose aggregate function `SUM` is used, and the size of sliding window $\omega = 2$.

T	A	B	C	D	M
1	a_1	b_2	c_1	d_1	6
1	a_1	b_1	c_1	d_1	2
2	a_1	b_1	c_2	d_2	3
2	a_2	b_1	c_2	d_1	4

Figure 3.2: The tuples at instants 1 and 2 in stream $S(T, A, B, C, D, M)$.

If we ignore the time-stamps and measures, the tuples in the sliding window can be organized into a prefix tree, as shown in Figure 3.3(a).



(a) Storing tuples into a prefix tree. (b) Prefix tree with aggregate table at each node.

Figure 3.3: Archiving a data stream in a prefix tree.

In order to store the information about the time-stamps and measures, we can register the information at the tree nodes, as shown in Figure 3.3(b). Each tree node has an *aggregate table*, such that the time-stamps and the aggregates by instants are registered.

Clearly, the leaf nodes in the prefix tree record the tuples in the sliding window. Each internal node in the tree registers the aggregate of the tuples whose paths go through this node. For example, the node a_1 in the tree registers the aggregates of tuples having a_1 and stores them by instants. □

In a prefix tree, one node can be represented by the path from the root of the tree to the node. Hereafter, we will write a node as a string of dimension values, such as a_1, a_1b_1 and $a_1b_1c_2$.

In the prefix tree shown in Figure 3.3(b), an internal node registers the aggregates of tuples sharing the "prefix" from the root to the node. They are called prefix aggregate cells.

Definition 3.1 (Prefix aggregate cell). *Consider a data stream $S(T, D_1, \ldots, D_n, M)$, where T and M are attributes of time-stamps and measure, respectively. For any tuple, we always list the dimension values in the order of $D_1, \ldots, D_n$. Let S_t be the set of tuples in the current sliding window $[(t-\omega+1), t]$ of S. An aggregate cell $c = (\tau, d_1, \ldots, d_n)$ is a* prefix aggregate cell *if (1) there exists a k such that $1 \leq k \leq n$, $d_1, \ldots, d_k$ are not $*$ and $d_{k+1}, \ldots, d_n$ are all $*$; and (2) there exists at least one tuple $c' = (d'_1, \ldots, d'_n)$ in S_t such that $d_i = d'_i$ for $(1 \leq i \leq k)$.*

Theorem 3.1. *By storing only the prefix aggregate cells, any ad hoc aggregate queries about the current sliding window can be answered.*

Proof. Clearly, the answer to any ad hoc aggregate query about the current sliding window can be derived from the complete set of tuples in the window. Let us consider tuples in the current window. If a tuple t is unique in the current sliding window, t is (trivially) a prefix aggregate cell. If t is not unique, then there exists a prefix aggregate cell which has the same value as t on every dimension. In other words, the set of prefix aggregate cells covers all tuples in the current sliding window. □

Theorem 3.2 (Numbers of aggregate cells/prefix aggregate cells). *Given a base table of n dimensions and k tuples, let n_{aggr} and n_p be the number of aggregate cells and that of prefix aggregate cells, respectively. Then,* $2^n \leq n_{aggr} \leq (k \cdot (2^n - 1) + 1)$ *and* $(n+1) \leq n_p \leq (k \cdot n + 1)$.

Proof. When tuples share some values in some dimensions, they share the corresponding aggregate cells. When all tuples in the base table have the same value on every dimension, n_{aggr} is minimized. When the k tuples do not share any common value in any dimension, each tuple leads to $(2^n - 1)$ unique aggregate cells, and all tuples share aggregate cell $(*, \ldots, *)$. Thus, n_{np} is maximized to $(k \cdot (2^n - 1) + 1)$.

When tuples share some prefixes, they share the corresponding prefix aggregate cells. When all tuples in the base table have the same value on every dimension, n_p is minimized. When the k tuples do not share any prefix, each tuple leads to n prefix aggregate cells, and all tuples share aggregate cell $(*, \ldots, *)$. Thus, n_{aggr} is maximized to $(k \cdot n + 1)$. □

Theorem 3.2 indicates that the number of prefix aggregate cells is, in the worst cases, linear to the number of tuples in the sliding window and the dimensionality, and thus is substantially smaller than that of all aggregate cells. It suggests that the set of prefix aggregate cells is a promising candidate of an online data warehouse for a data stream.

Given a prefix tree of the prefix aggregate cells, aggregate queries can be answered by browsing the tree and extracting the related tuples in the current sliding window. However, if the current sliding window is large and thus the prefix tree is also large, browsing a large tree may not be efficient. We should build some light-weight index in the tree to facilitate the search.

Let us consider how to derive the aggregate for $(*, b_1, *, *)$ from the prefix tree in Figure 3.3(b). To answer this query, we need to access all the tuples having value b_1 on dimension *B*. To facilitate the search, it is natural to introduce the *side links* that link all nodes carrying the same label together.

Can we add side links arbitrarily? Let us consider how to compute aggregate $(a_1, *, c_1, *)$ from the tree in Figure 3.3(b). To answer this query, we want to access the nodes carrying label c_1 in the subtree rooted at node a_1. That is, we want a *local* linked list of nodes having label c_1

in the subtree rooted at node a_1.

Clearly, maintaining multiple linked lists is not a good idea. Instead, we should construct a linked list that has the *locality property: in any subtree, the nodes carrying the same label should be linked consecutively.*

Moreover, if we can register the aggregate of all tuples having c_1 in the a_1-subtree, the query can be answered even faster. This information can be registered as the head of the sub-linked list of c_1 in the a_1-subtree.

To accommodate the above ideas, we can construct a linked list of all nodes having c_1 in the a_1-subtree, and set up a pointer to the head of the sub-linked list at node a_1. The corresponding aggregate, $(a_1, *, c_1, *)$ should also be stored and associated with the head of the linked list.

The above ideas can be generalized. For example, side links can be built in the prefix tree in Figure 3.3(b), resulting in a *prefix aggregate tree* structure, as shown in Figure 3.4.

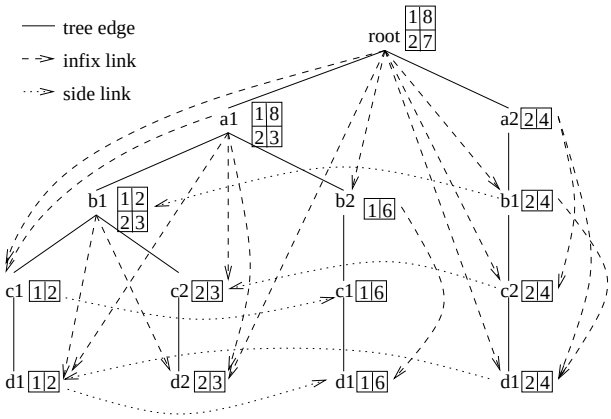


Figure 3.4: Prefix aggregate tree (the aggregate tables for infix links are omitted to make the graph easy to read).

Definition 3.2 (Prefix aggregate tree). *In general, given the current sliding window of a data stream, a* **prefix aggregate tree** *(*PAT *for short) is a prefix tree of the prefix aggregate cells in the window with the following two kinds of links.*

- **Side links**. *All nodes having the same dimension value as the label are linked together such that the locality property is hold: For any subtree, all the nodes carrying the same label in the subtree are linked consecutively.*

- **Infix links**. *At node $v = d_1 \cdots d_k$ $(1 \leq k \leq (n-2))$, for every dimension value $d_{i,j}$ on dimension D_i $((k+2) \leq i \leq n)$ that appears in the subtree rooted at v, all the nodes carrying label $d_{i,j}$ in the subtree rooted at v are linked consecutively by side-links. An* infix link *is built from v to the head of the sublist, and an aggregate table is stored and associated with the infix link recording the aggregates of $c = (d_1, \ldots, d_k, *, \ldots, *, d_{i,j}, *, \ldots, *)$. c is called an* infix aggregate cell.

Theorem 3.3. *Consider base table of n dimensions and the cardinality of each dimension is l. In the worst case, the number of infix aggregate cells (and thus the infix links in a PAT) is*

$$\sum_{i=1}^{n-2} l^i (n-i-1).$$

Proof. From the definition of *PAT*, the number of infix aggregate cells and that of infix links are identical. The worst case happens when every possible combination of dimension values appears in the base table, where the base table has l^n unique tuples. Each internal node in the *PAT* has l children. Thus we have the upper bound. In such a situation, there are $(l+1)^n$ aggregate cells. □

Theorem 3.2 and 3.3 shows that, in the worst case, the set of prefix aggregate cells and infix aggregate cells is still a small subset of all the aggregate cells. Practical data is usually skewed. As verified by our experimental results, the *PAT* is much smaller than the size of all the aggregate cells.

The size of a node in a *PAT* is regular. For a prefix aggregate cell $(d_1, \ldots, d_k, *, \ldots, *)$, the corresponding node in the tree is at the k-th level (the root node is at level 0), and stores the following pieces of information: (1) The aggregate table, which has 2 columns, the time-stamp and the aggregate, and at most ω records; (2) Pointers to up to l_{k+1} children, where l_{k+1} is the cardinality of dimension D_{k+1}; (3) At most $\sum_{i=k+2}^{n} l_i$ infix aggregate links and corresponding aggregate tables, where l_i is the cardinality of dimension D_i; and (4) A side link to the next node at the same level carrying the same label.

The total size of such a tree node is $O(\omega + l_{k+1} + \omega \cdot \sum_{i=k+2}^{n} l_i + 1) = O(\omega \cdot \sum_{i=k+1}^{n} l_i)$. Comparing to the number of tuples in a base table, which can be easily in millions, the number of

dimensions and the cardinality in each dimension are often pretty small. All nodes at the same level of the tree have the same size. A *PAT* can be easily stored and managed in main memory or on disk.

We assume that an order of dimensions is used to construct a *PAT*. In fact, the order of dimensions affects the size of the resulting *PAT*. Heuristically, if we order the dimensions in cardinality ascending order, then the tuples may have good chances to share prefixes and thus the resulting *PAT* may have a small number of nodes. The tradeoff is that the tree nodes may be large due to the large number of infix links. On the other hand, if we sort dimensions in the cardinality descending order, then the number of nodes may be large but the nodes themselves may be small. Theoretically, finding the best order to achieve a minimal *PAT* is an NP-hard problem. This problem is similar to the problem of computing a minimal FP-tree by ordering the frequent items properly. In Section 3.5, we will study the effect of ordering on the size of *PAT* by experiments.

3.3.2 Comparison: *PAT* vs. Previous Methods

Prefix tree (trie) structures have been extensively used in the previous studies on data mining and data warehousing. *PAT* is another prefix tree structure. At first glance, *PAT* may look similar to some of the previous structures, including Cube forest (Johnson & Shasha, 1997), FP-tree (Han et al., 2004), H-tree (Han et al., 2001), Dwarf structure (Sismanis et al., 2002) and QC-tree (Lakshmanan et al., 2003).

An FP-tree (Han et al., 2004), a data structure for frequent itemset mining, records transactions in a prefix tree structure such that transactions sharing common prefixes also collapses to the same prefix in the tree. There are three critical differences between an FP-tree and a *PAT*. First, FP-tree is for transaction data and *PAT* is for relation data. While transactions may have different lengths, all tuples stored in a *PAT* have the same length. Infix links do not appear in an FP-tree. Second, FP-tree does not bear the locality property. Instead, the side links in an FP-tree are built as transactions arrive. As shown later, the locality property in a PAT facilitates the aggregate query answering substantially. Last, an FP-tree is for frequent itemset mining.

During the mining, an FP-tree is scanned multiple times, and the mining results are output. A *PAT* is for aggregate query answering. It is built and maintained by one scan of the data stream. A query answering algorithm searches the *PAT* to answer aggregate queries.

Cube forest (Johnson & Shasha, 1997), H-tree (Han et al., 2001), Dwarf structure (Sismanis et al., 2002) and QC-tree (Lakshmanan et al., 2003) are for data warehousing. PAT distinguishes itself from those designs in the following two aspects.

First, *PAT stores only prefix and infix aggregates, while most of the previous structures, except for* H-tree, *potentially store all aggregates*. In sequel, the number of nodes in *PAT* is linear to the number of tuples in the sliding window and the dimensionality, while those structures are exponential to the dimensionality. Moreover, the size of tree nodes in *PAT* is regular, as analyzed before. The advantages on size and regularity of tree nodes make *PAT* feasible for streaming data.

An *H-tree* is a prefix tree of the tuples in base table. Thus, it is also linear to the dimensionality. However, *H-tree* does not have infix link and thus the query answering on *H-tree* directly can be very costly. Furthermore, *H-tree* is designed as an internal data structure for computing iceberg cells, and is not for data streams.

Second, *PAT is indexed by infix links and side-links, and the side-links have the locality property.* As will be shown soon, the locality property and the infix links facilitate query answering substantially. In most of the previous structures, the search is based on values dimension by dimension.

In terms of size, a *PAT* is larger than an *H-tree*: the difference is infix aggregates and infix links. The size of the infix aggregates and the infix links is quantified in 3.3. The infix links have to meet the locality requirement. Theorem 3.4 will discuss the procedure. As will be shown, it takes the extra cost in time linear in the dimensionality to maintain the locality.

3.3.3 *PAT* Construction

We consider constructing a *PAT* by reading tuples into main memory one by one (or batch by batch), and each tuple can be read into main memory only once. The algorithm is presented in

Figure 3.5 and elaborated in this subsection.

Example 3.2. Let us construct a *PAT* by reading the tuples in Figure 3.2 one by one.

A *PAT* is initialized as a tree with only one node, the root. Then, The first tuple, $(1, a_1, b_2, c_1, d_1, 6)$, is read and inserted into the tree. For each node in the path, a row $(1, 6)$ is registered in the aggregate table. The infix links from *root* to a_1b_2, $a_1b_2c_1$ and $a_1b_2c_1d_1$, infix links from a_1 to $a_1b_2c_1$ and $a_1b_2c_1d_1$, and infix links from a_1b_2 to $a_1b_2c_1d_1$ are created. The corresponding infix aggregate cells are $(*, b_2, *, *)$, $(*, *, c_1, *)$, $(*, *, *, d_1)$, $(a_1, *, c_1, *)$, $(a_1, *, *, d_1)$ and $(a_1, b_2, *, d_1)$, respectively. Once the information is recorded in the tree, we do not need the tuple any more.

Then, we read the second tuple $(1, a_1, b_1, c_1, d_1, 2)$ and insert it into the tree. The aggregate values at nodes *root* and a_1 should be both updated to $(1, 8)$, since they are on the path of the inserted tuple. The infix links from *root* to a_1b_1 and from a_1b_1 to $a_1b_1c_1d_1$ are created and associated with the infix aggregate cells $(*, b_1, *, *)$ and $(a_1, b_1, *, d_1)$, respectively.

The remaining tuples can be inserted similarly. It can be verified that the resulting *PAT* is exactly the one shown in Figure 3.4. □

The construction of a *PAT* by scanning tuples one by one has two major components: building the prefix tree, which is straightforward, and creating/maintaining the correct infix links and side-links, which should follow the procedure justified in the following theorem so that the locality property is respected.

Theorem 3.4. *Let T be a PAT that satisfies the locality property. When a new node v of label d_i is created, the following procedure adjusts the side-links and infix-links so that the resulting PAT preserves the locality property: If v is a child of the root node r, no infix link and side-link are needed; else, the side-links and infix links with respect to v should be adjusted as follows:*

1. *The closest ancestor v' of v should be allocated such that v' has an infix link of d_i;*

2. *If v' does not exist, then an infix link should be built from every ancestor of v in the tree to v, except for the parent node of v.*

3. *Otherwise, let u be the node pointed by the d_i-infix link of v'. Let V be the set of ancestor nodes of v' whose d_i-infix links also point to u.*

   (a) *If u is not the first node of global side link with d_i(in other words, the root node r does not exist in V), v should be inserted into the front of the sublist pointed by the d_i-infix link of v' and the d_i-infix link of v' should point to v.*

   (b) *If u is the first node of side link with d_i globally in the tree(in other words, the root node r exists in V), v should be inserted into the front of the sublist pointed by the d_i-infix link of v' and the d_i-infix link of v' and nodes in V should point to v.*

Proof. The correctness of Step 2 is clear since in such a case, v is the first node carrying label d_i. The corresponding infix links should be created.

Suppose *PAT* T already has some nodes carrying label d_i. To preserve the locality property, v should be inserted to the head of the non-empty consecutive sublist of its closest ancestor. If v is the first node of side links with d_i, the infix links of ancestor nodes of v' should point to v. Once the locality property holds for the smallest subtree containing the new node, the locality property holds for any larger subtrees containing the smallest subtree, since the *PAT* before the insertion has the locality property. □

The complexity of the procedure described in Theorem 3.4 is linear in the dimensionality. At each node, a table of aggregates is maintained. Each table has two columns: time-stamp and aggregate, and at most ω rows. The aggregate at instant t should be stored at the $(t \bmod \omega)$-th row. Therefore, the cost of maintaining and searching the table is constant.

3.3.4 Incremental Maintenance

Suppose that we have a *PAT* at instant t. At instant $(t+1)$, the new data tuples should be read in and inserted into the tree, and the data at instant $(t-\omega+1)$ should be removed, so that the sliding window is moved forward to $[t-\omega+2, t+1]$. The algorithm is shown in Figure 3.6. We explain the critical details as follows.

Algorithm Tree construction
Input: the current sliding window B
Output: a PAT
Method:

```
 1: initialize a tree with only the root node;
 2: read the tuples into main memory one by one, one at each time;
 3: for each tuple do
 4:    insert the tuple into the tree;
 5:    for each node on the path of the inserted tuple do
 6:       update the aggregate at the node;
 7:       if it is a new node then
 8:          adjust infix links and side-links according to Theorem 3.4
 9:       end if
10:    end for
11: end for
```

Figure 3.5: The *PAT* construction algorithm by scanning tuples one by one.

