2011

Mining Multiple Related Tables Using Object-Oriented Model

Dan Zhang
University of Windsor

Mining Multiple Related Tables Using Object-Oriented Model

By

Dan Zhang

A Thesis
Submitted to the Faculty of Graduate Studies through the School of Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2011

Your file Votre référence
ISBN: 978-0-494-81744-5
Our file Notre référence
ISBN: 978-0-494-81744-5

DECLARATION OF ORIGINALITY

ABSTRACT

An object-oriented database is represented by a set of classes connected by their class inheritance hierarchy through superclass and subclass relationships. An object-oriented database is suitable for capturing more details and complexity for real world data. Existing algorithms for mining multiple databases are either Apriori-based or machine learning techniques, but are not suitable for mining multiple object-oriented databases.

This thesis proposes an object-oriented class model and database schema, and a series of class methods including that for object-oriented join (*OOJoin*) which joins superclass and subclass tables by matching their type and super type relationships, mining Hierarchical Frequent Patterns (*MineHFPs*) from multiple integrated databases by applying an extended TidFP technique which specifies the class hierarchy by traversing the multiple database inheritance hierarchy. This thesis also extends *map-gen join* method used in TidFP algorithm to *oomap-gen join* for generating k-itemset candidate pattern to reduce the candidate itemset generation by indexing the (k-1)-itemset candidate pattern using two position codes of start position and end position codes tied to inheritance hierarchy level. Experiments show that the proposed MineHFPs algorithm for mining hierarchical frequent patterns is more effective and efficient for complex queries.

Keywords: Multiple database mining, object-oriented model, inheritance hierarchies, Hierarchical Frequent Patterns

ACKNOWLEDGEMENT

I would like to give my sincere appreciation to all of the people who have helped me throughout my education. I express my heartfelt gratitude to my father and mother for their support throughout my graduate studies.

I am very grateful to my supervisor, Dr. Christie Ezeife for her continuous support throughout my graduate study and also for numerous awards of research assistantship. She always guided me and encouraged me throughout the process of this research work, taking time to read all my thesis updates.

I would also like to thank my external reader, Dr. Mehdi S. Monfared, my internal reader, Dr. Ziad Kobti, and my thesis committee chair, Dr. Robert Kent for making time to be in my thesis committee, reading the thesis and providing valuable input. I appreciate all your valuable suggestions and time, which have helped improve the quality of this thesis.

At last, I would express my appreciations to all my friends and colleagues in Dr. Ezeife's WODD lab and school of Computer Science. Thank you all!

TABLE OF CONTENT

TABLE OF CONTENT

LIST OF FIGURES

LIST OF FIGURES

LIST OF TABLES

LIST OF TABLES

1. INTRODUCTION

Data mining is a process of extracting relevant and important knowledge from large data to facilitate decision making. Data mining has attracted tremendous amount of attention in database research because of its applicability in many areas, including decision support, market strategy and financial forecast. The process of discovering interesting information from huge set of data often employs different techniques and approaches. Some of the approaches include: Classification, Clustering, Regression, and Association rule mining. Agrawal and Srikant (1995) proposed the Apriori algorithm that mines frequent patterns from a single database table by generating candidate itemsets and scanning database to count the support of each candidate. The later research work such as FP-tree proposed by Han and *et al.* (2004) also mines a single database table by projecting the database into a compressed tree and mining frequent pattern by pattern-growth technique. However, mining single database table is not enough to extract useful information or answer more complex queries. Ezeife and Zhang (2009) proposed the Apriori-based TidFP algorithm that not only improves the efficiency of mining a single database table by intersecting transaction ids to avoid multiple scanning of the database, but also mines multiple tables by a set of operations such as intersect, minus, or union. The researchers have begun to work on mining multiple data sources about a decade ago. The early research work of mining multiple data sources was applying hardware techniques (e.g., Agrawal and Shafer (1996)), such as parallel computer or distributed system, to separate data sources. Then they utilize multiple computing resources to mine the data. The later research work on mining multiple data sources such as Sequential Pattern Mining in Multi-Databases via Multiple Alignment (Kum and *et al.* (2006))

applies machine learning techniques, such as clustering to discover both local patterns and global patterns of each data source. Mining multi-level association rule (Han and Fu (1995)) generalizes or specializes patterns to higher or lower concept hierarchy (class hierarchy) to discover the frequent pattern in different hierarchy levels. There are also research work that proposed frequent pattern mining methods for object-oriented databases. Kuba and Popelinsky (2003) proposed the algorithm OR-FP that mines frequent patterns that covers at least a given minimum number of objects in object-oriented data. However, these algorithms are not designed for object-oriented multiple database mining.