To be efficient, should insertions of tuples at instant $(t+1)$ *go first or deletions of tuples at instant* $(t-\omega+1)$ *first?* Consider a tree node v whose aggregate table contains only an aggregate at instant $(t-\omega+1)$. Suppose that some tuples from the stream at instant $(t+1)$ will contribute a new row in v's aggregate table. If deletions go first, the node would be removed since its aggregate table is empty after the deletion. Then, the insertion of the tuples at instant $(t+1)$ will have to recreate the node. To avoid such an unnecessary deletion-and-re-creation, we should let the insertions of tuples at instant $(t+1)$ go first before the deletions of tuples at instant $(t-\omega+1)$.

Insertion of tuples at instant $(t+1)$ can be done in the way similar to Algorithm in Figure 3.5, the tree construction by scanning the tuples one by one. That is, we take the existing *PAT* at instant t, and insert the tuples at instant $(t+1)$ into the tree.

Please note that, during the insertion, we do not need to scan any tuples in the previous instants. The only tuples scanned are those at instant $(t+1)$.

Now, let us consider *how to delete the tuples whose time-stamps are* $(t-\omega+1)$. A naïve method is as follows. We search the *PAT*. For each node that contains an aggregate at instant $(t-\omega+1)$ in its aggregate table, the corresponding row in the aggregate table should be removed. If the aggregate table becomes empty, then the node should be deleted.

The above naïve method is costly. There can be many nodes in the tree containing aggregates

Algorithm Incremental maintenance
Input: the *PAT* T at instant t,and the tuples at instant $(t+1)$
Output: the *PAT* at instant $(t+1)$
Method:

```
1: insert tuples at instant (t+1) into T,
2: for each node v on the path do
3:    if v's aggregate table contains a row of instant (t−ω+1) then
4:       remove the row from the aggregate table;
5:       if v is a leaf node then
6:          put v into the LUT list of LUT = t+1;
7:       end if
8:    end if
9: end for
10: for each node v in the LUT list of LUT = t−ω+1 do
11:    search v's ancestors upward until a node having aggregates after instant t−ω+1 is encountered;
12:    remove v and those ancestors of v, and the infix links, adjust the related side-links,too
13: end for
```

Figure 3.6: The *PAT* incremental maintenance algorithm.

T	A	B	C	D	M
3	a_1	b_2	c_2	d_2	5
3	a_2	b_2	c_1	d_2	1

Figure 3.7: The tuples at instant 3.

at instant $(t-\omega+1)$. Aggressively updating a large number of nodes may degrade the online performance. Moreover, how to locate the nodes containing aggregates at instant $(t-\omega+1)$ is another problem. Browsing the whole tree can be very expensive.

Here, we propose a *lazy approach*: the nodes whose aggregate table has only rows of instants $(t-\omega+1)$ or earlier have to be removed at instant $(t+1)$, in order to release the space. Other than that, the deletions of the old aggregates of instant $(t-\omega+1)$ from the nodes are deferred and conducted as a byproduct of future insertions. The idea is elaborated in the following example.

Example 3.3. Suppose the tuples at instant 3 are as shown in Figure 3.7. Since the size of the sliding window $\omega = 2$, the tuples at instant 3 should be inserted and the tuples of instant 1 should be removed. Let us consider how the *PAT* in Figure 3.4 should be incrementally maintained.

We first insert the tuples at instant 3 into the *PAT*. Tuple $(3, a_1, b_2, c_2, d_2, 5)$ is inserted from the root node as a path "a_1-b_2-c_2-d_2". A record $(3,5)$ will be stored at the first row of the ag-

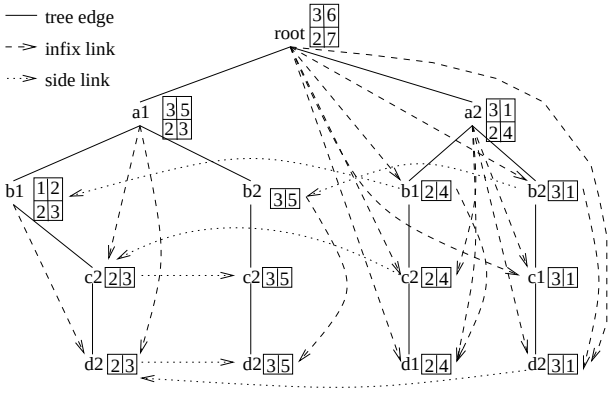


Figure 3.8: Prefix aggregate tree at instant 3.

gregate table, since 3 mod 2 = 1. It overwrites record $(1,8)$ automatically. Similarly, we update the aggregate tables at nodes a_1 and a_1b_2, and the aggregate tables for the related infix links, respectively. Please note that the removal of data at instant 1 from these nodes are conducted as a *byproduct* of the insertions, i.e., we do not actively search for the nodes whose aggregate tables having rows of instant 1. To complete the insertion, two new nodes, $a_1b_2c_2$ and $a_1b_2c_2d_2$, are created. Following Theorem 3.4, the aggregate tables at these nodes as well as the appropriate side-links and infix links are adjusted. The second tuple, $(3,a_2,b_2,c_1,d_2,1)$ can be inserted similarly.

Then, we should remove all those nodes whose aggregate tables contain only rows of instant 1. To find such nodes, we maintain an integer for each leaf node, called the *last update time-stamp* (LUT), which is the latest time-stamp that the node is updated. All leaf nodes having the same LUT are linked together as a linked list. By browsing the linked list for LUT$=1$, we remove the leaf nodes and their ancestors that have only aggregates at instant 1. The upward search stops when the first ancestor having aggregates at instant other than 1 is encountered. In this example, nodes $a_1b_1c_1$, $a_1b_1c_1d_1$, $a_1b_2c_1$, and $a_1b_2c_1d_1$ are removed. The resulting tree is shown in Figure 3.8.

Please note that the aggregate table at node a_1b_1 still contains the aggregate at instant 1. However, this information does not affect our query answering. This row will be removed in the future. At instant 4, if there is a new tuple having a_1b_1 as a prefix, the row will be overwritten and the node will be updated. Otherwise, the node will be removed when we clean up nodes

containing only aggregates at instant 2 or earlier. □

In summary, at instant $(t+1)$, the incremental maintenance algorithm only scans the tuples having time-stamp $(t+1)$ once, and inserts them into the existing *PAT*. By following the LUT list of $(t-\omega+1)$, the maintenance algorithm removes those tree nodes and corresponding infix links whose aggregate tables have only rows of instant $(t-\omega+1)$ or earlier. It never browses the complete *PAT* during the incremental maintenance.

Theorem 3.5. *The time complexity of constructing and incrementally maintaining a* PAT *is* $O(nl)$*, where n is the number of tuples needed to be inserted into the* PAT*, and l is the cardinality.*

Proof. The complexity follows the algorithms in Figures 3.5 and 3.6. For each tuple, the insertion time and the time to maintain the locality of the infix links are linear in the dimensionality l. □

Moreover, if the incremental tuples can be held into main memory, then, they can be sorted and inserted in batch.

3.4 Aggregate Query Answering

We consider how to answer queries of two categories: point queries and range queries.

3.4.1 Answering Point Queries

Point queries can be answered efficiently using a *PAT*. The algorithm is shown in Figure 3.9. We illustrate the major idea in the following example.

Example 3.4 (Point query answering). Let us use the *PAT* shown in Figure 3.4 to answer some illustrative point queries about the sliding window $[1,2]$ in the data stream of our running example.

Algorithm Answering point queries
Input: the *PAT T* at instant t, and a query cell $q(\tau, d_1, \ldots, d_n)$
Output: the aggregate of the query cell
Method:

1: **if** $\tau \neq *$ and the root's aggregate table does not have a row about τ **then**
2: RETURN $(null)$;
3: **end if**
4: let *current-node*=*root*;
5: $m = search(current\text{-}node, q)$;
6: RETURN (m);

Function *search*(*current-node, q-cell*)

1: suppose $q\text{-}cell = (\tau, d_1, \ldots, d_l)$;
2: **if** $d_1 \neq *$ **then**
3: **if** *current-node* has a child v of label d_1 **then**
4: **if** $\tau \neq *$ and there is no row of instant τ in the aggregate table of v **then**
5: RETURN $(null)$;
6: **else**
7: $current\text{-}node = v$; $q' = (\tau, d_2, \ldots, d_l)$; $m = search(current\text{-}node, q')$;
8: RETURN (m);
9: **end if**
10: **else**
11: RETURN (m);
12: **end if**
13: **else**
14: **if** $d_1 = \cdots = d_l = *$ **then**
15: RETURN the aggregate at *current-node*;
16: **end if**
17: let d_i be the first dimension non-$*$ value in q;
18: **if** *current-node* has no infix link of label d_i **then**
19: RETURN (m);
20: **end if**
21: **if** $\tau \neq *$ and the aggregate table of the infix link of d_i has no row of τ **then**
22: RETURN (m);
23: **end if**
24: $m = 0$;
25: follow the side links, visit every node carrying label d_i in the subtree of v,
26: **for** each node v' in the linked list **do**
27: let $current\text{-}node = v'$; $q' = (d_{i+1}, \ldots, d_l)$; $m = aggr(m, search(current\text{-}node, q'))$;
28: **end for**
29: RETURN (m)
30: **end if**

Figure 3.9: The algorithm answering point queries.

First, consider query cell $q_1 = (1, a_1, b_1, *, *)$, which itself is a prefix aggregate cell. Its aggregate, 2, is registered in the aggregate table of node a_1b_1. Following the path from the root to the node, we retrieve the answer immediately.

Let us consider query cell $q_2 = (*, *, b_1, *, *)$. It is not a prefix aggregate cell. Instead, it is an infix aggregate cell. Thus, by the aggregate table associated with the infix link of label b_1 at node *root*, we can retrieve the aggregate. Please note that the aggregates are stored by instants in the aggregate table, i.e., two rows $(1, 2)$ and $(2, 7)$ are in the aggregate table of the infix link. We need to get the sum of them since $\tau = *$ in this query cell.

As the third example, let $q_3 = (*, *, b_1, *, d_1)$. This query cannot be answered by one node in the tree. Instead, following the infix link of label b_1 at node *root*, we can reach the linked list of all nodes having b_1. Following the side-links, we can access all the nodes of b_1 in the tree.

For each node in the linked list, we recursively retrieve the aggregates from the infix link of label d_1 at the node. For example, at node a_1b_1, by the infix link of d_1, we immediately know the aggregate of $(*, a_1, b_1, *, d_1)$ is 1 even without visiting any node of d_1. Similarly, at node a_2b_1, we can retrieve the aggregate of $(*, a_2, b_1, *, d_1)$, 4, from the infix link. The sum of the aggregates from the infix aggregate cells, 6, answers the query.

As another example, let us consider query cell $q_4 = (*, a_2, *, c_2, d_2)$. Following the tree edge *root*-a_2 and infix link of c_2 at node a_2, we reach the local linked list of c_2 in the subtree of a_2. The locality property of the side-links and the infix link avoids the search of the complete linked list of c_2. Since $a_2b_1c_2$ has only one child, which is $a_2b_1c_2d_1$, the query returns *null*. Here, *null* means there is not any tuple matching the aggregate cell.

The aggregate tables can also be used to prune the search. For example, consider query cell $q_5 = (2, a_1, b_2, *, d_1)$. It follows the path a_1b_2 in the *PAT*. However, the aggregate table at node a_1b_2 does not contain any row of instant 2. Thus, we can return *null* immediately without searching the subtree any more. □

As can be seen, in answering point queries, infix links and side-links are used to search only the related tuples. Moreover, the locality property of side-links guarantees that we do not need to search any unnecessary nodes or branches.

3.4.2 Answering Range Queries

In principle, a range query can be rewritten as a set of point queries. Then, Algorithm of answering point queries in Figure 3.9 can be called repeatedly to answer the queries. However, calling the algorithm of answering point queries and thus searching the PAT repeatedly may not be efficient.

Here, we propose a *duel pruning* approach, as exemplified in the following example.

Example 3.5 (Range query answering). Let us consider how to answer range query $(2, *, \{b_0, b_1\}, *, \{d_1, d_2\})$. The query cell can be rewritten as a set of 4 queries cells: $(2, *, b_0, *, d_1)$, $(2, *, b_0, *, d_2)$, $(2, *, b_1, *, d_1)$ and $(2, *, b_1, *, d_2)$. Algorithm in Figure 3.9 can be called four times to answer the point queries, respectively, and the sum, 7, should be returned.

Instead of calling algorithm of answering point queries four times, we can search the *PAT* as follows.

We start from the root node, since the first non-$*$ dimension value in the query cell should be either b_0 or b_1 on dimension *B*, we search the infix links of b_0 and b_1 from the root node. Since there exists no infix link of b_0, we prune the range query cell to $(2, *, b_1, *, \{d_1, d_2\})$. At the same time, we only need to search subtrees rooted at the nodes having label b_1, which are linked by the side-links. That is, a part of the search space is also pruned.

There are two nodes carrying label b_1 in the *PAT*, a_1b_1 and a_2b_1. We search them one by one. For node a_1b_1, since the next non-$*$ dimension value in the query cell should be either d_1 or d_2, and the time-stamp is 2, only the infix links of d_1 and d_2 are searched, and only the infix link of d_2 has a row of time-stamp 2. Thus, the aggregate 3 is extracted. Similarly, aggregate value 4 is extracted from the subtree of a_2b_1. Thus, the sum 7 is returned. □

As shown in Example 3.5, the major idea of progressive pruning method for answering range queries is that, instead of searching a *PAT* many times, we conduct the search using the range query from the root of a *PAT*. At each node under the search, the query range is narrowed using the information of the available infix links and the corresponding aggregate tables, and the unnecessary nodes are pruned from the search space using the range specification in the

query. By progressive pruning, we search the *PAT* only once for any range query.

3.5 Experimental Results

In this section, we report the experimental results from a systematic performance study. All the algorithms are implemented in C++ on a laptop PC with a 2.8 GHz Pentium 4 processor, 60 G hard drive, and 512 MB main memory, running Microsoft Windows XP. In all of our experiments, the PATs reside in main memory.

We generate the synthetic data sets following the Zipf distribution. To generate the data sets, our data generator takes the parameters of the Zipf factor, the dimensionality, the number of tuples, and the cardinality in each dimension. To generate a tuple, we generate the data for each dimension independently following the Zipf distribution.

Such a data generation method is popularly used in many previous studies (Beyer & Ramakrishnan, 1999; Ross & Srivastava, 1997; W. Wang et al., 2002; Sismanis et al., 2002) on data cube and data warehouse computation. To some extent, it is a benchmark approach to generating synthetic data sets for data cube computation.

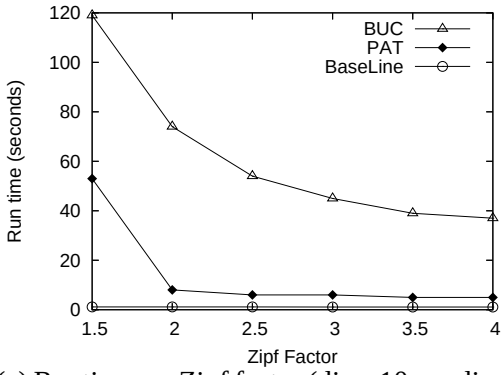
In our test, we also used the weather data set (Hahn, 1994) which is a real data set. The weather data set is well accepted a benchmark data set for data cube computation (Beyer & Ramakrishnan, 1999; Ross & Srivastava, 1997; W. Wang et al., 2002; Sismanis et al., 2002).

We tested three methods: the *PAT* method developed in this dissertation, the *BUC* method (Beyer & Ramakrishnan, 1999) and a baseline method. The baseline method just simply stores and sorts all tuples in the current sliding window. As expected, the baseline method uses the least main memory to store the data and costs the least runtime to maintain the current sliding window. The tradeoff is the slowest query answering performance. To answer any query, the baseline method has to scan the tuples in the current sliding window. A binary search can be used to locate the tuples matching the time interval of the query. On the other hand, the *BUC* method computes the whole data cube. It costs the most in computing the whole cube and storing the aggregates. We measure both the main memory cost of *BUC* and the size of the data

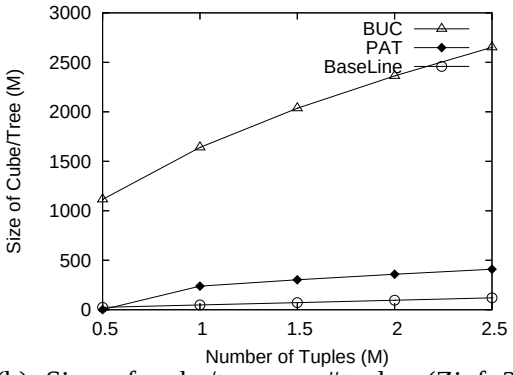
cube computed by *BUC*, which is stored on disk. To answer a query, *BUC* only needs to conduct a binary search to allocate the corresponding aggregate tuple. Thus, the query answering time is fast. *PAT* is in between: to compute and store the aggregates for the current sliding window, it costs more space and runtime than the baseline method but less than the *BUC* method; on the other hand, in query answering, it searches less than the baseline method accordingly.

3.5.1 Building Prefix Aggregate Trees

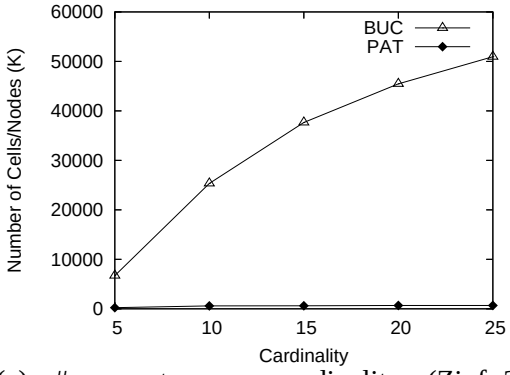
We tested the size of the data cube computed by *BUC*, the size of *PAT*, the number of aggregates computed, the memory usage (the highest watermark of memory usage during the running of the programs) and scalability (runtime) of the three methods. Four factors are considered: the Zipf factor, the dimensionality, the cardinality of the dimensions and the number of tuples in the current sliding window. The results are consistent. Some results are shown in Figure 3.10.



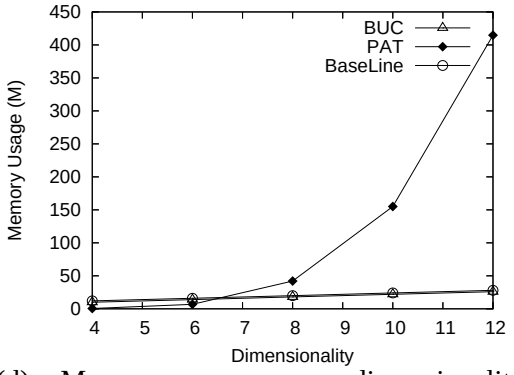
(a) Runtime vs. Zipf factor (dim=10, cardinality=10, #tuples=500K)



(b) Size of cube/tree vs. #tuples (Zipf=2, dim=10, cardinality=10)



(c) #aggregates vs. cardinality (Zipf=2, dim=10, #tuples=500K)



(d) Memory usage vs. dimensionality (Zipf=2, cardinality=10, #tuples=500K)

Figure 3.10: Results on constructing *PAT*.

As shown Figure 3.10(a), the baseline method is not sensitive to the Zipf factor at all, since it simply maintains the tuples in the current sliding window and does not pre-compute any aggregate. Both *BUC* and *PAT* are sensitive to the Zipf factor: the smaller the Zipf factor, the more distinct aggregates exist in the data set. However, *PAT* runs faster than *BUC* since it computes much less aggregate cells than *BUC*.

Figure 3.10(b) measures the size of the data cube of *BUC*, the size of the *PAT* and the size of the current sliding window in Baseline. The size of *PAT* counts both the prefix aggregates, infix aggregates and links. The number of tuples varies from 500 thousand to 2.5 million so that the scalability of the methods are tested. All three methods are roughly linear on the number of tuples, but *PAT* and the baseline method generate much smaller results than *BUC*. In other words, Baseline does not generate any aggregates but only the base tuples are maintained. PAT generates the prefix aggregates and the infix aggregates, which form a substantially small subset of all the aggregates generated by *BUC*. Even when the whole data cube is over 2.5 GB, the *PAT* including the links occupies less than 500 MB, which is less than 3 times of the size of all base tuples and can be easily accommodated in main memory.

Figure 3.10(c) shows that the number of aggregate cells (including prefix aggregates and infix aggregates) in *PAT* is linear in the cardinality of the dimensions. The trend is mild. When the cardinality increases, the data set becomes sparse and thus the total number of aggregates also increases. *BUC* computes all aggregates. As shown in the same figure, the increase of all aggregates is sub-linear in our experiments, but the number of all aggregates is much larger than the number of prefix aggregates and infix aggregates computed by *PAT*.

Figure 3.10(d) shows the memory usage of the three methods. Please note that *BUC* stores the aggregates on disk. It only maintains the base table in main memory. Thus, it uses the same amount of main memory as the baseline method. Since the *PAT* resides in main memory, the memory usage of constructing a *PAT* increases as the dimensionality increases. When there are many dimensions, *PAT* has many levels.

In summary, a *PAT* is usually much smaller than the size of a data cube. Constructing a *PAT* is also much faster than computing a data cube using *BUC*.

3.5.2 Incremental Maintenance

When testing the performance of incremental maintenance, we always set the number of tuples at each instant to a constant. A sliding window of 10 instants was used. We set the number of tuples in the original *PAT* to 500 K. We only compare the *PAT* method and the baseline method. For *BUC*, there is not incremental maintenance algorithm. Thus, to incrementally maintain all the aggregates, we have to compute the data cube on the new data and merge the new aggregates with the existing ones. It has a similar performance as shown in Section 3.5.1. Some results are shown in Figure 3.11.

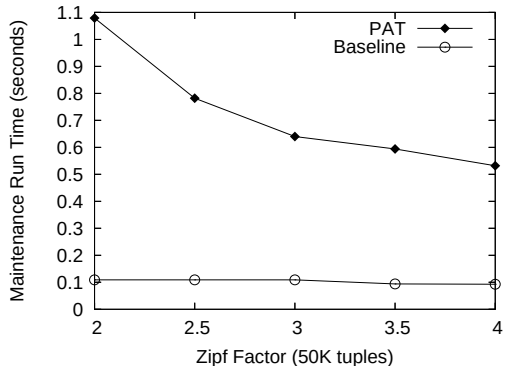
Figure 3.11(a) shows the maintenance runtime versus the Zipf factor. It is consistent with Figure 3.10(a). With a lower Zipf factor, the data set is sparser and thus *PAT* computes more prefix aggregates and infix aggregates. The baseline method is constant since it does not compute any aggregates. However, by comparing Figures 3.11(a) and 3.10(a), we can see that the incremental maintenance time is much shorter, since many paths in the existing *PAT* can be reused in the incremental maintenance.

As shown in Figure 3.11(b), the incremental maintenance time of *PAT* increases as the dimensionality increases, since the tree becomes larger and taller on high dimensional data sets. The maintenance time of Baseline also increases, but it is linear.

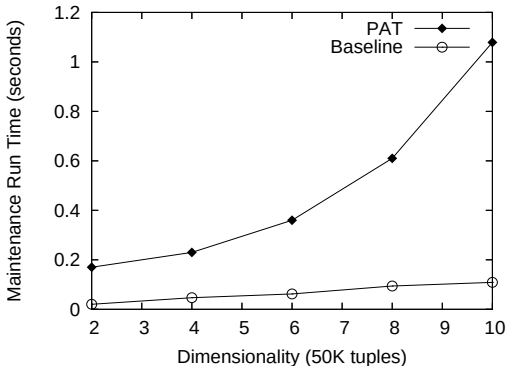
In Figure 3.11(c), we tested the scalability of the incremental maintenance of *PAT* and Baseline with respect to the number of new tuples at each instant. The result shows that both *PAT* and Baseline have an approximately linear scalability. This is consistent with the analysis of the *PAT* incremental maintenance algorithm.

Figure 3.11(d) examines the size of the *PAT* with respect to the number of new tuples at each instant. Interestingly, we observed that, under a given data distribution, the size of the *PAT* is stable and insensitive to the number of tuples in the incremental part. In other words, the size of *PAT* mainly depends on the number of tuples in the sliding window, and is stable during the incremental maintenance. This is a nice property for data stream processing: no matter how large the data stream is, we will have an index structure of a stable size.

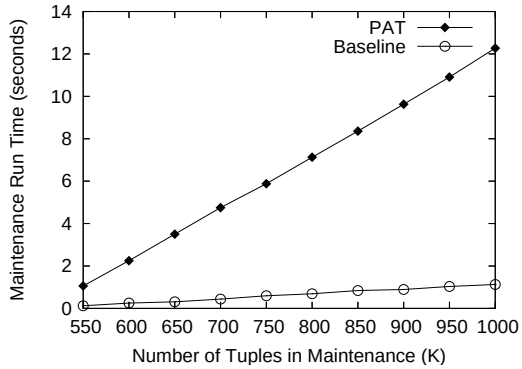
Figure 3.11(e) shows the memory usage with respect to the Zipf factor. Again, when the



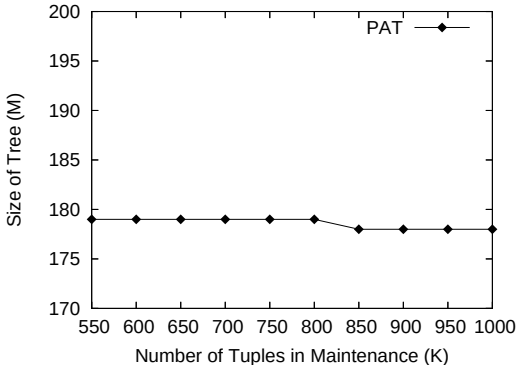
(a) Runtime vs. Zipf factor (dim=10, cardinality=10, #new tuples=50K)



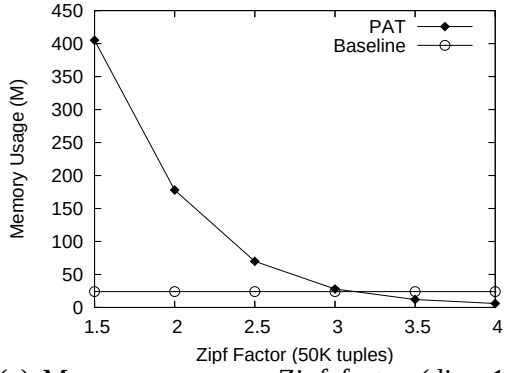
(b) Runtime vs. dimensionality (Zipf=2, cardinality=10, #new tuples=50K)



(c) Runtime vs. #new tuples (Zipf=2, dim=10, cardinality=10)



(d) Size of tree vs. #new tuples (Zipf=2, dim=10, cardinality=10)



(e) Memory usage vs. Zipf factor (dim=10, cardinality=10,#new tuples=50K)

Figure 3.11: Results on incremental maintenance of *PAT*.

Zipf factor is small, the data is sparse and thus not many prefixes can be shared. When the Zipf factor becomes larger, the data becomes more skewed, and the *PAT* becomes smaller due to the more sharing of the prefixes. The baseline uses constant memory in the maintenance, since it only needs to load the current sliding window into main memory.

In summary, incremental maintenance of a *PAT* is scalable in both runtime and space usage with respect to the size of sliding window.

3.5.3 The Order of Dimensions

We also tested the effect of different orders of dimensions on the size of the *PATs* and the runtime of *PAT* construction. We made up a synthetic data set of 10 dimensions, Zipf factor 3 and 1 million tuples. The i-th dimension $(1 \leq i \leq 10)$ has a cardinality of i. We tested the effects of the following 4 orders of dimensions:

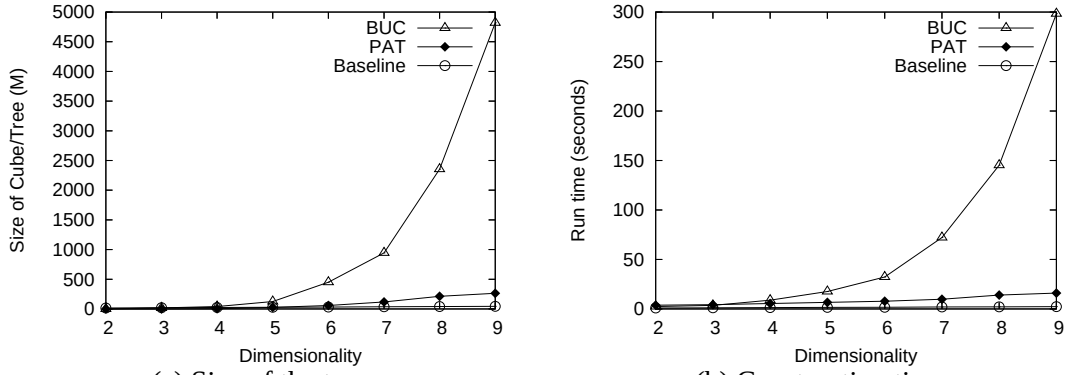
- R_1: cardinality ascending order;
- R_2: cardinality descending order;
- R_3: D_5-D_6-D_4-D_7-D_3-D_8-D_2-D_9-D_1-D_{10}; and
- R_4: D_1-D_{10}-D_2-D_9-D_3-D_8-D_4-D_7-D_5-D_6.

Order	Runtime	#nodes	Tree size (bytes)
R_1	16.37	6,433	1,521,640
R_2	16.67	16,441	2,548,404
R_3	17.14	8,694	2,180,736
R_4	16.74	8,575	1,812,868

Figure 3.12: The effect of orders of dimensions.

The results are shown in Figure 3.12. The *PAT* construction time is insensitive to the order of dimensions, since the number of tree node accesses is basically the same no matter which order is used.

Both the number of nodes in the *PAT* and the size of the tree are sensitive to the orders. With order R_1, putting dimensions of low cardinality ahead strongly facilitates the prefix sharing,



(a) Size of the tree. (b) Construction time.

Figure 3.13: Results on real data set Weather.

and leads to the smallest number of nodes. As discussed at the end of Section 3.3.1, at level i, the size of the tree nodes is proportional to the sum of cardinalities in dimensions D_{i+1} to D_n. Therefore, the average size of tree nodes using order R_1 is the largest. However, the advantage of reduction on number of nodes well overcomes the disadvantage of large tree node size. Thus, order R_1 achieves the smallest tree. Order R_2 suffers from deficiency in sharing the prefixes. Although its average tree node size is the smallest, the tree size turns out to be the largest. Orders R_3 and R_4 stay in between.

Based on this experiment, we recommend ordering the dimensions in cardinality ascending order to explore possible sharing of prefixes. However, there is no theoretical guarantee that the cardinality ascending order always leads to the smallest tree.

3.5.4 Results on the Weather Data Set

We also tested the *PAT* construction using the well-accepted weather data set (Hahn, 1994), which contains $1,015,367$ tuples and 9 dimensions. The dimensions with the cardinalities of each dimension are as follows: station-id $(7,037)$, longitude (352), solar-altitude (179), latitude (152), present-weather (101), day (30), weather-change-code (10), hour (8), and brightness (2). 8 data sets with 2 to 9 dimensions are generated by projecting the weather data set on the first k dimensions $(1 \leq k \leq 9)$. Figure 3.13 shows the results.

Interestingly, the size of the *PAT* on the complete data set is only 263 MB, comparable to

the size of QC-tree (241.2 Mb reported in (Lakshmanan et al., 2003)), a recently developed data cube compression method. However, to construct a QC-tree, the base table has to be scanned and sorted multiple times, and incremental maintenance of a QC-tree is more costly than *PAT*. The *PAT* construction is also much faster than computing the whole cube using *BUC* but slower than Baseline(Figure 3.13(b)). As indicated in (Lakshmanann et al., 2002), construction of a quotient cube is slower than *BUC*, since extra work is needed to achieve compression.

In summary, the experimental results on the real data set are consistent with the observations that we obtained from the experiments on the synthetic data sets.

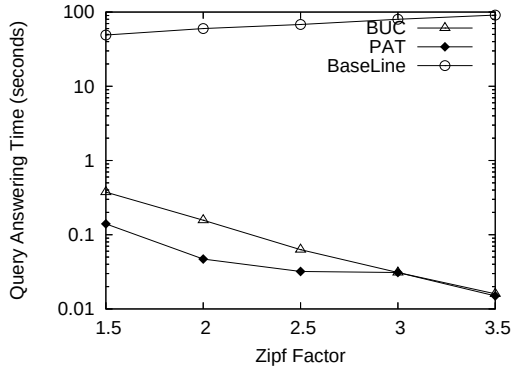
3.5.5 Query Answering

We reported the performance of answering point queries. The results on range queries are similar. In each test, we randomly select $1,000$ aggregate cells from the data cube and use them as the point queries. In other words, all queries are on non-empty aggregates.

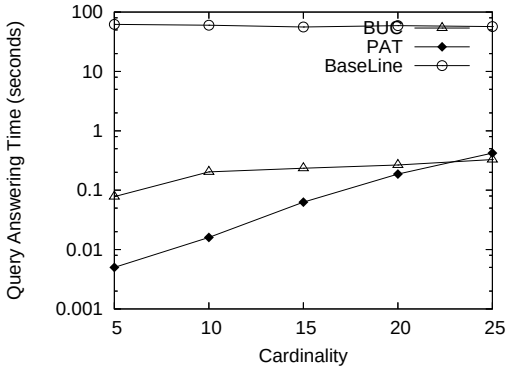
Beyer & Ramakrishnan (1999) do not give a query answering algorithm for *BUC*. In this performance study, we store the whole data cube *in main memory* as a table, and all tuples are sorted according to the dictionary order. Therefore, answering any point query using the whole cube can be done by a binary search. This is probably the best case of query answering using a data cube. In real applications, usually a whole data cube cannot be held into main memory. To this extent, our experiments favor the query answering using the whole cube. In the Figure 3.14, we measure the query answering time taken by $1,000$ queries. All curves are plotted in logarithmic scale.

From the figures, we can clearly see that both *PAT* and *BUC* have a much better query answering performance than the baseline method. The baseline method is two orders of magnitudes slower. That simply indicates that, in order to answer aggregate queries online, materializing some aggregate cells in very effective.

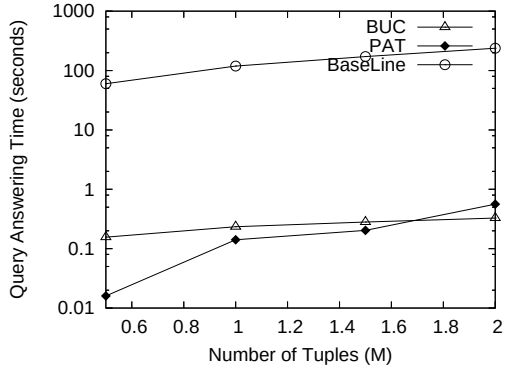
PAT and *BUC* have comparable performance in terms of query answering. However, remember that *PAT* computes and materializes much fewer aggregate cells than *BUC*. In sequel, the space over head for storing a *PAT* is much smaller than the space for all the aggregate cells



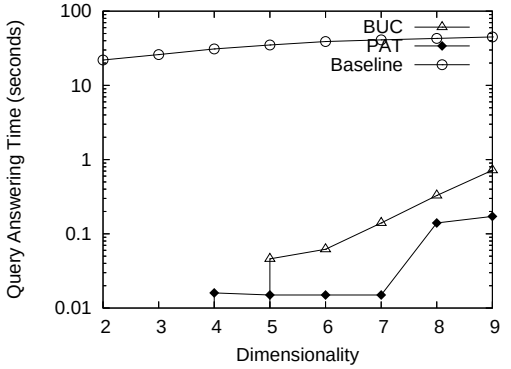
(a) Runtime vs. Zipf factor (dim=10, cardinality=10, #tuples=500K, #queries=1K)



(b) Runtime vs. cardinality (Zipf=2, dim=10, #tuples=500K, #queries=1K)



(c) Runtime vs. #tuples (Zipf=2, dim=10, cardinality=10, #queries=1K)



(d) The Weather data set

Figure 3.14: Results on query answering using PATs.

computed by *BUC*, as shown in the previous experiments. Moreover, *BUC* needs to scan the base table multiple times to compute a complete data cube. Thus, *PAT* is a nice tradeoff between space and query answering time.