1.1 Object-oriented data model

Real world data are complex and good to be presented as objects (Dzeroski and Lavrac (2001)). An object-oriented database is suitable for capturing more comprehensive and detailed complexity of real world data, such as different products on a Business to Customer (B2C) website, their histories, versions, price, images, or specifications. Data captured from B2C websites are not suitable to store in a relational database, because of the scalability of the relational database is limited. For example, changing the schema of a relational database is a complicated process. Products sold on B2C websites are being changed frequently. There are always new products coming. New products have their new features and specification. For example, several years ago, B2C websites such as Bestbuy^{TM} or Futureshop^{TM}, were not selling one type of computer called “Pad”. The new product “Pad” is an extension of “laptop”, but has its new features or specifications, such as “3G device” or “Touch screen”. However, it still has common feature as laptops, such as “CPU”, “RAM”, “LCD screen size”. If we want to store the information of “Pad” in a relational database, we need to modify the database schema of the table for Laptop, such as adding more columns into the table for Laptop. Alternatively, we can create a new database table for “Pad” which stores all features and specifications of “Pad”. However, this solution will be hard to show the relationship between “Laptop” and “Pad”. In addition, mining queries of the database also need to be modified due to the changes of the database schema. Therefore, an object-oriented database schema is more suitable for the complexity of Web data. All products are grouped into classes and show the inheritance hierarchy relationships. For example, the new products “Pad” has its own features and specifications, but it does have some common features or specifications as “Laptop”. In an object-oriented database, a new class called “Pad” can be added as a subclass of “Laptop” class. The new class “Pad” inherits all features and specifications of “Laptop”, but has its own features and specifications as well. Also, the new class “Pad” will have its own method to deal with mining queries.

An object-oriented data model is a logical organization of real world objects (entities), for example, an object-oriented data model represents classes of computers (computer, laptop, or desktop) about which a company wishes to hold information. There exist constraints (for example, properties of a computer object, CUP, RAM, and Hard_drive must be valid to satisfy a computer object) and relationships (for example, there exists inheritance hierarchical relationship between object “computer” and object “laptop”) among the objects. An object-oriented database system is the database system that implements the object-oriented data model. The object-oriented data model consists of the following core concepts:

1) The object and object identifier

Real world entity such as a "computer" is represented as an object. In object-oriented database, every instantiated object is associated with a unique object-oriented identifier, also called *OID*. The object-oriented database system requires every instantiated object to get a unique identifier at the object instantiation time and it has to be guaranteed that this identifier remains unique throughout the lifetime of the object. Therefore, this object-identifier is also called logical object identifier.

2) Attributes and methods

Attributes of an object, such as specifications of a computer object (for example, CPU, RAM, or Hard_drive specifications), have a set of values assigned to them, are the properties that an object has. The values of attributes represent the state of the object. The methods of an object (for example, a method of a computer object is getCPU) ar e behaviors of the object. These behaviors operate on the state (for example, a state a computer object having CPU = "2GHz", RAM = "2GB", and Hard_drive = "250GB") of the object. For example, for Computer class, there is a computer object and its CPU type is known. There is a class method, called getCPU() that returns this CPU type.

3) Class:

Class is an abstraction of all the instantiated objects (instances) which share the same set of attributes and methods. An instantiated object must belong to only one class as an instance of that class. A class may also be primitive (no attributes), e.g., integer, string, Boolean. For example,

```
Computer{

  CPU: String
  RAM: String
  Hard_drive: String

  private set MineComputerFPs(float min_supp)
 private int GetCountCPU(String cpu_type)

}
```

In the above example, the class Computer has the attributes: CPU, RAM, and Hard_drive that are all string type. The class computer also has class methods MineComputerFPs(float min_supp) and GetCountCPU(String cpu_type). The private method MineComputerFPs(float min_supp) takes a percentage number for minimum support as a parameter and mines the attributes of a computer class as frequent patterns. For example, mine all frequent attributes of Computer class having minimum support of 50%. The private method GetCountCPU(String cpu_type) takes CPU type as a parameter and return the number of CPU attributes of Computer class according to the specified CPU types. For example, get the number of "2GHz" CPUs in the computer class.

4) Class Hierarchy and Inheritance

Class hierarchy in Computer Science is the classification of objects class type, which is a set of classes and their inter relationships. Entities and objects with similar characteristics are grouped together and described as a class type. Similar characteristics means that the objects have the same structure and behaviours, or the same attributes and methods. If objects slightly differ in their structure or behaviours, they should belong to different class types. The relationship between the classes that host slightly different objects is called class inheritance hierarchy. Class inheritance hierarchy shows a relationship between all classes in the object database, which derives a new child class (subclass) from an existing parent class (superclass). The subclass inherits all attributes and class methods from the superclass, but also has its own attributes and methods. Subclass can inherit from either one superclass or multiple superclasses. The general idea behind inheritance is the **is-a** relationship between object types, as shown in Figure 1.1.

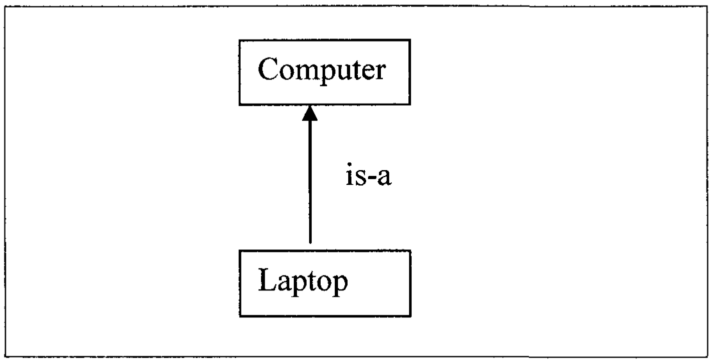

Figure 1.1 is-a relationship

In Figure 1.1, *Computer* is a super type (superclass) and *Laptop* is a sub type (subclass). Super type and sub type relationship represents the concept of generalization and specification respectively. *Computer* is a generalization of *Laptop* and *Laptop* is a specialization of *Computer*.

1.2 Object-oriented database schema

"An *object-oriented database* is a database management system in which information is represented in the form of objects" (Wikepedia (2011)). The Object-oriented database model has several advantages over a relational database model. The First Normal form (1NF) of the relational database model stipulates that there must be atomic values for attributes in a tuple. It does not allow complex values, such as sets, lists, or other data structures. However, the attributes of an object-oriented database model can be a complex collection of types, such as, sets, lists, or some other data structure. The object-oriented database model does not need additional tables to store the data represented in a collection type. In a relational database model, procedures must be maintained outside of the relational model itself. However, in an object-oriented database model, these procedures can be considered as behaviors of the objects and can be maintained as methods of the classes (Wikepedia (2011)). Some object-oriented databases are designed to work with object-oriented programming languages such as Delphi, Ruby, Python, Perl, Java, C#/Visual Basic.NET, C++, Objective-C and Smalltalk. The research/commercial products of the object-oriented databases include EXODUS, Vodak, Gemstone, Gbase, Objectivity/DB, ObjectStore, and etc. Figure 1.2 provides an example of an object-oriented database model.

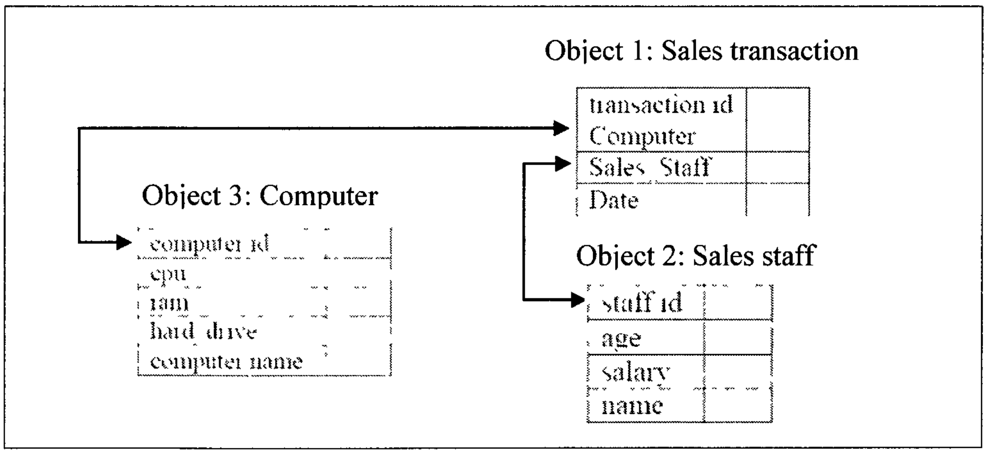

Figure 1.2 an example of object-oriented database model

In Figure 1.2, there are three objects, Sales transaction, Computer, and Sales staff. The computer object and the sales staff object are attributes of the sales transaction object.

"An *object-relational database* is a database management system (DBMS) similar to a relational database, but with an object-oriented database model: objects, classes and inheritance are directly supported in database schemas and in the query language." Compared with relational database model, the object-relational database supports complex data types for the attributes of every tuple, type inheritance, and object behaviors such as methods of the classes. The products of object-relational database management systems include Illustra, PostgreSQL, Omniscience, and etc. The database management systems such as IBM's DB2, Oracle database, and Microsoft SQL Server, make claims to support the *object-relational* technology (Wiki (2011)).

An object-oriented database is represented by a set of classes connected by their class inheritance hierarchy through superclass and subclass relationships (Kemper and Moerkotte (1994)). Other hierarchies like object link or part-of hierarchies may also exist. An object-oriented database consists of a set of classes, C_i, with a class inheritance hierarchy ***H***. Each class is defined as an ordered relation $\boldsymbol{C_i = (K, T, S, A, M, O)}$, where K is the class identifier, T is the class type, S is the super type (superclass) of the class, A is a set of attributes (Ezeife and Barker (1998)). M is a set of methods, O is a set of objects. Class inheritance hierarchy H is used to depict superclass and subclass relationship between classes in an object-oriented database and can be represented as a set of pairs of class and superclass in the form of (class, superclass).