3.5.6 Summary

Based on the above experimental results, we have the following observations. First, the size of a *PAT* is substantially smaller than that of a data cube. That makes the *PAT* feasible in space for data streams. Second, our algorithms for constructing and incrementally maintaining a *PAT* are efficient and highly scalable for data streams. The construction and maintenance cost is dramatically smaller than the cost of materializing the whole cube. Third, query answering using a *PAT* is comparable to the best cases using a full cube. It is much faster than the baseline method. The *PAT* approach can be regarded as a good tradeoff between the construction/maintenance

cost and the query answering performance.

Chapter 4

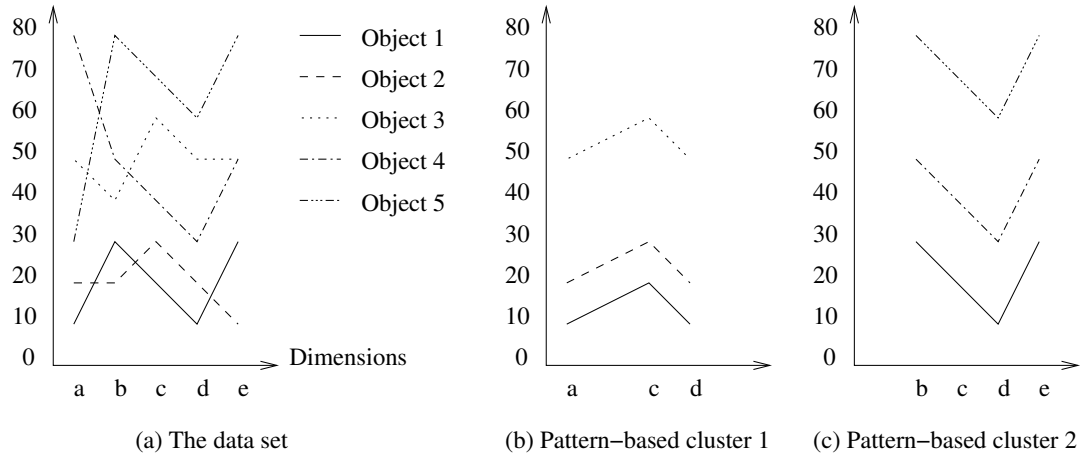
Warehousing pattern-based clusters

4.1 Preliminaries

Clustering large databases is a challenging data mining task with many important applications. Most of the previously proposed methods are based on similarity measures defined globally on a (sub)set of attributes/dimensions. However, in some applications, it is hard or even infeasible to define a good similarity measure on a global subset of attributes to serve the clustering.

To appreciate the problem, let us consider clustering the 5 objects in Figure 4.1(a). There are 5 dimensions, namely *a*, *b*, *c*, *d* and *e*. No patterns among the 5 objects are visibly explicit. However, as elaborated in Figure 4.1(b) and (c), respectively, objects 1, 2 and 3 follow the same pattern in dimensions *a*, *c* and *d*, while objects 1, 4 and 5 share another similar pattern in dimensions *b*, *c*, *d* and *e*. If we use the patterns as features, they form two *pattern-based clusters*.

The flexibility of pattern-based clustering may provide interesting and important insights in some applications where conventional clustering methods may meet difficulties. For example, in DNA micro-array data analysis, the gene expression data are organized as matrices, where rows represent genes and columns represent samples/conditions. The number in each cell records the expression level of the particular gene under the particular condition. The matrices often contain thousands of genes and tens of conditions. It is important to identify subsets of



(a) The data set (b) Pattern−based cluster 1 (c) Pattern−based cluster 2

Figure 4.1: A set of objects as a motivating example.

genes whose expression levels change coherently under a subset of conditions. Such information is critical in revealing the significant connections in gene regulatory networks. As another example, in the applications of automatic recommendation and target marketing, it is essential to identify sets of customers/clients with similar behavior/interest. As a concrete example, suppose that the ranks of movies given by customers are collected. To identify customer groups, it is essential to find the subsets of customers who rank subsets of movies similarly. In the above two examples, pattern-based clustering is the major data mining task.

The pattern-based clustering problem is proposed and a mining algorithm is developed by H. Wang et al. (2002). However, some important problems remain not thoroughly explored. In particular, we address the following two fundamental issues and make corresponding contributions in this chapter.

- *What is the effective representation of pattern-based clusters?* As can be imagined, there can exist many pattern-based clusters in a large database. Given a pattern-based cluster C, any non-empty subset of the objects in the cluster is trivially a pattern-based cluster on any non-empty subset of the dimensions. Mining and analyzing a huge number of pattern-based clusters may become the bottleneck of effective analysis. *Can we devise a succinct representation of the pattern-based clusters?*

  Our contributions. In this chapter, we propose the mining of *maximal pattern-based*

clusters. The idea is to report only those non-redundant pattern-based clusters, and skip their trivial sub-clusters. We show that, by mining maximal pattern-based clusters, the number of clusters can be reduced substantially. Moreover, many unnecessary searches for sub-clusters can be pruned and thus the mining efficiency can be improved dramatically as well.

- *How to mine the maximal pattern-based clusters efficiently?* Our experimental results indicate that the algorithm *p-Clustering* (H. Wang et al., 2002) may not be satisfactorily efficient or scalable in large databases. The major bottleneck is that it has to search many possible combinations of objects and dimensions.

  Our contributions. In this chapter, we develop two novel mining algorithms, *MaPle* and *MaPle+* (*MaPle* for Maximal Pattern-based Clustering). They conduct a depth-first, progressively refining search to mine maximal pattern-based clusters. We propose techniques to guarantee the completeness of the search and also prune unpromising search branches whenever it is possible. *MaPle+* also integrates several interesting heuristics further.

  An extensive performance study on both synthetic data sets and real data sets is reported. The results show that *MaPle* and *MaPle+* are significantly more efficient and more scalable in mining large databases than method *p-Clustering* (H. Wang et al., 2002). In many cases, *MaPle+* is better than *MaPle*.

The remainder of the chapter is organized as follows. In Section 4.2, we define the problem of mining maximal pattern-based clusters, review related work, compare pattern-based clustering and traditional partition-based clustering, and discuss the complexity. Particularly, we exemplify the idea of method *p-Clustering* (H. Wang et al., 2002). In Section 4.3, we develop algorithms *MaPle* and *MaPle+*. An extensive performance study is reported in Section 4.4.

4.2 Problem Definition and Related Work

In this section, we propose the problem of maximal pattern-based clustering and review related work. In particular, *p-Clustering*, a pattern-based clustering method developed by H. Wang et al. (2002), will be examined in detail.

4.2.1 Pattern-Based Clustering

Given a set of objects, where each object is described by a set of attributes. A pattern-based cluster (R,D) is a subset of objects R that exhibit a coherent pattern on a subset of attributes D. To formulate the problem, it is essential to describe, given a subset of objects R and a subset of attributes D, how coherent the objects are on the attributes. The measure *pScore* serves this purpose.

Definition 4.1 (pScore (H. Wang et al., 2002)). *Let $DB = \{r_1, \ldots, r_n\}$ be a database with n* objects. *Each object has m* attributes $A = \{a_1, \ldots, a_m\}$. *We assume that each attribute is in the domain of real numbers. The value on attribute a_i of object r_j is denoted as $r_j.a_i$. For any objects $r_x, r_y \in DB$ and any attributes $a_u, a_v \in A$, the **p**Score is defined as* $pScore\left(\begin{bmatrix} r_x.a_u & r_x.a_v \\ r_y.a_u & r_y.a_v \end{bmatrix}\right) = \|(r_x.a_u - r_y.a_u) - (r_x.a_v - r_y.a_v)\|$.

The meaning of *pScore* is shown in Figure 4.2. *pScore* describes the difference of changes between two objects on two attributes. As illustrated in Figure 4.2, the smaller the *pScore* value, the more similar the two objects on the two dimensions.

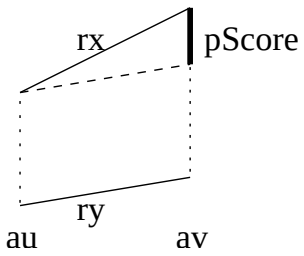


Figure 4.2: The *pScore* of two objects r_x and r_y on attributes a_v and a_u.

Pattern-based clusters can be defined as follows.

Definition 4.2 (Pattern-based cluster (H. Wang et al., 2002)). *Let $R \subseteq DB$ be a subset of objects in the database and $D \subseteq A$ be a subset of attributes. (R,D) is said a* **δ-pCluster** *(***pCluster** *is for pattern-based cluster) if for any objects $r_x, r_y \in R$ and any attributes $a_u, a_v \in D$,*

$$pScore\left(\begin{bmatrix} r_x.a_u & r_x.a_v \\ r_y.a_u & r_y.a_v \end{bmatrix}\right) \leq \delta, \text{ where } \delta \geq 0.$$

Given a database of objects, the pattern-based clustering is to find the pattern-based clusters from the database. In a large database with many attributes, there can be many coincident, statistically insignificant pattern-based clusters, which consist of very few objects or on very few attributes. A cluster may be considered *statistically insignificant* if it contains a small number of objects, or a small number of attributes. Thus, in addition to the quality requirement on the pattern-based clusters using an upper bound on *pScore*, a user may want to impose constraints on the minimum number of objects and the minimum number of attributes in a pattern-based cluster.

In general, given (1) a cluster threshold δ, (2) an *attribute threshold* min_a (i.e., the minimum number of attributes), and (3) an *object threshold* min_o (i.e., the minimum number of objects), the task of *mining delta-pClusters* is to find the complete set of δ-pClusters (R,D) such that $(\|R\| \geq min_o)$ and $(\|D\| \geq min_a)$. A δ-pCluster satisfying the above requirement is called *significant*.

4.2.2 Comparison Between Pattern-Based Clustering and Partition-Based Clustering

It is interesting to note that pattern-based clustering and (traditional) partition-based clustering (e.g., *k*-means) are defining clustering from two different angles.

In traditional partition-based clustering, such as *k*-means, the task is to partition the objects into several subsets such that the similarity between objects in the same subset is as high as possible. This can be regarded as an optimization procedure. The number of clusters is often small. We usually need a similarity measure defined for any two objects. Moreover, a quality function

such as the sum of similarity between objects in the same clusters is used for optimization.

The partition-based clustering problem defined in many forms is NP-Complete. Thus, many approximation algorithms are proposed.

On the other hand, pattern-based clustering specifies quality requirements on clusters, such as the *pScore*, the minimum number of objects and the minimum number of attributes in a cluster. The task is to search for all subsets of objects as clusters and the corresponding subsets of attributes that satisfy the quality requirement. It can be regarded as an enumeration problem.

As will be shown later, the pattern-based clustering problem is also NP-Complete. That is partially due to its inherent difficulty of enumerating all combinations that satisfy the quality requirements. Algorithm $p-Clustering$ and the two algorithms developed in this chapter are not approximations methods. Instead, they return the complete set of answers if they can finish. We summarize the above comparison in Figure 4.3.

	Partition-based Clustering	Pattern-based Clustering
Input	A similarity measure, a quality function	A set of requirements on quality of each cluster
Task	Optimize the quality function	Search for all combinations of objects as clusters and the corresponding subsets of attributes that satisfy the quality requirements
Difficulty	NP-Complete	NP-Complete
Typical solutions	Approximation methods	Search for complete answers with heuristics to speed up the search

Figure 4.3: A comparison between partition-based clustering and pattern-based clustering.

4.2.3 Maximal Pattern-Based Clustering

Although the attribute and object thresholds are used to filter out insignificant pClusters, there still can be some "*redundant*" significant pClusters. For example, consider the objects in Figure 4.1. Let $\delta = 5$, $min_a = 3$ and $min_o = 3$. Then, we have 6 significant pClusters: $C_1 = (\{1,2,3\},\{a,c,d\}), C_2 = (\{1,4,5\},\{b,c,d\}), C_3 = (\{1,4,5\},\{b,c,e\}), C_4 = (\{1,4,5\},\{b,d,e\})$, $C_5 = (\{1,4,5\},\{c,d,e\})$, and $C_6 = (\{1,4,5\},\{b,c,d,e\})$. Among them, C_2, C_3, C_4 and C_5 are

subsumed by C_6, i.e., the objects and attributes in the four clusters, C_2-C_5, are subsets of the ones in C_6.

In general, a pCluster $C_1 = (R_1, D_1)$ is called a *sub-cluster* of $C_2 = (R_2, D_2)$ provided $(R_1 \subseteq R_2) \wedge (D_1 \subseteq D_2) \wedge (\|R_1\| \geq 2) \wedge (\|D_1\| \geq 2)$. C_1 is called a *proper sub-cluster* of C_2 if either $R_1 \subset R_2$ or $D_1 \subset D_2$. Pattern-based clusters have the following property.

Lemma 5 (Monotonicity). *Let $C = (R, D)$ be a δ-pCluster. Then, every sub-cluster (R', D') is a δ-pCluster.*

Proof. The Lemma follows the definition of δ-pCluster immediately. □

Clearly, mining the redundant sub-clusters is tedious and ineffective for analysis. Therefore, it is natural to mine only the "maximal clusters", i.e., the pClusters that are not sub-cluster of any other pClusters.

Definition 4.3 (maximal pCluster). *A δ-pCluster C is said* **maximal** *(or called a δ-MPC for short) if there exists no any other δ-pCluster C' such that C is a proper sub-cluster of C'.*

Problem Statement (mining maximal δ-pClusters). Given (1) a cluster threshold δ, (2) an attribute threshold min_a, and (3) an object threshold min_o, the task of *mining maximal δ-pClusters* is to find the complete set of maximal δ-pClusters with respect to min_a and min_o. □

4.2.4 Maximal Pattern-based Clusters As Skyline Pattern-based Clusters

Intuitively, maximal pattern-based clusters capture the non-redundant clusters. In pattern-based clustering analysis, two measures are often of particular interest, namely the set of objects and the set of attributes covered by a cluster. A cluster is called a skyline if it is not subsumed by some other cluster in both the set of objects and the set of attributes. Following the definition of maximal pattern-based clusters, we have the following result immediately.

Theorem 4.1. *Every maximal pattern-based cluster is a skyline cluster. That is, there exist no two maximal pattern-based clusters (R_1, D_1) and (R_2, D_2) such that $R_1 \subseteq R_2$ and $D_1 \subseteq D_2$ and at least one equivalence does not hold.* □

Theorem 4.1 shows that the set of maximal pattern-based clusters is not redundant. On the other hand, any non-maximal pattern-based clusters are redundant given the quality requirements of δ, the object threshold and the attribute threshold. The fact that a non-maximal pattern-based cluster C satisfies the quality requirements can be derived from any of the maximal pattern-based clusters C' that is a super-cluster of C. Please be note that, as discussed in Section 4.2.2, in pattern-based clustering, the quality requirements are given as input and the task is to search for all clusters that satisfy the quality requirements. We do not distinguish the difference in quality between clusters satisfying the quality requirements. Among all pattern-based clusters, their δ values may vary, but all of them are no greater than the threshold given as input.

Since maximal pattern-based clusters are skylines, they can be used to answer any queries about clusters with a preference function. For example, let us consider two forms of preference functions,

$$f_1 = \alpha \cdot \#\text{ objects} + \beta \cdot \#\text{ attributes}$$

where α and β are positive real numbers, and

$$f_2 = \#\text{ objects} \cdot \#\text{ attributes}.$$

We have the following result.

Corollary 4.1. *Any δ-pCluster maximizing preference function f_1 or f_2 must be a maximal pattern-based cluster.*

Proof. The corollary is straightforward. For any pCluster C that is not maximal, let C' be a maximal pCluster such that C is a sub-cluster of C'. Then, C' must have a higher preference value than C since C' either has more attributes or has more objects than C. □

4.2.5 *p-Clustering*: A δ-pCluster Mining Algorithm

A pattern-based clustering method (H. Wang et al., 2002), *p-Clustering*[1], is proposed. According to the extensive performance study reported in the paper, *p-Clustering* outperforms all previous methods.

The method works in the following three steps.

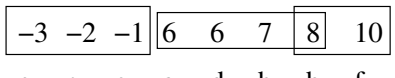
Step 1: Finding attribute-pair and object-pair MDSs.

Clearly, a pCluster must have at least two objects and two attributes. Intuitively, we can use those pClusters containing only two objects or two attributes to construct larger pClusters having more objects and attributes. An object/attribute-pair MDS (for maximal dimension set) is a maximal δ-MPC containing only two objects/attributes.

Given a pair of objects, how to compute the object-pair MDS efficiently? For example, Figure 4.4(a) shows the attribute values of two objects. The last row shows the differences of the attribute values.

Object	Attributes							
	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*
o_1	13	11	9	7	9	13	2	15
o_2	7	4	10	1	12	3	4	7
$o_1 - o_2$	6	7	−1	6	−3	10	−2	8

(a) The attribute values of two objects



(b) Finding MDS

Figure 4.4: Finding MDS for two objects.