Example 1.1 (An object database class inheritance hierarchy): A computer object-oriented database consists of three classes (computer, laptop, desktop), which are related through class inheritance hierarchy H={(Laptop, Computer), (Desktop, Computer)}. Database schema for this database is provided as:

C1 = (K1, Type, Super, {oid, CPU, RAM, Hard_drive, computer_name }, $o_1, o_2, \ldots, o_n$);

C2 = (K2, Type, Super, {oid, Screen_size, Battery_life}, $o_1, o_2, \ldots, o_n$);

C3 = (K3, Type, Super, {oid, Graphic}, $o_1, o_2, \ldots, o_n$);

Provide the class inheritance hierarchy for this computer object-oriented database.

Solution 1.1: The class inheritance hierarchy of the computer object-oriented database described above is shown in Figure 1.3.

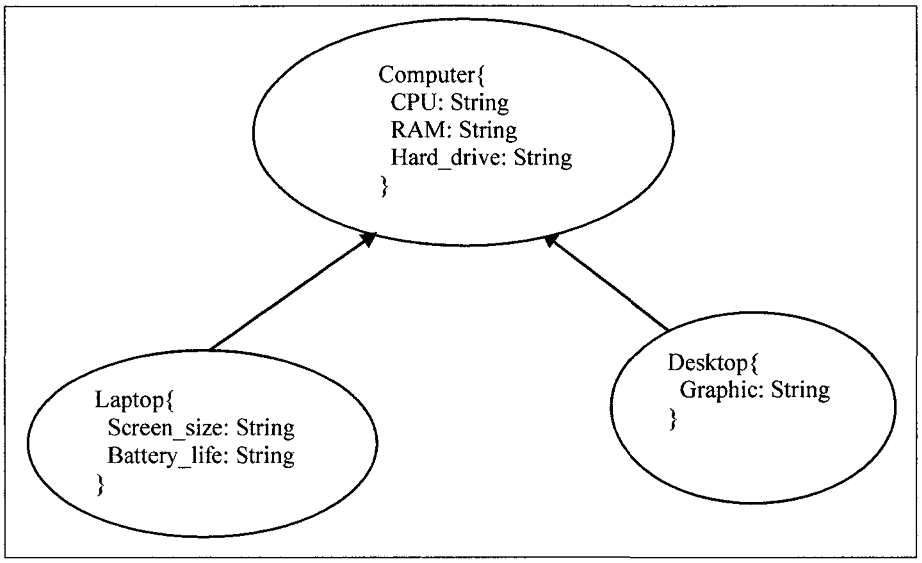

Figure 1.3 Example of class inheritance hierarchies for Computer object database

In Figure 1.3, class *Computer* is the super class of *Laptop* class and *Desktop* class. Computer class has attributes CPU, RAM and Hard_drive. Class *Laptop* and *Desktop* are sublcasses of class *Computer* that inherit the class *Computer* and have inherited all attributes from class *Computer*, but have their own attributes Screen size, Battery_life, and Graphic.

Definition 1 (tree structure of multiple database class inheritance hierarchy): ***MHTree*** is the tree structure representation of multiple databases inheritance hierarchy. For example, three object-oriented computer databases for IBM, Dell, and HP are shown in Figure 1.3.

Example 1.2 (A multiple database class inheritance hierarchy): Inheritance hierarchy in multiple databases, *MH* represents three computer object databases such as IBM, Dell, and HP. Every database consists of three classes (computer, laptop, desktop). MH is represented in a set of pairs used to depict supperclass and subclass relationship. MH={(IBM, Root), (Dell, Root), (HP, Root), (Computer, IBM), (Computer, Dell), (Computer, HP), (Laptop, Computer), (Desktop, Computer)}. Provide multiple database inheritance hierarchy in a tree structure.

Solution 1.2: The multiple database inheritance hierarchy for three object-oriented Computer databases having "Computer", "Laptop", and "Desktop" inheritance hierarchy relationships for IBM, Dell, and HP can be represented as a tree structure in Figure 1.4.