To compute the object-pair MDS, *p-Clustering* sorts the attributes in the difference ascending order, as shown in Figure 4.4(b). Suppose $\delta = 2$. *P-Clustering* runs through the sorted list using a sliding window of variable width. Clearly, the attributes in the sliding window form a δ-pCluster provided the difference between the rightmost element and the leftmost one is no more than δ. For example, *p-Clustering* firstly sets the left edge of the sliding window at the left end of the sorted list, and moves the right edge of the window until it sees the first 6. The attributes in between, $\{e, g, c\}$, is the set of attributes of an object-pair MDS. Then, *p-Clustering*

[1]H. Wang et al. (2002) did not give a specific name to their algorithm. We call it *p-Clustering* since the main function in the algorithm is *pCluster*() and we want to distinguish the algorithm from the pclusters.

moves the left edge of the sliding window to attribute g, and repeats the process until the left end of the window runs through all elements in the list. In total, three MDSs can be found, i.e., $(\{o_1, o_2\}, \{e, g, c\})$, $(\{o_1, o_2\}, \{a, d, b, h\})$ and $(\{o_1, o_2\}, \{h, f\})$.

A similar method can be used to find the attribute-pair MDSs.

Step 2: Pruning Unpromising MDS.

In an object-pair MDS $(\{o_1, o_2\}, D)$, if the number of attributes in D is less than min_a, then o_1 and o_2 cannot appear together in any significant pCluster. Similarly, in an attribute-pair MDS $(R, \{a_1, a_2\})$, if the number of objects in R is less than min_o, then a_1 and a_2 cannot appear together in any significant pCluster. Such MDSs should be pruned.

After the pruning, each object-pair MDS must have at least min_a attributes, and each attribute-pair MDS must have at least min_o objects. Trivially, if min_a=2 or min_o=2, the mining is done. For $min_a > 2$ and $min_o > 2$, *p-Clustering* conducts the dual pruning between the object-pair MDSs and the attribute-pair MDSs. For example, suppose $(\{o_0, o_2\}, \{a_0, a_1, a_2\})$ is an object-pair MDS, but o_0 does not appear in the attribute-pair MDS of $\{a_0, a_2\}$, then MDS $(\{o_0, o_2\}, \{a_0, a_1, a_2\})$ can be pruned, since $\{a_0, a_2\}$ and o_0 cannot appear in the same significant pCluster. Such a dual pruning can be repeated until no MDS can be pruned further.

Step 3: Generating significant pClusters.

After the pruning in Step 2, *p-Clustering* inserts the surviving object-pair MDSs into a prefix tree. For each object-pair MDS, all attributes are sorted according to a global order $\mathcal{R}$ and then inserted into the tree. The two objects are registered in the last node of the path corresponding to the sorted attribute list. If two object-pair MDSs share the same prefix with respect to $\mathcal{R}$, then they share the corresponding path from the root in the tree. Figure 4.5 exemplifies a prefix tree.

Clearly, since every object-pair MDS surviving from the pruning must have at least min_a attributes, no object will be registered in any node whose depth is less than min_a. After all object-pair MDSs are inserted into the tree, *p-Clustering* treats each node in the tree whose depth is at least min_a as a candidate pCluster, and verifies whether the objects registered at the node really form a pCluster. Moreover, according to Lemma 5, if all objects at a node in the

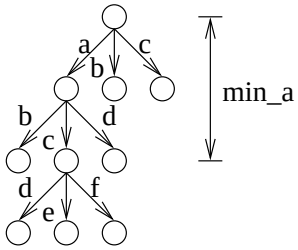


Figure 4.5: A prefix tree of object-pair MDSs.

tree form a pCluster, any ancestor of the node in the tree registering the same set of objects also form a pCluster. Therefore, a post-order traversal of the prefix tree is conducted to examine the nodes whose depths are no less than *min_a*, and generate the pClusters.

According to the performance study of *p-Clustering* (H. Wang et al., 2002) and our experimental results, this step is the bottleneck of the mining. For each node, *p-Clustering* has to examine the possible combinations of objects on the attributes registered in the path. The worst case complexity of prefix-tree depth-first clustering is exponential with respect to the number of attributes. This is the major cause that *p-Clustering* may not be efficient or scalable in large databases with many attributes.

4.2.6 Related Work

The study of pattern-based clustering is related to the previous work on subspace clustering and frequent itemset mining.

The meaning of clustering in high dimensional data sets is often unreliable (Beyer et al., 1999). Some recent studies (Agrawal et al., 1998; Aggarwal & Yu, 2000; Aggarwal et al., 1999; C. Cheng et al., 1999) focus on mining clusters embedded in some subspaces. For example, CLIQUE (Agrawal et al., 1998) is a density and grid based method. It divides the data into hyper-rectangular cells and use the dense cells to construct subspace clusters.

Subspace clustering can be used to semantically compress data. An interesting study (Jagadish et al., 1999) employs a randomized algorithm to find fascicles, the subsets of data that share similar values in some attributes. While their method is effective for compression, it does not guarantee the completeness of mining the clusters.

In some applications, global similarity-based clustering may not be effective. Still, strong correlations may exist among a set of objects even if they are far away from each other as measured by distance functions (such as Euclidean) used frequently in traditional clustering algorithms. Many scientific projects collect data in the form of Figure 4.1(a), and it is essential to identify clusters of objects that manifest coherent patterns. A variety of applications, including DNA microarray analysis, E-commerce collaborative filtering, will benefit from fast algorithms that can capture such patterns.

Y. Cheng & Church (2000) propose the biclustering model, which captures the coherence of genes and conditions in a sub-matrix of a DNA micro-array. J. Yang et al. (2002) develop a move-based algorithm to find biclusters more efficiently.

Recently, some variations of pattern-based clustering have been proposed. For example, the notion of OP-clustering (Liu & Wang, 2003) is developed. The idea is that, for an object, the list of dimensions sorted in the value ascending order can be used as its signature. Then, a set of objects can be put into a cluster if they share a part of their signature. OP-clustering can be viewed as a (very) loose pattern-based clustering. That is, every pCluster is an OP-cluster, but not vice versa.

On the other hand, a transaction database can be modelled as a binary matrix, where columns and rows stand for items and transactions, respectively. A cell $r_{i,j}$ is set to 1 if item j is contained in transaction i. Then, the problem of mining frequent itemsets (Agrawal et al., 1993) is to find subsets of rows and columns such that the sub-matrix is all 1's, and the number of rows is more than a given support threshold. If a minimum length constraint min_a is imposed to find only frequent itemsets of no less than min_a items, then it becomes a problem of mining 0-pClusters on binary data. Moreover, a maximal pattern-based cluster in the transaction binary matrix is a closed itemset (Pasquier et al., 1999). Interestingly, a maximal pattern-based cluster in this context can also be viewed as a formal concept, and the sets of objects and attributes are exactly the extent and intent of the concept, respectively (Ganter & Wille, 1996).

Although there are many efficient methods (Agarwal et al., 2001; Agrawal & Srikant, 1994; Han et al., 2004; Zaki et al., 1997) for frequent itemset mining, they cannot be extended to

handle the general pattern-based clustering problem since they can only handle the binary data.

4.2.7 Complexity

About the complexity of the problem of mining maximal pattern-based clusters, we have the following result.

Theorem 4.2. *The problem of finding the complete set of maximal pattern-based clusters is in NP-Complete.*

Proof. As shown in section refmaple-sec:relatedwork, mining frequent closed itemsets from a transaction database is a special case of mining 0-pClusters on binary data. In such a case, min_o is the minimum support threshold, and min_a is set to 1.

G. Yang (2004) showed that mining frequent closed itemsets is in NP-Complete. Thus, mining the complete set of maximal pattern-based clusters is in NP-Complete. □

4.3 Algorithms *MaPle* and *MaPle+*

In this section, we develop two novel pattern-based clustering algorithms, *MaPle* (for Maximal Pattern-based Clustering) and *MaPle+*. An early version of *MaPle* (Pei et al., 2003) is preliminarily proposed. *MaPle+* integrates some interesting heuristics on the top of *MaPle*.

We first overview the intuitions and the major technical features of *MaPle*, and then present the details.

4.3.1 An Overview of *MaPle*

Essentially, *MaPle* enumerates all the maximal pClusters systematically. It guarantees the completeness of the search, i.e., every maximal pCluster will be found. On the other hand, *MaPle* also guarantees the non-redundancy of the search, i.e., each combination of attributes and objects will be tested at most once.

The general idea of the search in *MaPle* is as follows. *MaPle* enumerates every combination of attributes systematically according to an order of attributes. For example, suppose that there are four attributes, a_1, a_2, a_3 and a_4 in the database, and the alphabetical order, i.e., a_1-a_2-a_3-a_4, is adopted. Let attribute threshold $min_a = 2$. For each subset of attributes, we can list the attributes alphabetically. Then, we can enumerate the subsets of two or more attributes according to the dictionary order, i.e., a_1a_2, $a_1a_2a_3$, $a_1a_2a_3a_4$, $a_1a_2a_4$, a_1a_3, $a_1a_3a_4$, a_1a_4, a_2a_3, $a_2a_3a_4$, a_2a_4, a_3a_4.

For each subset of attributes D, *MaPle* finds the maximal subsets of objects R such that (R,D) is a δ-pCluster. If (R,D) is not a sub-cluster of another pCluster (R,D') such that $D \subset D'$, then (R,D) is a maximal δ-pCluster. This "*attribute-first-object-later*" search is illustrated in Figure 4.6.

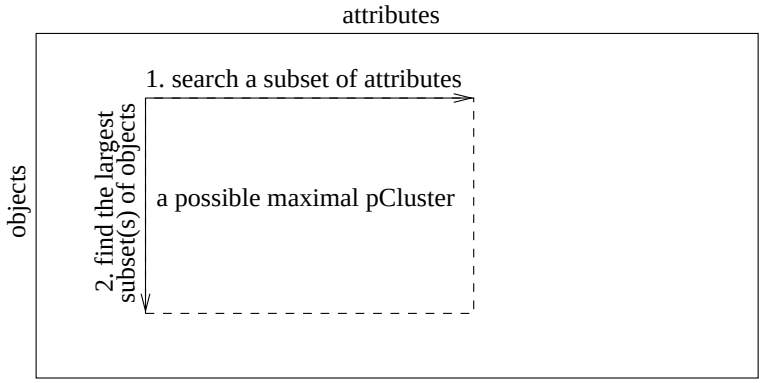


Figure 4.6: The attribute-first-object-later search.

There can be a huge number of coombinations of attributes. *MaPle* prunes many combinations unpromising for δ-pClusters. Following Lemma 5, for subset of attributes D, if there exists no subset of objects R such that (R,D) is a significant pCluster, then we do not need to search any superset of D. On the other hand, when search under a subset of attributes D, *MaPle* only checks those subsets of objects R such that (R,D') is a pCluster for every $D' \subset D$. Clearly, only subsets $R' \subseteq R$ may achieve δ-pCluster (R',D). Such pruning techniques are applied recursively. Thus, *MaPle* progressively refines the search step by step.

Moreover, *MaPle* also prunes searches that are unpromising to find maximal pClusters. It detects the attributes and objects that can be used to assemble a larger pCluster from the current

pCluster. If *MaPle* finds that the current subsets of attributes and objects as well as all possible attributes and objects together turn out to be a sub-cluster of a pCluster having been found before, then the recursive searches rooted at the current node are pruned, since it cannot lead to a maximal pCluster.

Why does MaPle *enumerate attributes first and then objects later, but not in the reverse way?* In real databases, the number of objects is often much larger than the number of attributes. In other words, the number of combinations of objects is often dramatically larger than the number of combinations of attributes. In the pruning using maximal pClusters discussed above, if the attribute-first-object-later approach is adopted, once a set of attributes and its descendants are pruned, all searches of related subsets of objects are pruned as well. Heuristically, the attribute-first-object-later search may bring a better chance to prune a more bushy search sub-tree.[2] Symmetrically, for data sets that the number of objects is far smaller than the number of attributes, a similar object-first-attribute-later search can be applied.

Essentially, we rely on MDSs to determine whether a subset of objects and a subset of attributes together form a pCluster. Therefore, as a preparation of the mining, we compute all non-redundant MDSs and store them as a database before we conduct the progressively refining, depth-first search.

Based on the above discussion, we have the framework of *MaPle* as shown in Figure 4.7.

Input: database *DB*, cluster threshold δ, attribute threshold min_a and object threshold min_o;
Output: the complete set of maximal δ-pClusters;
Method:
(1) compute and prune attribute-pair MDSs and object-pair MDSs; // Section 4.3.2
(2) progressively refining, depth-first search for maximal δ-pClusters; // Section 4.3.3

Figure 4.7: Algorithm *MaPle*.

Comparing to *p-Clustering*, *MaPle* has several advantages.

- First, in the third step of *p-Clustering*, for each node in the prefix tree, combinations of the object registered at the node will be explored to find pClusters. This can be expensive if

[2]However, there is no theoretical guarantee that the attribute-first-object-later search is optimal. There exist counter examples that object-first-attribute-later search wins.

there are many objects at a node. In *MaPle*, the information of pClusters is inherited from the "parent node" in the depth-first search and the possible combinations of objects can be reduced substantially. Moreover, once a subset of attributes D is determined hopeless for pClusters, the searches of any superset of D will be pruned.

- Second, *MaPle* prunes non-maximal pClusters. Many unpromising searches can be pruned in their early stages.

- Last, new pruning techniques are adopted in the computing and pruning of MDSs. That also speeds up the mining.

In the remainder of the section, we will explain the two steps of *MaPle* in detail.

4.3.2 Computing and Pruning MDSs

Given a database *DB* and a cluster threshold δ. A δ-pCluster $C_1 = (\{o_1, o_2\}, D)$ is called an *object-pair MDS* if there exists no δ-pCluster $C_1' = (\{o_1, o_2\}, D')$ such that $D \subset D'$. On the other hand, a δ-pCluster $C_2(R, \{a_1, a_2\})$ is called an *attribute-pair MDS* if there exists no δ-pCluster $C_2' = (R', \{a_1, a_2\})$ such that $R \subset R'$.

MaPle computes all attribute-pair MDSs as *p-Clustering* does. The method is illustrated in Figure 4.4(b). Limited by space, we omit the detailed algorithm here and only show the following example.

Example 4.1 (Running example – finding attribute-pair MDSs). Let us consider mining maximal pattern-based clusters in a database *DB* as shown in Figure 4.8(a). The database has 6 objects, namely $o_1, \ldots, o_6$, while each object has 5 attributes, namely $a_1, \ldots, a_5$.

Suppose $min_a = 3$, $min_o = 3$ and $\delta = 1$. For each pair of attributes, we calculate the attribute pair MDSs. The attribute-pair MDSs returned are shown in Figure 4.8(b). □

Generally, as shown in Figure 4.4, a pair of objects may have more than one object-pair MDS. Symmetrically, a pair of attributes may have more than one attribute-pair MDS.

Object	a_1	a_2	a_3	a_4	a_5
o_1	5	6	7	7	1
o_2	4	4	5	6	10
o_3	5	5	6	1	30
o_4	7	7	15	2	60
o_5	2	0	6	8	10
o_6	3	4	5	5	1

(a) The database

Objects	Attribute-pair
$\{o_1, o_2, o_3, o_4, o_6\}$	$\{a_1, a_2\}$
$\{o_1, o_2, o_3, o_6\}$	$\{a_1, a_3\}$
$\{o_1, o_2, o_6\}$	$\{a_1, a_4\}$
$\{o_1, o_2, o_3, o_6\}$	$\{a_2, a_3\}$
$\{o_1, o_2, o_6\}$	$\{a_2, a_4\}$
$\{o_1, o_2, o_6\}$	$\{a_3, a_4\}$

(b) The attribute-pair MDSs

Figure 4.8: The database and attribute-pair MDSs in our running example.

We can also generate all the object-pair MDSs similarly. However, if we utilize the information on the number of occurrences of objects and attributes in the attribute-pair MDSs, the calculation of object-pair MDSs can be speeded up.

Lemma 6 (Pruning MDSs). *Given a database DB and a cluster threshold* δ*, object threshold* min_o *and attribute threshold* min_a*.*

1. *An attribute a cannot appear in any significant* δ*-pCluster if a appears in less than* $\frac{min_o \cdot (min_o - 1)}{2}$ *object-pair MDSs, or appears in less than* $(min_a - 1)$ *attribute-pair MDSs;*

2. *An object o cannot appear in any significant* δ*-pCluster if o appears in less than* $\frac{min_a \cdot (min_a - 1)}{2}$ *attribute-pair MDSs, or appears in less than* $(min_o - 1)$ *object-pair MDSs.*

Proof. We prove the first half of the lemma. The second half can be proved dually. Let (R, D) be a significant δ-pCluster, and $a \in D$ be an attribute.

For any objects $o_i, o_j \in R$, there must be an object-pair MDS $(\{o_i, o_j\}, D_{ij})$ such that $a \in D_{ij}$. There are at least $\frac{\|R\|(\|R\|-1)}{2}$ such object-pair MDSs. Since $\|R\| \geq min_o$, a appears in at least $\frac{min_o \cdot (min_o - 1)}{2}$ object-pair MDSs. On the other hand, following Lemma 5, for any attribute $a' \in D$, $(R, \{a, a'\})$ is also a δ-pCluster. Therefore, there must be some attribute-pair MDS $(R', \{a, a'\})$ such that $R \subseteq R'$. There are $(\|D\| - 1)$ such attribute-pair MDSs. Since $\|D\| \geq min_a$, a appears in at least $(min_a - 1)$ attribute-pair MDSs. □

Example 4.2 (Pruning using Lemma 6). Let us check the attribute-pair MDSs in Figure 4.8(b). Object o_5 does not appear in any attribute-pair MDS, and object o_4 appears in only 1 attribute-

pair MDS. According to Lemma 6, o_4 and o_5 cannot appear in any significant δ-pCluster. Therefore, we do not need to check any object-pairs containing o_4 or o_5.

There are 6 objects in the database. Without this pruning, we have to check $\frac{6\times5}{2} = 15$ pairs of objects. With this pruning, only four objects, o_1, o_2, o_3 and o_6 survive. Thus, we only need to check $\frac{4\times3}{2} = 6$ pairs of objects. A 60% of the original searches is pruned.

Moreover, since attribute a_5 does not appear in any attribute-pair MDS, it cannot appear in any significant δ-pCluster. The attribute can be pruned. That is, when generating the object-pair MDS, we do not need to consider attribute a_4.

In summary, after the pruning, only attributes a_1, a_2, a_3 and a_4, and objects o_1, o_2, o_3 and o_6 survive. We use these attributes and objects to generate object-pair MDSs. The result is shown in Figure 4.9(a). In method *p-Clustering*, it uses all attributes and objects to generate object-pair MDSs. The result is shown in Figure 4.9(b). As can be seen, not only the computation cost in *MaPle* is less, the number of object-pair MDSs in *MaPle* is also one less than that in method *p-Clustering*. □

Object-pair	Attributes
$\{o_1, o_2\}$	$\{a_1, a_2, a_3, a_4\}$
$\{o_1, o_3\}$	$\{a_1, a_2, a_3\}$
$\{o_1, o_6\}$	$\{a_1, a_2, a_3, a_4\}$
$\{o_2, o_3\}$	$\{a_1, a_2, a_3\}$
$\{o_2, o_6\}$	$\{a_1, a_2, a_3, a_4\}$
$\{o_3, o_6\}$	$\{a_1, a_2, a_3\}$

(a) Object-pair MDSs in *MaPle*.

Object-pair	Attributes
$\{o_1, o_2\}$	$\{a_1, a_2, a_3, a_4\}$
$\{o_1, o_6\}$	$\{a_1, a_2, a_3, a_4\}$
$\{o_2, o_3\}$	$\{a_1, a_2, a_3\}$
$\{o_1, o_3\}$	$\{a_1, a_2, a_3\}$
$\{o_2, o_6\}$	$\{a_1, a_2, a_3, a_4\}$
$\{o_3, o_4\}$	$\{a_1, a_2, a_4\}$
$\{o_3, o_6\}$	$\{a_1, a_2, a_3\}$

(b) Object-pair MDSs in method *p-Clustering*

Figure 4.9: Pruning using Lemma 6.

Once we get the initial object-pair MDSs and attribute-pair MDSs, we can conduct a mutual pruning between the object-pair MDSs and the attribute-pair MDSs, as method *p-Clustering* does. Furthermore, Lemma 6 can be applied in each round to get extra pruning. The pruning algorithm is shown in Figure 4.10.

```
(1)  REPEAT
(2)      count the number of occurrences of objects and attributes in the attribute-pair MDSs;
(3)      apply Lemma 6 to prune objects and attributes;
(4)      remove object-pair MDSs containing less than min_a attributes;
(5)      count the number of occurrences of objects and attributes in the object-pair MDSs;
(6)      apply Lemma 6 to prune objects and attributes;
(7)      remove attribute-pair MDSs containing less than min_o objects;
(8)  UNTIL no pruning takes place
```

Figure 4.10: The algorithm of pruning MDSs.

4.3.3 Progressively Refining, Depth-first Search of Maximal pClusters

The algorithm of the progressively refining, depth-first search of maximal pClusters is shown in Figure 4.11. We will explain the algorithm step by step in this subsection.

```
(1)  let n be the number of attributes; make up an attribute list AL = a_1-···-a_n;
(2)  FOR i = 1 TO n − min_o + 1 DO                          //Theorem 4.3, item 1
(3)      FOR j = i + 1 TO n − min_o + 2 DO
(4)          find attribute-maximal pClusters (R, {a_i, a_j});      //Section 4.3.3
(5)          FOR EACH lcoal maximal pCluster (R, {a_i, a_j}) DO
(6)              call search(R, {a_i, a_j});
(7)          END FOR EACH
(8)      END FOR
(9)  END FOR
(10)
(11) FUNCTION search(R, D);                                 // (R, D) is a attribute-maximal pCluster.
(12)     compute PD, the set of possible attributes;        //Optimization 1 in Section 4.3.3
(13)     apply optimizations in Section 4.3.3 to prune, if possible;
(14)     FOR EACH attribute a ∈ PD DO                       //Theorem 4.3, item 2
(15)         find attribute-maximal pClusters (R', D ∪ {a});        //Section 4.3.3
(16)         FOR EACH attribute-maximal pCluster (R', D ∪ {a}) DO
(17)             call search(R', D ∪ {a});
(18)         END FOR EACH
(19)         IF (R', D ∪ {a}) is not a subcluster of some maximal pCluster having been found
(20)         THEN output (R', D ∪ {a});
(21)     END FOR EACH
(22)     IF (R, D) is not a subcluster of some maximal pCluster having been found
(23)     THEN output (R, D);
(24) END FUNCTION
```

Figure 4.11: The algorithm of projection-based search.

Dividing Search Space

By a list of attributes, we can enumerate all combinations of attributes systematically. The idea is shown in the following example.

Example 4.3 (Enumeration of combinations of attributes). In our running example, there are four attributes survived from the pruning: a_1, a_2, a_3 and a_4. We list the attributes in any subset of attributes in the order of a_1-a_2-a_3-a_4. Since $min_a = 3$, every maximal δ-pCluster should have at least 3 attributes. We divide the complete set of maximal pClusters into 3 exclusive subsets according to the first two attributes in the pClusters: (1) the ones having attributes a_1 and a_2, (2) the ones having attributes a_1 and a_3 but no a_2, and (3) the ones having attributes a_2 and a_3 but no a_1. □

Since a pCluster has at least 2 attributes, *MaPle* first partitions the complete set of maximal pClusters into exclusive subsets according to the first two attributes, and searches the subsets one by one in the depth-first manner. For each subset, *MaPle* further divides the pClusters in the subset into smaller exclusive sub-subsets according to the third attributes in the pClusters, and search the sub-subsets. Such a process proceeds recursively until all the maximal pClusters are found. This is implemented by line (1)-(3) and (14) in Figure 4.11. The correctness of the search is justified by the following theorem.

Theorem 4.3 (Completeness and non-redundancy of *MaPle*). *Given an attribute-list AL* : a_1-$\cdots$-a_m*, where m is the number of attributes in the database. Let* min_a *be the attribute threshold.*

1. *All attributes in each pCluster are listed in the order of AL. Then, the complete set of maximal* δ*-pClusters can be divided into* $\frac{(m-min_a+2)(m-min_a+1)}{2}$ *exclusive subsets according to the first two attributes in the pClusters.*

2. *The subset of maximal pClusters whose first* 2 *attributes are* a_i *and* a_j *can be further divided into* $(m - min_a + 3 - j)$ *subsets: the* k^{th} $(1 \leq k \leq (m - j - min_a - 1))$ *subset contains pClusters whose first* 3 *attributes are* a_i, a_j *and* a_{j+k}.

Proof. We prove the first item of the theorem. The second item can be shown similarly.

Trivially, every pCluster must have at least 2 attributes. Clearly, according to *AL*, the first two attributes in every pCluster are determined. Therefore, there exists no a pCluster C such that C is in more than one subset, i.e., the subsets are exclusive. Since a maximal pCluster C must be significant, C must have at least min_a attributes. The first attribute of C must be from a_1 to a_{m-min_a+1}. Suppose the first attribute of C is a_i. The, the second attribute of C must be from a_{i+1} to a_{m-min_a+2}. In total, there are $\frac{(m-min_a+2)(m-min_a+1)}{2}$ possible combinations. So we have the theorem. □

Finding Attribute-maximal pClusters

Now, the problem becomes how to find the maximal δ-pClusters on the subsets of attributes. For each subset of attributes D, we will find the maximal subsets of objects R such that (R,D) is a pCluster. Such a pCluster is a maximal pCluster if it is not a sub-cluster of some others.

Given a set of attributes D such that $(\|D\| \geq 2)$. A pCluster (R,D) is called a *attribute-maximal δ-pCluster* if there exists no any δ-pCluster (R',D) such that $R \subset R'$. In other words, a attribute-maximal pCluster is maximal in the sense that no more objects can be included so that the objects are still coherent on the same subset of attributes. For example, in the database shown in Figure 4.8(a), $(\{o_1,o_2,o_3,o_6\},\{a_1,a_2\})$ is a attribute-maximal pCluster for subset of attributes $\{a_1,a_2\}$.

Clearly, a maximal pCluster must be a attribute-maximal pCluster, but not vice versa. In other words, if a pCluster is not a attribute-maximal pCluster, it cannot be a maximal pCluster.

Given a subset of attributes D, how can we find all attribute-maximal pClusters efficiently? We answer this question in two cases.

If D has only two attributes, then the attribute-maximal pClusters are the attribute-pair MDSs for D. Since the MDSs are computed and stored before the search, they can be retrieved immediately.

Now, let us consider the case where $(\|D\| \geq 3)$. Suppose $D = \{a_{i_1},\ldots,a_{i_k}\}$ where the attributes in D are listed in the order of attribute-list *AL*. Intuitively, (R,D) is a pCluster if R is

shared by attribute-pair MDSs from any two attributes from D. (R,D) is a attribute-maximal pCluster if R is a maximal set of objects.

One subtle point here is that, in general, there can be more than one attribute-pair MDS for given attributes a_u, a_v. Thus, there can be more than one attribute-maximal pCluster on a subset of attributes D. Technically, (R,D) is a attribute-maximal pCluster if for each pair of attributes $\{a_u,a_v\} \subset D$, there exists an attribute-pair MDS $(\{a_u,a_v\},R_{uv})$, such that $R = \bigcap_{\{a_u,a_v\}\subset D} R_{uv}$. Recall that *MaPle* searches the combinations of attributes in the depth-first manner, all attribute-maximal pClusters for subset of attributes $D-\{a\}$ is found before we search for D, where a is the last attribute in D according to the attribute list. Therefore, we only need to find the subset of objects in a attribute-maximal pCluster of $D-\{a\}$ that are shared by attribute-pair MDSs of a_{i_j}, a_{i_k} $(j<k)$.

Pruning and Optimizations

Several optimizations can be used to prune the search so that the mining can be more efficient. We explain them as follows.

Optimization 1: Only *possible attributes* should be considered to get larger pClusters.

Suppose that (R,D) is a attribute-maximal pCluster. *For every attribute a such that a is behind all attributes in D in the attribute-list, can we always find a significant pCluster $(R',D\cup\{a\})$ such that $R' \subseteq R$?*

If $(R',D\cup\{a\})$ is significant, i.e., has at least *min_o* objects, then a must appear in at least $\frac{min_o(min_o-1)}{2}$ object-pair MDSs $(\{o_i,o_j\},D_{ij})$ such that $\{o_i,o_j\} \subseteq R'$. In other words, for an attribute a that appears in less than $\frac{min_o(min_o-1)}{2}$ object-pair MDSs of objects in R, there exists no attribute-maximal pCluster with respect to $D\cup\{a\}$.

Based on the above observation, an attribute a is called a *possible attribute* with respect to attribute-maximal pCluster (R,D) if a appears in $\frac{min_o(min_o-1)}{2}$ object-pair MDSs $(\{o_i,o_j\},D_{ij})$ such that $\{o_i,o_j\} \subseteq R$. In line (12) of Figure 4.11, we compute the possible attributes and only those attributes are used to extend the set of attributes in pClusters.

Optimization 2: Pruning local maxiaml pClusters having insufficient possible attributes.

Suppose that (R,D) is a attribute-maximal pCluster. Let PD be the set of possible attributes with respect to (R,D). Clearly, if $\|D \cup PD\| < min_a$, then it is impossible to find any maximal pCluster of a subset of R. Thus, such a attribute-maximal pCluster should be discarded and all the recursive search can be pruned.

Optimization 3: Extracting common attributes from possible attribute set directly.

Suppose that (R,D) is a attribute-maximal pCluster with respect to D, and D' is the corresponding set of possible attributes. If there exists an attribute $a \in D'$ such that for every pair of objects $\{o_i, o_j\}$, $\{a\} \cup D$ appears in an object pair MDS of $\{o_i, o_j\}$, then we immediately know that $(R, D \cup \{a\})$ must be a attribute-maximal pCluster with respect to $D \cup \{a\}$. Such an attribute is called a *common attribute* and should be extracted directly.

Example 4.4 (Extracting common attributes). In our running example, $(\{o_1, o_2, o_3, o_6\}, \{a_1, a_2\})$ is a attribute-maximal pCluster with respect to $\{a_1, a_2\}$. Interestingly, as shown in Figure 4.9(a), for every object pair $\{o_i, o_j\} \subset \{o_1, o_2, o_3, o_6\}$, the object-pair MDS contains attribute a_3. Therefore, we immediately know that $(\{o_1, o_2, o_3, o_6\}, \{a_1, a_2, a_3\})$ is a attribute-maximal pCluster. □

Optimization 4: Prune non-maximal pClusters.

Our goal is to find maximal pClusters. If we can find that the recursive search on a attribute-maximal pCluster cannot lead to a maximal pCluster, the recursive search thus can be pruned. The earlier we detect the impossibility, the more search efforts can be saved.

We can use the *dominant attributes* to detect the impossibility. We illustrate the idea in the following example.

Example 4.5 (Using dominant attributes to detect non-maximal pClusters). Again, let us consider our running example. Let us try to find the maximal pClusters whose first two attributes are a_1 and a_3. Following the above discussion, we identify a attribute-maximal pCluster $(\{o_1, o_2, o_3, o_6\}, \{a_1, a_3\})$.

One interesting observation can be made from the object-pair MDSs on objects in $\{o_1, o_2, o_3, o_6\}$ (Figure 4.9(a)): attribute a_2 appears in every object pair. We called a_2 a *dominant attribute*. That

means $\{o_1, o_2, o_3, o_6\}$ also coherent on attribute a_2. In other words, we cannot have a maximal pCluster whose first two attributes are a_1 and a_3, since a_2 must also be in the same maximal pCluster. Thus, the search of maximal pClusters whose first two attributes are a_1 and a_3 can be pruned. □

The idea in Example 4.5 can be generalized. Suppose (R, D) is a attribute-maximal pCluster. If there exists an attribute a such that a is before the last attribute in D according to the attribute-list, and $\{a\} \cup D$ appears in an object-pair MDS $(\{o_i, o_j\}, D_{ij})$ for every $(\{o_i, o_j\} \subseteq R)$, then the search from (R, D) can be pruned, since there cannot be a maximal pCluster having attribute set D but no a. Attribute a is called a *dominant attribute* with respect to (R, D).

4.3.4 *MaPle+*: Further Improvements

MaPle+ is an enhanced version of *MaPle*. In addition to the techniques discussed above, the following two ideas are implemented in *MaPle+*.

Block-based Pruning of Attribute-pair MDSs

In Step 2 of algorithm *p-Clustering* (please see Section 4.2.5) and *MaPle* (please see Section 4.3.2), an MDSs can be pruned if it cannot be used to form larger pClusters. The pruning is based on comparing an MDS with the other MDSs.

Since there can be a large number of MDSs, the pruning may not be efficient. Instead, we can adopt a block-based pruning as follows.

For an attribute a, all attribute-pair MDSs that a is an attribute form the *a-block*. We consider the blocks of attributes in the attribute-list order.

For the first attribute a_1, the a_1-block is formed. Then, for an object o, if o appears in any significant pCluster that has attribute a_1, o must appear in at least $(min_a - 1)$ different attribute-pair MDSs in the a_1-block. In other words, we can remove an object o from the a_1-block MDSs if its count in the a_1-block is less than $(min_a - 1)$. After removing the objects, the attribute-pair MDSs in the block that do not have at least $(min_o - 1)$ objects can also be removed safely.

Moreover, according to Lemma 6, if there are less than $(min_a - 1)$ MDSs in the resulted a_1-block, then a_1 cannot appear in any significant pCluster, and thus all the MDSs in the block can be removed.

The blocks can be considered one by one. Such a block-based pruning is more effective. In Section 4.3.2, we prune an object from attribute-pair MDSs if it appears in less than $\frac{min_a \cdot (min_a - 1)}{2}$ different attribute-pair MDSs (Lemma 6). In the block-based pruning, we consider pruning an object with respect to every possible attribute. It can be shown that any object pruned by Lemma 6 must also be pruned in some block, but not vice versa, as shown in the following example.

Example 4.6 (Block-based pruning of attribute-pair MDSs). Suppose we have the attribute-pair MDSs as shown in Figure 4.12, and $min_o = min_a = 3$.

Attribute-pairs	objects
$\{a_1, a_2\}$	$\{o_1, o_2, o_4\}$
$\{a_1, a_3\}$	$\{o_2, o_3, o_4\}$
$\{a_1, a_4\}$	$\{o_2, o_4, o_5\}$
$\{a_2, a_3\}$	$\{o_1, o_2, o_3\}$
$\{a_2, a_4\}$	$\{o_1, o_3, o_4\}$
$\{a_2, a_5\}$	$\{o_2, o_3, o_5\}$

Figure 4.12: The attribute-pair MDSs in Example 4.6.

In the a_1-block, which contains the first three attribute-pair MDSs in the table, objects o_1, o_3 and o_5 can be pruned. Moreover, all attribute-pair MDSs in the a_1-block can be removed.

However, in *MaPle*, since o_1 appears 3 times in all the attribute-pair MDSs, it cannot be pruned by Lemma 6, and thus attribute-pair MDS $(\{a_1, a_2\}, \{o_1, o_2, o_4\})$ cannot be pruned, either. □

The block-based pruning is also more efficient. To use Lemma 6 to prune in *MaPle*, we have to check both the attribute-pair MDSs and the object-pair MDSs mutually. However, in the block-based pruning, we only have to look at the attribute-pair MDSs in the current block.

Computing Attribute-pair MDSs Only

In many data sets, the numbers of objects and attributes are different dramatically. For example, in the microarray data sets, there are often many genes (thousands or even tens of thousands), but very few samples (up to one hundred). In such cases, a significant part of the runtime in both *p-Clustering* and *MaPle* is to compute the object-pair MDSs.

Clearly, computing object-pair MDSs for a large set of objects is very costly. For example, for a data set of $10,000$ objects, we have to consider $10000 \times 9999 \div 2 = 49,995,000$ object pairs!

Instead of computing those object-pair MDSs, we develop a technique to compute only the attribute-pair MDSs. The idea is that we can compute the attribute-maximal pClusters on-the-fly without materializing the object-pair MDSs.