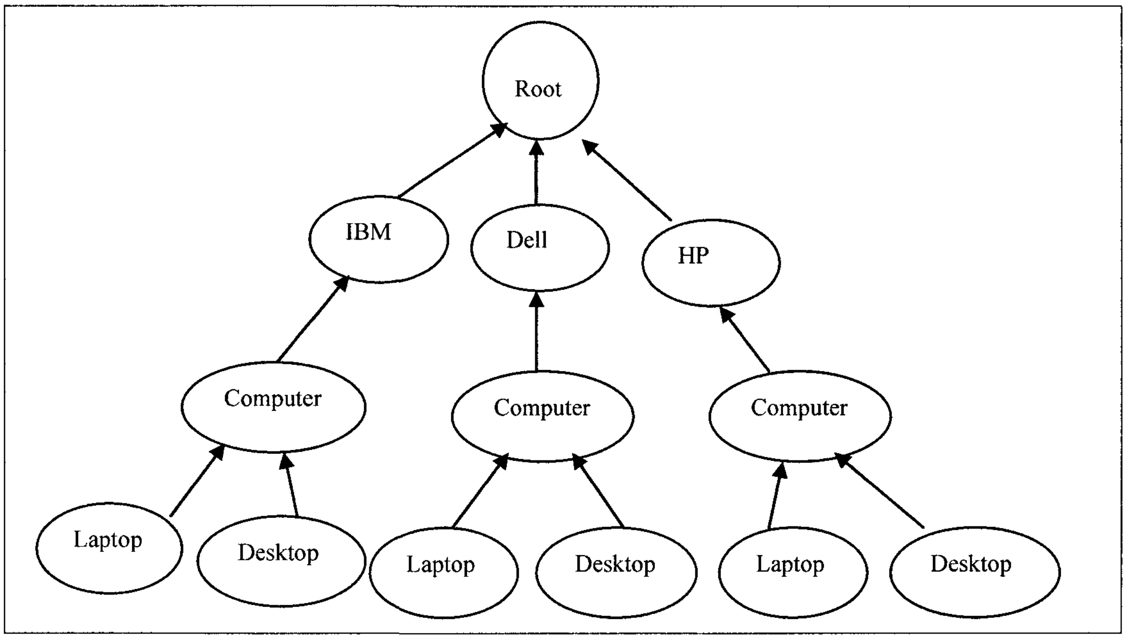

Figure 1.4 Multiple databases inheritance hierarchy Tree (MHTree)

In Figure 1.4, the nodes "IBM", "Dell", and "HP" represent three companies that make computers and also they represent three databases. Every database has Computer class table and inherited by two other class tables "Laptop" and "Desktop". Computer, Laptop, and Desktop class tables store the specification of all computers, such as CPU, RAM, Hard_drive, Screen_size, or Graphic. The Computer database table schema is shown in Figure 1.3. The object database consists of all classes with the ***Root***.

Definition 2 (database schema of *Root* table)***: Root(K, T, S, A, M, O).*** *Root* class table is a transactional table which records the transactions, for example, sales transactions from different object databases. *K* is the object id which stands for the transaction id. *T* is the class type of the transaction (name of the database where the transaction comes from). *S* is the super type of the transaction. *A* is a set of attributes which includes a set of super_type *S* of C_i, super_i (number of super_i depends on levels of hierarchy of C_i) and all attributes *A* of C_i. *M* is a set of class methods which are behaviors of *Root* class, such as updating *Root* table and mining patterns in *Root* table. *O* is a set of objects of transaction (one object stands for one transaction). For example, a sale transaction of a purchased desktop from IBM database recorded in Root table (sales transaction table) has object id *K* which is a transaction id (an integer number), class type *T* which indicates the database where the transaction comes from (database is "IBM" in this case), super_type *S* which is "Root", the attributes *A* include two super type, super_1 and super_2 (in computer object database, there are two levels of hierarchy) and all attributes of C_i, CPU, RAM, Hard_drive, Screen_size, battery_life and Graphic.

In an object-oriented database model, the instantiated objects (instance) are referenced (retrieved) by following the object pointers, so do the inheritance relationships. Therefore, it will be a problem to retrieve the information of inheritance hierarchy in an object-oriented database model. Inheritance hierarchy can be represented by foreign keys in a relational model. The relational database model provides a clear vision of all attributes of a table including the information of inheritance hierarchy. However, converting the object-oriented database model into a relational model is also a challenge, such as the *join* operation. In an object-oriented database model, there is no specific *join* operation, because the instantiated objects are referenced by the object pointers. We will provide the solution to address the problem of the *join* operation in section 3.2.2.

We can also define the object-oriented database as a relational database represented with a set of tables (relations) connected through foreign key relationships where the foreign keys in our object-oriented database model of $C_i = (K, T, S, A, M, O)$ and *Root*(K, T, S, A, M, O) are realizing the inheritance hierarchy with the subclass and supperclass relationships as defined in the T, S attributes of an object class.

In this thesis, we are dealing with data from different databases and or data captured from different B2C websites. Different databases and websites may give different names to the same product. For example, a laptop computer may be given the name of "Laptop" in one database or website and "Notebook computer" in a different database or website. Therefore, data from different database or websites must be cleaned. For example, we should change all such names to be consistent across the databases. As an example the name "Notebook computer" should be changed to the name "Laptop". This thesis is not concerned with such as cleaning problems and assumes that data had been previous cleaned such that all schemas from the different databases of the same domain are the same.

Example 1.3 (object database represented with relational model): Provide relational database and tables that represent the computer object database in Figure 1.2 and the *Root* class table defined in ***Definition 2***.

Solution 1.3:

Computer (comp_id: string, type: string, super_type: string, cpu: string, ram: string, hard_drive: string);

Laptop (lap_id: string, type: string, super_type: string, screen_size: string, battery_life: string);

Desktop (desk_id: string, type: string, super_type: string, graphic: string);

Root (transaction_id: integer, type: string, super_type: string, $super_1$: string, $super_2$: string, cpu: string, ram: string, hard_drive: string, screen_size: string, battery_life: string, graphic: string)

In the above relational database schema, *comp_id* is the primary key of the computer table. *lap_id* is the primary key of laptop table, *desk_id* is the primary key of the desktop table, and *transaction_id* is the primary key of Root table. All class tables have the composite foreign keys consisting of the two attributes type and super_type in each table. A sample computer object database owned by IBM company is shown in Table 1.1, Table 1.2, and Table 1.3.

Comp_id	Type	Super_type	CPU	RAM	Hard Drive	Computer_name
comp1	Laptop	Computer	2GHz	2GB	250GB	Ideapad laptop
comp2	Laptop	Computer	2GHz	2GB	320GB	Ideapad laptop
comp3	Laptop	Computer	3GHz	4GB	350GB	Thinkpad laptop
comp4	Desktop	Computer	3GHz	4GB	500GB	IBM workstation
comp5	Desktop	Computer	3GHz	4GB	500GB	IBM workstation
comp6	Desktop	Computer	3GHz	4GB	500GB	IBM PC

Table 1.1 Object Table of *Computer* Class in IBM DB

Lap_id	Type	Super_type	Screen_size	Battery_life
lapt1	Ideapad Laptop	Laptop	15"	3 hours
lapt2	Ideapad Laptop	Laptop	15"	3 hours
lapt3	Thinkpad Laptop	Laptop	17"	3.5 hours

Table 1.2 Objects table of *Laptop* class in IBM DB

Desk_id	Type	Super_type	Graphic
desk1	Work station	Desktop	256M
desk2	Work station	Desktop	256M
desk3	Desktop	Desktop	512M

Table 1.3 Objects table of *Desktop* class in IBM DB

Table 1.1 is the Computer class table that stores the specifications of computers. Table 1.2 and Table 1.3 stores the specifications of laptop and desktop that inherit the computer class. An example of Root class table that records all computers purchased from different databases, such as IBM, Dell, or HP is shown in Table 1.4

Oid (Tid)	Type	Super Type	$super_1$	$super_2$	CPU	RAM	Hard Drive	Computer Name	Screen Size	Battery Life	Graphic
1	IBM	Root	computer	laptop	2GHz	2GB	250GB	Ideapad laptop	15"	3hrs	
2	IBM	Root	computer	laptop	2GHz	4GB	320GB	Ideapad laptop	15"	3hrs	
3	Dell	Root	computer	laptop	2GHz	2GB	350GB	Thinkpad laptop	17"	3.5hrs	
4	HP	Root	computer	desktop	3GHz	4GB	500GB	Media centre			256M
5	HP	Root	computer	desktop	3GHz	4GB	500GB	Media centre			256M
6	Dell	Root	computer	desktop	3GHz	4GB	500GB	precision			512M
7	IBM	Root	computer	laptop	2GHz	2GB	320GB	Thinkpad Laptop	15"	3hrs	
8	HP	Root	computer	laptop	3GHz	4GB	350GB	Media centre	17"	3.5hrs	

Table 1.4 example of *Root* class table integrated from tables of three databases IBM, HP, and Dell

Table 1.4 is a sales transactions table. There are 8 *Root* class objects. Every object indicates one transaction of computer purchased. In the schema of *Root*(*K, T, S, A, M, O*), *Oid* is the object id, *K*, which indicates the transaction id represented by integer number. *Type* is the class type *T* which indicates the database where the computer comes from, such as "IBM", "Dell", or "HP". *Super_type* is *S*. A set of attributes of class *Root, A* include:

1) $super_1$ and $super_2$ are super types of the computer object database. In this example, class *Computer* has subclasses *Laptop* and *Desktop*. There are 2 levels of hierarchy, so there are 2 "super" attributes $super_1$and $super_2$. If there are n levels of hierarchy, there will be $super_1$, $super_2$... and $super_n$

2) CPU, RAM, Hard_driver, Screen_size, Battery_life, Graphic are all attributes of Computer, Laptop, and Desktop classes.