Example 4.7 (Computing attribute-pair MDS's only). Consider the attribute-pair MDS's in Figure 4.8(b) again. We can compute the attribute-maximal pCluster for attribute set $\{a_1, a_2, a_3\}$ using the attribute-pair MDS's only.

We observe that an object pair o_u are in an attribute-maximal pCluster of $\{a_1, a_2, a_3\}$ if and only if there exist three attribute-pair MDS's for $\{a_1, a_2\}$, $\{a_1, a_3\}$, and $\{a_2, a_3\}$, respectively, such that $\{o_u, o_v\}$ are in the object sets of all those three attribute-pair MDS's. Thus, the intersection of the three object sets in those three attribute-pair MDS's is the set of objects in the attribute-maximal pCluster.

In this example, $\{a_1, a_2\}$, $\{a_1, a_3\}$, and $\{a_2, a_3\}$ have only one attribute-pair MDS, respectively. The intersection of their object sets are $\{o_1, o_2, o_3, o_6\}$. Therefore, the attribute-maximal pCluster is $(\{o_1, o_2, o_3, o_6\}, \{a_1, a_2, a_3\})$. □

When the number of objects is large, computing the attribute-maximal pClusters directly from attribute-pair MDS's and smaller attribute-maximal pClusters can avoid the costly materialization of object-pair MDS's. The computation can be conducted level-by-level from smaller attribute sets to their supersets.

Generally, if a set of attributes D has multiple attribute-maximal pClusters, then its superset

D' may also have multiple attribute-maximal pClusters. For example, suppose $\{a_1, a_2\}$ has attribute-pair MDS's $(R_1, \{a_1, a_2\})$ and $(R_2, \{a_1, a_2\})$, and $(R_3, \{a_1, a_3\})$ and $(R_4, \{a_2, a_3\})$ are attribute-pair MDS's for $\{a_1, a_3\}$ and $\{a_1, a_3\}$, respectively. Then, $(R_1 \cap R_3 \cap R_4, \{a_1, a_2, a_3\})$ and $(R_2 \cap R_3 \cap R_4, \{a_1, a_2, a_3\})$ should be checked. If the corresponding object set has at least min_o objects, then the pCluster is an attribute-maximal pCluster. We also should check whether $(R_1 \cap R_3 \cap R_4) = (R_2 \cap R_3 \cap R_4)$. If so, we only need to keep one attribute-maximal pCluster for $\{a_1, a_2, a_3\}$.

To compute the intersections efficiently, the sets of objects can be represented as bitmaps. Thus, the intersection operations can be implemented using the bitmap `AND` operations.

4.4 Empirical Evaluation

We test *MaPle*, *MaPle+* and *p-Clustering* extensively on both synthetic and real life data sets. In this section, we report the results.

MaPle and *MaPle+* are implemented using C/C++. We obtained the executable of the improved version of *p-Clustering* (H. Wang et al., 2002) from the authors. Please note that the authors of *p-Clustering* improved their algorithm dramatically after their publication in SIGMOD'02. The authors of *p-Clustering* also revised the program so that only maximal pClusters are detected and reported. Thus, the output of the two methods are comparable directly. All the experiments are conducted on a PC with a P4 1.2 GHz CPU and 384 M main memory running a Microsoft Windows XP operating system.

4.4.1 The Data Sets

The algorithms are tested against both synthetic and real life data sets. Synthetic data sets are generated by a synthetic data generator (H. Wang et al., 2002). The data generator takes the following parameters to generate data sets: (1) the number of objects; (2) the number of attributes; (3) the average number of rows of the embedded pClusters; (4) the average number of columns; and (5) the number of pClusters embedded in the data sets. The synthetic data

generator can generate only perfect pClusters, i.e., $\delta = 0$.

We also report the results on a real data set, the Yeast microarray data set (Tavazoie et al., 2000). This data set contains the expression levels of $2,884$ genes under 17 conditions. The data set is preprocessed as described in the paper by H. Wang et al. (2002).

4.4.2 Results on Yeast Data Set

The first issue we want to examine is whether there exist significant pClusters in real data sets. We test on the Yeast data set. The results are shown in Figure 4.13. From the results, we can obtain the following interesting observations.

δ	min_a	min_o	# of max-pClusters	# of pClusters
0	9	30	5	5520
0	7	50	11	N/A
0	5	30	9370	N/A

Figure 4.13: Number of pClusters on Yeast raw data set.

- There are significant pClusters existing in real data. For example, we can find pure pCluster (i.e., $\delta = 0$) containing more than 30 genes and 9 attributes in Yeast data set. That shows the effectiveness and utilization of mining maximal pClusters in the real data sets.

- While the number of maximal pClusters is often small, the number of all pClusters can be huge, since there are many different combinations of objects and attributes as subclusters to the maximal pClusters. This shows the effectiveness of the notation of maximal pClusters.

- Among the three cases shown in Figure 4.13, *p-Clustering* can only finish in the first case. In the other two cases, it cannot finish and outputs a huge number of pClusters that overflow the hard disk. In contrast, *MaPle* and *MaPle+* can finish and output a small number of pClusters, which cover all the pClusters found by *p-Clustering*.

To test the efficiency of mining the Yeast data set with respect to the tolerance of noise, we fix the thresholds of $min_a = 6$ and $min_o = 60$, and vary the δ from 0 to 4. The results are shown in Figure 4.14.

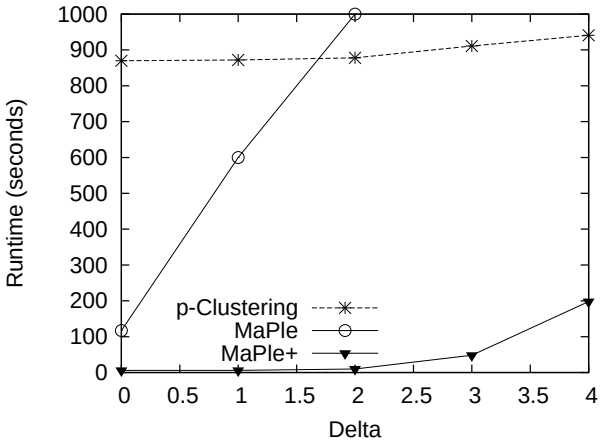


Figure 4.14: Runtime vs. δ on the Yeast data set, $min_a = 6$ and $min_o = 60$.

As shown, both *p-Clustering* and *MaPle+* are scalable on the real data set with respect to δ. When δ is small, *MaPle* is fast. However, it scales poorly with respect to δ. The reason is that, as the value of δ increases, a subset of attribute has more and more attribute-maximal pClusters on average. Similarly, there are more and more object-pair MDS's. Managing a large number of MDS's and conducting iteratively pruning still can be costly. The block-based pruning technique and the technique of computing attribute-maximal pClusters from attribute-pair MDS's, as described in Section 4.3.4, helps *MaPle+* to reduce the cost effectively. Thus, *MaPle+* is substantially faster than *p-Clustering* and *MaPle*.

4.4.3 Results on Synthetic Data Sets

We test the scalability of the algorithms on the three parameters, the minimum number of objects min_o, the minimum number of attributes min_a in pClusters, and δ. In Figure 4.15, the runtime of the algorithms versus min_o is shown. The data set has 6000 objects and 30 attributes.

As can be seen, all the three algorithms are in general insensitive to parameter min_o, but *MaPle+* is much faster than *p-Clustering* and *MaPle*. The major reason that the algorithms are insensitive is that the number of pClusters in the synthetic data set does not changes dramatically

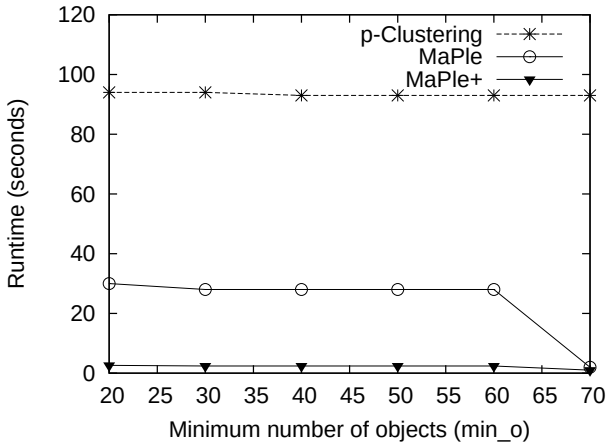


Figure 4.15: Runtime vs. minimum number of objects in pClusters.

as min_o decreases and thus the overhead of the search does not increase substantially. Please note that we do observe the slight increases of runtime in all the three algorithms as min_o goes down.

One interesting observation here is that, when $min_o > 60$, the runtime of *MaPle* decreases significantly. The runtime of *MaPle+* also decreases from 2.4 seconds to 1 second. That is because there is no pCluster in such a setting. *MaPle+* and *MaPle* can detect this in an early stage and thus can stop early.

We observe the similar trends on the runtime versus parameter min_a. That is, both algorithms are insensitive to the minimum number of attributes in pClusters, but *MaPle* is faster than *p-Clustering*. The reasoning similar to that on min_o holds here.

We also test the scalability of the algorithms on δ. The result is shown in Figure 4.16. As shown, both *MaPle+* and *pClustering* are scalable with respect to the value of δ, while *MaPle* is efficient when the δ is small. When the δ value becomes large, the performance of *MaPle* becomes poor. The reason is as analyzed before: when the value of δ increases, some attribute pairs may have multiple MDS's and some object pairs may have multiple MDS's. *MaPle*has to check many combinations. *MaPle+* uses the block-based pruning technique to reduce the cost substantially. Among the three algorithms, *MaPle+* is clearly the best.

We test the scalability of the three algorithms on the number of objects in the data sets. The result is shown in Figure 4.17. The data set contains 30 attributes, where there are 30 embedded

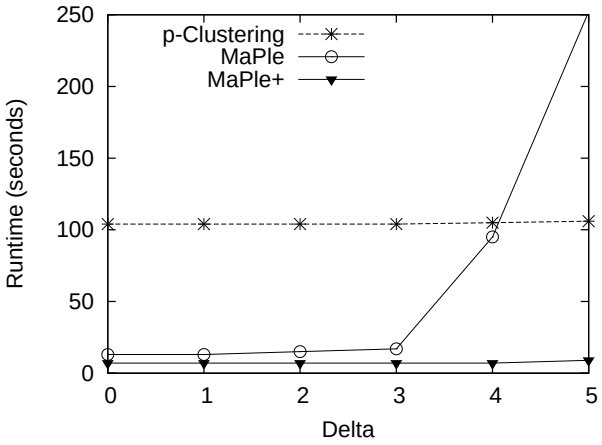


Figure 4.16: Runtime vs. δ.

clusters. We fix $min_a = 5$ and set $min_o = n_{obj} \cdot 1\%$, where n_{obj} is the number of objects in the data set. $\delta = 1$.

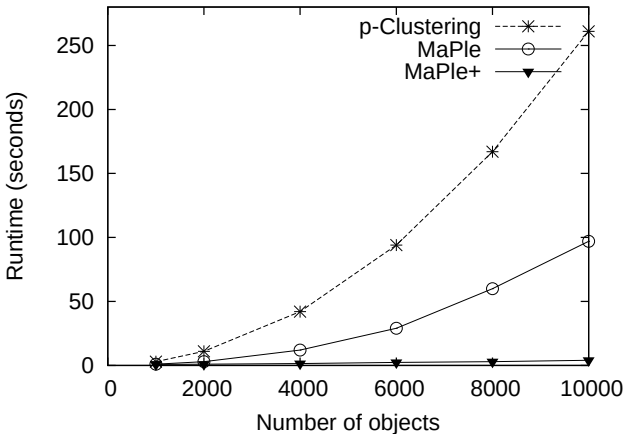


Figure 4.17: Scalability with respect to the number of objects in the data sets.

The result in Figure 4.17 clearly shows that *MaPle* performs substantially better than *p-Clustering* in mining large data sets. *MaPle+* is up to two orders of magnitudes faster than *p-Clustering* and *MaPle*. The reason is that both *p-Clustering* and *MaPle* use object-pair MDS's in the mining. When there are 10000 objects in the database, there are $\frac{10000 \times 9999}{2} = 49995000$ object-pairs. Managing a large database of object-pair MDS's is costly. *MaPle+* only uses attribute-pair MDS's in the mining. In this example, there are only $\frac{30 \times 29}{2} = 435$ attribute pairs. Thus, *MaPle+* does not suffer from the problem.

To further understand the difference, Figure 4.17 shows the numbers of local maximal

pClusters searched by *MaPle*and *MaPle+*. As can be seen, *MaPle+* searches substantially less than *MaPle*. That partially explains the difference of performance of the two algorithms.

We also test the scalability of three algorithms on the number of attributes. The result is shown in Figure 4.18. In this test, the number of objects is fixed to $3,000$ and there are 30 embedded pClusters. We set $min_o = 30$ and $min_a = n_{attr} \cdot 20\%$, where n_{attr} is the number of attributes in the data set.

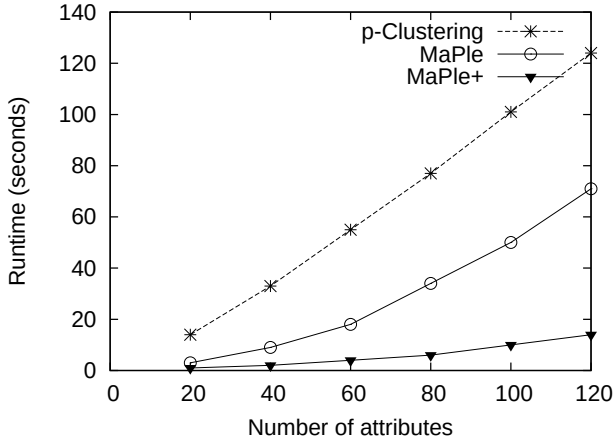


Figure 4.18: Scalability with respect to the number of attributes in the data sets.

The curves show that all the three algorithms are approximately linearly scalable with respect to number of attributes, and *MaPle+* performs consistently better than *p-Clustering* and *MaPle*.

In summary, from the tests on synthetic data sets, we can see that *MaPle* outperforms *p-Clustering* clearly. *MaPle* is efficient and scalable in mining large data sets.

Chapter 5

Conclusion

In this dissertation we described emerging challenges and our approaches to tackle them in data warehousing and OLAP. As seen so far, a data warehousing and OLAP are useful facilities for a decision making of users. A data warehouse stores summarized and compressed information and data cubes and iceberg cubes are tools for access to a data warehouse efficiently. We delve into challenges which each area of data warehouse technology faces. This chapter summarizes the contributions of this dissertation and discusses potential future research topics.

Mining iceberg cubes is to compute aggregates to find aggregate values satisfying specified threshold. Since a data warehouse has large amount of data in general, the computation of iceberg cubes should use little memory and fewer scans over data. In order to compute iceberg cubes efficiently, we focus on an assumption of previous methods to compute iceberg cubes from a data warehouse, which is the computation of iceberg cubes based on a universal base table. Materializing the universal base table costs high in time and space due to the redundancy of dimensional data in the universal base table and multiple scans of dimensional table. A new method is required to compute iceberg cube without materializing universal base table. It contributes to the computation of iceberg cubes technology since it reduces the cost of computation of iceberg cubes.

An algorithm CTC(Cross Table Cubing) is developed. Unlike all of the previous methods, *CTC* avoids materializing the universal base table. Instead, it computes local iceberg cells and

derives the global iceberg cells from local ones. It removes redundancy and multiple scans of dimensional data. It does not access dimensional tables after computation of local iceberg cells. The investigation and the experimental results clearly indicate that *CTC* is efficient and scalable in computing iceberg cubes for large data warehouses. It is consistently more efficient and more scalable than *BUC*. We also show how the techniques in *CTC* can be generalized to handle more complicated schemas, such as snowflake schema.

Online warehousing data streams and answering ad hoc aggregate queries are interesting and challenging research problems with broad applications. Since a data stream has high rate of data input and it is often infeasible to maintain all data in memory, online data warehousing is required to maintain the recent data in a sliding window, and provide online answers to ad hoc aggregate queries over the current sliding window. We propose a novel *PAT* data structure, which is a prefix tree and has links facilitating online ad hoc query answers efficiently. It stores a subset of aggregate cells, *prefix aggregates cells* and *infix aggregate cells*, from the recent data in a slinding window. Efficient algorithms are developed to construct and incrementally maintain a *PAT* over a data stream, and answer various ad hoc aggregate queries from a *PAT*. A systematic performance study to examine the effectiveness and efficiency of our design shows that the size of a *PAT* is small enough to be feasible in space for data streams and the cost of construction and maintenance for a *PAT* is smaller than the cost of materialization of the whole cube. This work extends data warehousing technology to applications with the properties of a data stream.

Pattern-based clustering is a practical data mining task with many applications. However, mining pattern-based clusters efficiently and effectively is still challenging. Since a pattern-based cluster consists of a subset of attributes/dimensions, the number of output pattern-based clusters may too huge to be understood well and there may exist redundancy between clusters. We propose the mining of maximal pattern-based clusters, which are non-redundant pattern-based clusters and develop two efficient and scalable algorithms, *MaPle* and *MaPle+*, for mining maximal pattern-based clusters in large databases. Test results on both real life data sets and synthetic data sets show that *MaPle+* clearly outperforms the best method previously pro-

posed. Recently, there are several interesting variations of pattern-based clustering, such as OP-clustering (Liu & Wang, 2003). As future work, it is interesting to use ideas in *MaPle* to develop efficient algorithms for mining such clusters.