The Root class also has a set of class methods *M*:

private InsertTransactions();

private MineRootFPs(float min_supp);

The private method *InsertTransactions*() is used to insert computer purchase transactions into the *Root* table. Such a transaction record contains the model and specifications of the computer, and the company that made the computer. When the method is called, the new purchase transaction will be inserted into the *Root* table. The private method *MineRootFPs(float min_supp)* takes a percentage number representing the minimum support as parameter and mines the attributes of a *Root* class as frequent patterns. For example, this method can answer queries such as: *What are the most popular hardware component specifications (CPU, RAM, Hard_drive, screen size, battery life, and Graphics card) among the computer systems that have been sold (with a minimum support of 50%)?*

In the computer object database, there are three classes. *Computer*, *Laptop*, and *Desktop* are shown in Figure 1.2. When a purchase is made, it specifies the purchase as laptop or desktop. Therefore, the computer specifications need to be extracted from two object tables, *Computer* and *Laptop*, or *Computer* and *Desktop.* A method, called *object-oriented join* (*OOJoin*) needs to be defined to join two object tables which have inheritance relationship.

Definition 3 (join superclass and subclass tables): Object-oriented Join, *OOJoin*, is used to join tuples in two object class tables, superclass table, C_{super} and subclass table, C_{sub}. Every tuple in the object class table has the primary key *K*, foreign keys type *T* and super type *S*, and a set of other attributes A. *OOJoin* select a set of tuples based on distinct $C_{super}.K$ and $C_{sub}.K$ from the result of $C_{super} \bowtie C_{sub}$ where($C_{super}.T = C_{sub}.T$ or $C_{super}.T = C_{sub}.S$). For example, *OOJoin* algorithm joins the *Computer* class table and *Laptop* class table and the resulting join will result in a table that contains all Laptop attributes with the expanded attributes of the superclass Computer.

Example 1.5 (Object-oriented Join): Show the result of *object-oriented join* of two object tables, the *Computer* class table (Table 1.1) and the *Laptop* class table (Table 1.2). The resulting table should contain all the attributes of the *Laptop* class and its superclass *Computer.*

Solution 1.5: Table 1.5 is the result of *object-oriented join* of Table 1, the *Computer* class table and Table 1.2, the *Laptop* class table.

ID	Type	Super	CPU	RAM	Hard Drive	Comp Name	ID	Type	Super	Screen Size	Battery Life
comp1	Laptop	Computer	2GHz	2GB	250GB	I laptop	lapt1	Ideapad Laptop	Laptop	15"	3hrs
comp2	Laptop	Computer	2GHz	2GB	320GB	I laptop	lapt2	Ideapad Laptop	Laptop	15"	3hrs
comp3	Laptop	Computer	3GHz	4GB	350GB	T laptop	lapt3	Thinkpad Laptop	Laptop	17"	3 5hrs

Table 1.5 result of *object-oriented join* Table 1.1 and Table 1.2

Details of the Object-oriented join algorithm will be discussed in section 3.2.

1.3 Association rule Mining

Frequent patterns are itemsets that appear in a data set with frequency (also called support) no less than a user-specified threshold (also called minimum support). For example, a set of items, such as milk, juice, and bread that appear frequently together in a transaction dataset, forms a frequent itemset. Frequ ent pattern mining is the task of discovering the frequent patterns from the transactional databases. Frequent pattern mining is the essential step of association rule mining. Association Rule is an implication of the form $X \Rightarrow Y_i$, where X is a set of some items in the set of all items Y, and Y_i is a single item in Y that is not present in X. In above example, frequently occurred items milk, juice, and bread may lead to find association rules milk & juice => bread, which means that the customers who purchase milk and juice may usually purchase bread.

1.4 Frequent pattern Mining in Object-oriented Model

Frequent pattern mining in a single relational database table is used to find the itemsets whose frequencies are no less than a user-specified threshold (also called minimum support). The frequency of an itemset is the occurrence over all transactions. One transaction is one row in the database table. Therefore, frequent patterns in traditional database system are just items or combination of items (itemsets).

In an object table, every instantiated object can be considered as one row in a relational database table. The attributes of the object can be considered as itemsets (pattern). Mining the frequent pattern in an object table is used to discover an object attribute or combination of object attributes that appear frequently in all objects. In Example 1.3, Table 1.1, a Computer class table has attributes "CPU", "RAM", "Hard_drive". The objects in Table 1.1 have attributes, such as <2GHz>, <3GHz>, <2GB>, <4GB>, or <500GB>. These attributes can be considered as itemsets.

Based on Example 1.3 (sample of Computer database), some frequent pattern mining queries can be answered:

Query 1: What are the most frequent hardware components (CPU, RAM, hard drive) used by IBM in their computer model lineup? (with a minimum support of 50%)

Query 1 can be answered by calling the method *MineComputerFPs(50%)* of the class *Computer*. The method can apply one of the frequent pattern mining algorithms, such as the TidFP algorithm to Table 1.1.

Query 2: What are the most frequent hardware components (CPU, RAM, Screen size) used by IBM in their laptop model lineup? (with a minimum support of 50%)

Query 2 cannot be answered by applying the TidFP algorithm on table 1.1. It involves two tables, table1.1 and table 1.2. An ***Object-oriented Join*** needs to be applied on Table

1.1 and Table 1.2. Following the join we need to apply a frequent pattern mining algorithm on the joined table.