As a future research, there are a few interesting topics as the extended researches of this thesis in data warehousing and data cube. One of the topics is the combination of data cubing techniques and classification for finding interesting decision factors from multidimensional data. As an motivating example, suppose we have a multidimensional data of real estate. A realtor wants to analyze her customers according to their attributes such as income level, age, ethnic groups, education, family size. She may have two types of queries: (1) comparison of the decision making factors from two groups of data, i.e., What is the difference of decision factors between a group of customers with annual income [200 k,100 k] and the whole group of customers in terms of purchasing a new house?, and (2) specific decision making pattern analysis, i.e., in which other groups hold a specific purchase pattern same as Asian customer group does? None of recent studies can be used to efficiently answer these queries and conduct the analysis. The general idea for this *associative classification cubes* is to maintain a classifier for every non-empty group of attribute value combination, i.e., a cell in a data cube, and store rules by materializing a data cube such that each cell stores only the class distribution, and present a set of fundamental operations for analysis based on associative classification cubes: Rule Extraction and Group Comparison.

Warehousing distributed databases and data cubing on other types of non-relational data are interesting extended researches of warehousing central and static databases. In practice, databases are often distributed in multiple data sources of the network environments and the formats of data are the non-relational types such as time-series data and XML. Since types and framework of data are different from those of previous warehousing and data cubing techniques, many sub problems are needed to be addressed, such as the network communication cost, data integration from multiple sources in the network, definition of aggregate functions on non-relational data, etc. These potential researches will help to broaden the applications of data warehouse and data cube.

References

Agarwal, R. C., Aggarwal, C. C., & Prasad, V. V. V. (2001). A tree projection algorithm for generation of frequent item sets. *Journal of Parallel and Distributed Computing*, *61*(3), 350–371.

Aggarwal, C. C., Wolf, J. L., Yu, P. S., Procopiuc, C., & Park, J. S. (1999, June). Fast algorithms for projected clustering. In *Sigmod '99: Proceedings of the 1999 acm sigmod international conference on management of data* (p. 61-72). Philadelphia, PA.

Aggarwal, C. C., & Yu, P. S. (2000, May). Finding generalized projected clusters in high dimensional spaces. In *Sigmod '00: Proceedings of the 2000 acm sigmod international conference on management of data* (p. 70-81). Dallas, TX.

Agrawal, R., Gehrke, J., Gunopulos, D., & Raghavan, P. (1998). Automatic subspace clustering of high dimensional data for data mining applications. In *Sigmod '98: Proceedings of the 1998 acm sigmod international conference on management of data* (pp. 94–105). New York, NY, USA: ACM Press.

Agrawal, R., Imielinski, T., & Swami, A. N. (1993). Mining association rules between sets of items in large databases. In P. Buneman & S. Jajodia (Eds.), *Sigmod '93: Proceedings of the 1993 acm sigmod international conference on management of data* (pp. 207–216). Washington, D.C.

Agrawal, R., & Srikant, R. (1994, September). Fast algorithms for mining association rules. In

Vldb '94: Proceedings of the 1994 vldb international conference on very large data bases (p. 487-499). Santiago, Chile.

Arasu, A., & Manku, G. S. (2004). Approximate counts and quantiles over sliding windows. In *Pods '04: Proceedings of the twenty-third acm sigmod-sigact-sigart symposium on principles of database systems* (pp. 286–296). New York, NY, USA: ACM Press.

Babcock, B., Babu, S., Datar, M., Motwani, R., & Widom, J. (2002). Models and issues in data stream systems. In *Pods '02: Proceedings of the twenty-first acm sigmod-sigact-sigart symposium on principles of database systems* (pp. 1–16). New York, NY, USA: ACM Press.

Babu, S., & Widom, J. (2001). Continuous queries over data streams. *SIGMOD Record*, *30*, 109-120.

Barbar, D., & Wu, X. (2000). Using loglinear models to compress datacube. In *Waim '00: Proceedings of the first international conference on web-age information management* (pp. 311–322). London, UK: Springer-Verlag.

Barbara, D., & Sullivan, M. (1997). Quasi-cubes: Exploiting approximation in multidimensional databases. *SIGMOD Record*, *26*, 12-17.

Beyer, K., Goldstein, J., Ramakrishnan, R., & Shaft, U. (1999, January). When is "nearest neighbor" meaningful? In C. Beeri & P. Buneman (Eds.), *Icdt '99: Proceedings of the 4th international international conference on database theory* (p. 217-235). Berlin, Germany.

Beyer, K., & Ramakrishnan, R. (1999, June). Bottom-up computation of sparse and iceberg cubes. In *Sigmod '99: Proceedings of the 1999 acm sigmod international conference on management of data* (p. 359-370).

Chang, J. H., & Lee, W. S. (2003, August). Finding recent frequent itemsets adaptively over online data streams. In *Kdd '03: Proceedings of the ninth acm sigkdd international conference on knowledge discovery and data mining* (p. 487-492). Washinton D.C.

Chaudhuri, S., & Dayal, U. (1997). An overview of data warehousing and olap technology. *SIGMOD Record*, *26*(1), 65–74.

Chen, Y., Dong, G., Han, J., Wah, B. W., & Wang, J. (2002, August). Multi-dimensional regression analysis of time-series data streams. In *Vldb '02: Proceedings of the 2002 vldb international conference on very large data bases* (pp. 323–334). Hongkong, China.

Cheng, C., Fu, A. W., & Zhang, Y. (1999). Entropy-based subspace clustering for mining numerical data. In *Kdd '99: Proceedings of the fifth acm sigkdd international conference on knowledge discovery and data mining* (pp. 84–93). New York, NY, USA: ACM Press.

Cheng, Y., & Church, G. M. (2000). Biclustering of expression data. In *Ismb '00: Proceedings of the 8th international conference on intelligent system for molecular biology* (p. 93-103).

Cohen, S., Nutt, W., & Serebrenik, A. (1999). Rewriting aggregate queries using views. In *Pods '99: Proceedings of the eighteenth acm sigmod-sigact-sigart symposium on principles of database systems* (p. 155-166). Philadelphia, Pennsylvania: ACM Press.

Cormode, G., Korn, F., Muthukrishnan, S., & Srivastava, D. (2003, September). Finding hierarchical heavy hitters in data streams. In *Vldb '03: Proceedings of the 2003 vldb international conference on very large data bases.* Berline, Germany.

Cormode, G., & Muthukrishnan, S. (2003). What's hot and what's not: tracking most frequent items dynamically. In *Pods '03: Proceedings of the twenty-second acm sigmod-sigact-sigart symposium on principles of database systems* (p. 296-306). New York, NY.

Datar, M., Gionis, A., Indyk, P., & Motwani, R. (2002, January). Maintaining stream statistics over sliding windows. In *Soda '02: Proceedings of 13th annual acm-siam symposium on discrete algorithms* (pp. 635–644).

Dobra, A., Garofalakis, M., Gehrke, J., & Rastogi, R. (2002). Processing complex aggregate queries over data streams. In *Sigmod '02: Proceedings of the 2002 acm sigmod international conference on management of data* (pp. 61–72). New York, NY, USA: ACM Press.

Fang, M., Shivakumar, N., Garcia-Molina, H., Motwani, R., & Ullman, J. D. (1998, 24–27). Computing iceberg queries efficiently. In *Vldb '98: Proceedings of the 1998 vldb international conference on very large data bases* (pp. 299–310).

Feng, Y., Agrawal, D., Abbadi, A. E., & Metwally, A. (2004). Range cube: Efficient cube computation by exploiting data correlation. In *Icde'04: Proceedings of the 2004 ieee international conference on data engineering* (p. 658-670).

Ganter, B., & Wille, R. (1996). *Formal concept analysis – mathematical foundations*. Springer.

Gehrke, J., Korn, F., & Srivastava, D. (2001). On computing correlated aggregates over continual data streams. In *Sigmod '01: Proceedings of the 2001 acm sigmod international conference on management of data* (pp. 13–24). New York, NY, USA: ACM Press.

Giannella, C., Han, J., Pei, J., & Yu, P. S. (2004). *Mining frequent patterns in data streams at multiple time granularities*. AAAI/MIT.

Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., et al. (1997). Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Journal of Data Mining and Knowledge Discovery*, *1*(1), 29–53.

Gupta, A., Mumick, I. S., & Subrahmanian, V. S. (1993). Maintaining views incrementally. In *Sigmod '93: Proceedings of the 1993 acm sigmod international conference on management of data* (pp. 157–166). New York, NY, USA: ACM Press.

Hahn. (1994). *Edited synoptic cloud reports from ships and land stations over the globe, 1892-1991*. (http://cdiac.ornl.gov/ftp/ndp026b/)

Han, J., Pei, J., Dong, G., & Wang, K. (2001, May). Efficient computation of iceberg cubes with complex measures. In *Sigmod '01: Proceedings of the 2001 acm sigmod international conference on management of data* (pp. 1–12). Santa Barbara, California.

Han, J., Pei, J., Yin, Y., & Mao, R. (2004). Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Journal of Data Mining and Knowledge Discovery*, *8*(1), 53–87.

Harinarayan, V., Rajaraman, A., & Ullman, J. D. (1996, June). Implementing data cubes efficiently. In *Sigmod '96: Proceedings of the 1996 acm sigmod international conference on management of data* (pp. 205–216). Montreal, Canada.

Inmon, W. H. (2002). *Building the data warehouse*. John Wiley Sons, Inc.

Jagadish, H. V., Madar, J., & Ng, R. (1999, September). Semantic compression and pattern extraction with fascicles. In *Vldb '99: Proceedings of the 1999 vldb international conference on very large data bases* (p. 186-198). Edinburgh, UK.

Jiang, D., Pei, J., Ramanathan, M., Tang, C., & Zhang, A. (2004). Mining coherent gene clusters from gene-sample-time microarray data. In *Kdd '04: Proceedings of the tenth acm sigkdd international conference on knowledge discovery and data mining* (pp. 430–439). New York, NY, USA: ACM Press.

Jiang, D., Pei, J., & Zhang, A. (2003). Dhc: A density-based hierarchical clustering method for time series gene expression data. In *Bibe '03: Proceedings of the 3rd ieee symposium on bioinformatics and bioengineering* (p. 393). Washington, DC, USA: IEEE Computer Society.

Johnson, T., & Shasha, D. (1997). Some approaches to index design for cube forests. *Bulletin of the Technical Committee on Data Engineering*, *20*(1), 27-35.

Karp, R. M., Papadimitrious, C. H., & Shanker, S. (2003, March). A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems*, *28*(1), 51–55.

Lakshmanan, L. V. S., Pei, J., & Zhao, Y. (2003). Qc-trees: an efficient summary structure for semantic olap. In *Sigmod '03: Proceedings of the 2003 acm sigmod international conference on management of data* (pp. 64–75). New York, NY, USA: ACM Press.

Lakshmanann, L. V. S., Pei, J., & Han, J. (2002, August). Quotient cube: How to summarize the semantics of a data cube. In *Vldb '02: Proceedings of the 2002 vldb international conference on very large data bases* (p. 778-789). Hong Kong, China.

Levy, A. Y., Mendelzon, A. O., Sagiv, Y., & Srivastava, D. (1995). Answering queries using views. In *Pods '95: Proceedings of the fourteenth acm sigmod-sigact-sigart symposium on principles of database systems* (pp. 95–104). San Jose, California.

Liu, J., & Wang, W. (2003, November). Op-cluster: Clustering by tendency in high dimensional space. In *Icdm'03: Proceedings of the 2003 ieee international conference on data mining* (pp. 187–194). Melbourne, Florida.

Mendelzon, A. O., & Vaisman, A. A. (2000). Temporal queries in olap. In *Vldb '00: Proceedings of the 26th international conference on very large data bases* (pp. 242–253). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Mumick, I. S., Quass, D., & Mumick, B. S. (1994, May). Maintenance of data cubes and summary tables in a warehouse. In *Sigmod '94: Proceedings of the 1994 acm sigmod international conference on management of data* (pp. 100–111). Tucson, Arizona.

Ng, R. T., Wagner, A., & Yin, Y. (2001). Iceberg-cube computation with pc clusters. In *Sigmod '01: Proceedings of the 2001 acm sigmod international conference on management of data* (pp. 25–36). New York, NY, USA: ACM Press.

Pasquier, N., Bastide, Y., Taouil, R., & Lakhal, L. (1999, January). Discovering frequent closed itemsets for association rules. In *Icdt '99: Proceedings of the 4th international international conference on database theory* (p. 398-416). Jerusalem, Israel.

Pei, J., Zhang, X., Cho, M., Wang, H., & Yu, P. S. (2003). Maple: A fast algorithm for maximal pattern-based clustering. In *Icdm '03: Proceedings of the third ieee international conference on data mining* (p. 259). Washington, DC, USA: IEEE Computer Society.

Quass, D., Gupta, A., Mumick, I. S., & Widom, J. (1996, December). Making views self-maintainable for data warehousing. In *Pdis '96:proceedings of the 4th international conference on parallel and distributed information systems* (pp. 158–169). Miami Beach, Florida.

Quass, D., & Widom, J. (1997). On-line warehouse view maintenance. In *Sigmod '97: Proceedings of the 1997 acm sigmod international conference on management of data* (pp. 393–404). New York, NY, USA: ACM Press.

Ross, K. A., & Srivastava, D. (1997, 25–27). Fast computation of sparse datacubes. In M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, & M. A. Jeusfeld (Eds.), *Vldb '97: Proceedings of the 1997 vldb international conference on very large data bases* (pp. 116–125). Morgan Kaufmann.

Ross, K. A., & Zaman, K. A. (2000). Optimizing selections over datacubes. In *Ssdbm '00: Proceedings of the 12th international conference on scientific and statistical database management* (p. 139). Washington, DC, USA: IEEE Computer Society.

Roussopoulos, N., Kotidis, Y., & Roussopoulos, M. (1997, May). Cubetree: Organization of and bulk updates on the data cube. In *Sigmod '97: Proceedings of the 1997 acm sigmod international conference on management of data* (pp. 89–99). Tucso, Arizona.

Sarawagi, S. (1997). Indexing OLAP data. *Data Engineering Bulletin, 20*(1), 36–43.

Shanmugasundaram, J., Fayyad, U., & Bradley, P. S. (1999). Compressed data cubes for olap aggregate query approximation on continuous dimensions. In *Kdd '99: Proceedings of the fifth acm sigkdd international conference on knowledge discovery and data mining* (pp. 223–232). New York, NY, USA: ACM Press.

Sismanis, Y., Deligiannakis, A., Roussopoulos, N., & Kotidis, Y. (2002). Dwarf: shrinking the petacube. In *Sigmod '02: Proceedings of the 2002 acm sigmod international conference on management of data* (pp. 464–475). New York, NY, USA: ACM Press.

Srivastava, D., Dar, S., Jagadish, H. V., & Levy, A. Y. (1996, September). Answering queries with aggregation using views. In *Vldb '02: Proceedings of the 2002 vldb international conference on very large data bases* (pp. 318–329). Bombay, India.

Tavazoie, S., Hughes, J., Campbell, M., Cho, R., & Church, G. (2000). *Yeast micro data set.* (http://arep.med.harvard.edu/biclustering/yeast.matrix)

Teng, W. G., Chen, M. S., & Yu, P. S. (2003, September). A regression-based temporal pattern mining scheme for data streams. In *Vldb '03: Proceedings of the 2003 vldb international conference on very large data bases* (pp. 93–104). Berlin, Germany.

TPC. (1998). *Tpc transaction processing performance council.* (http://www.tpc.org/tpch)

Vitter, J., Wang, M., & Iyer, B. (1998, November). Data cube approximation and histograms via wavelets. In *Cikm '98: Proceedings of the 7th international conference on information and knowledge management* (pp. 96–104). Washington D.C.

Wang, H., Wang, W., Yang, J., & Yu, P. S. (2002). Clustering by pattern similarity in large data sets. In *Sigmod '02: Proceedings of the 2002 acm sigmod international conference on management of data* (pp. 394–405). New York, NY, USA: ACM Press.

Wang, K., Jiang, Y., Yu, J. X., Dong, G., & Han, J. (2003). Pushing aggregate constraints by divide-and-approximate. In *Icde'03: Proceedings of the 2003 ieee international conference on data engineering* (p. 291- 302).

Wang, W., Lu, H., Feng, J., & Yu, J. X. (2002). Condensed cube: An effective approach to reducing data cube size. In *Icde'02: Proceedings of the 2002 ieee international conference on data engineering* (p. 155-165).

Widom, J. (1995). Research problems in data warehousing. In *Cikm '95: Proceedings of the 4th international conference on information and knowledge management* (pp. 25–30). New York, NY, USA: ACM Press.

Xin, D., Han, J., Li, X., & Wah, B. W. (2003). Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In *Vldb '03: Proceedings of the 2003 vldb international conference on very large data bases.*

Yang, G. (2004). The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *Kdd '04: Proceedings of the tenth acm sigkdd international conference on knowledge discovery and data mining* (pp. 344–353). New York, NY, USA: ACM Press.

Yang, J., Wang, W., Wang, H., & Yu, P. S. (2002, April). δ-cluster: Capturing subspace correlation in a large data set. In *Icde'02: Proceedings of the 2002 ieee international conference on data engineering* (pp. 517–528). San Fransisco, CA.

Yu, J. X., Chong, X., Lu, H., & Zhou, A. (2004, August). False positive or false negative: Mining frequent itemsets from high speed transactional data streams. In *Vldb '04: Proceedings of the 2004 vldb international conference on very large data bases* (pp. 204–215). Toronto, ON, Canada.

Zaki, M. J., Parthasarathy, S., Ogihara, M., & Li, W. (1997, August). New algorithms for fast discovery of association rules. In *Kdd '97: Proceedings of the third acm sigkdd international conference on knowledge discovery and data mining* (p. 283-286). Newport Beach, CA.

Zhao, L., & Zaki, M. J. (2005). Tricluster: an effective algorithm for mining coherent clusters in 3d microarray data. In *Sigmod '05: Proceedings of the 2005 acm sigmod international conference on management of data* (pp. 694–705). New York, NY, USA: ACM Press.

Zhao, Y., Deshpande, P. M., & Naughton, J. F. (1997). An array-based algorithm for simultaneous multidimensional aggregates. In *Sigmod '97: Proceedings of the 1997 acm sigmod international conference on management of data* (pp. 159–170). New York, NY, USA: ACM Press.