If we want to determine (mine) the most popular hardware components or hardware component combinations among the computer systems that have been sold, we need to mine the sales transaction table (shown in Example 1.5). Assuming that we want to answer the query such as:

Query 3: What are the most popular hardware component specifications (CPU, RAM, Hard_drive, screen size, battery life, and Graphics card) among the computer systems that have been sold (with a minimum support of 50%)?

If we apply the TidFP algorithm on Table 1.4, it returns the patterns in the following format, <Tidlist, itemset>. Each pattern consists of a transaction id list and an itemset. For example: <1,2,3,7, 2GHz>, <4,5,6,8, 3GHz>, <1,3,7, 2GB>, <2,4,5,6,8, 4GB>, <1,3,7, 2GHz,2GB>, and <4,5,6,8, 3GHz,4GB>. Unfortunately, query 3 is not good enough to discover patterns at different hierarchies for an informative database table such as Table 1.4. For example in Table 1.4, 2GHz processors are not considered to be of a significant enough of a frequency when both desktop and laptop computers are considered, however, 2GHz processors are of a significant frequency when only laptop computers are considered. This table integrates information of hierarchy from multiple class tables from different databases.

Therefore, we need an algorithm which can handle queries that not only mine the frequent patterns in a table, but which also specify at what hierarchy level the pattern is frequent. Examples of such queries are query 4 and query 5.

Query 4: What are the most popular hardware component specifications (CPU, RAM, Hard_drive, screen size, battery life, and Graphics card) among the computer systems that have been sold by a ***particular company*** *like Dell (with a minimum support of 50%)?*

Query 5: What are the most popular hardware component specifications (CPU, RAM, Hard_drive, screen size, and battery life) among a computer system ***subgroup*** *such as laptops and sold by a* ***particular company*** *like Dell (with a minimum support of 50%)?*

The algorithm for mining multi-level association rules (Han and Fu (1995)) can discover a frequent pattern(s) at different concept hierarchy levels. In this algorithm, a pattern can be generalized or specialized by a roll-up or a drill-down along the concept hierarchy (class hierarchy). Then the pattern(s) will be replaced by another pattern from a higher or a lower concept hierarchy level. For example, two patterns <2%milk> and <1%milk> can be replaced by the pattern <milk> from a higher concept hierarchy (class hierarchy) level or <Dairyland 1%milk> from a lower concept hierarchy (class hierarchy) level, as shown in Figure 1.5.

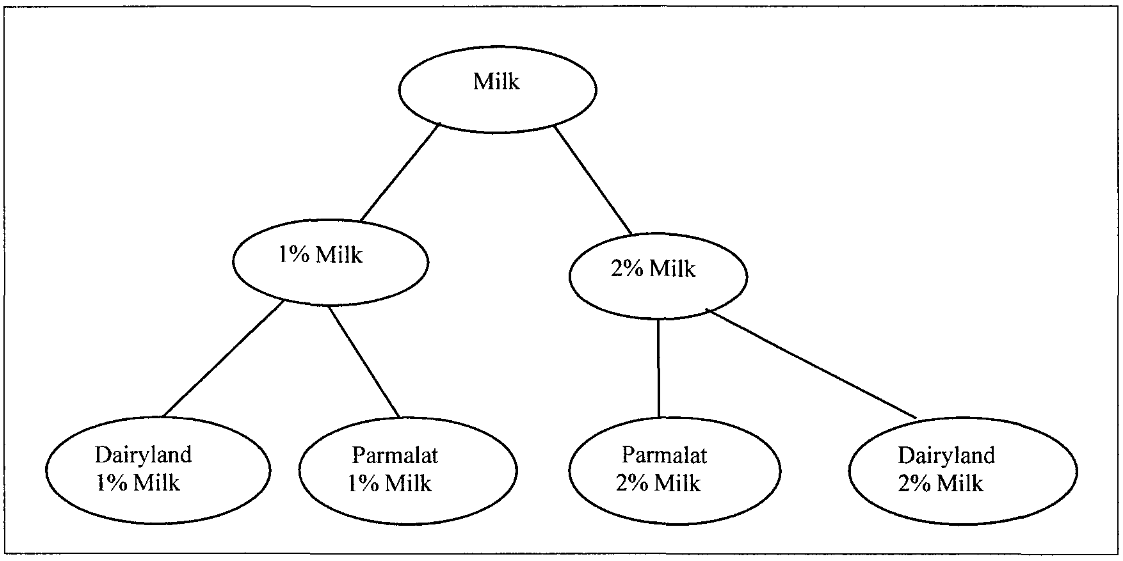

Figure 1.5 Concept hierarchy (class hierarchy) of Milk product

However, the algorithm for mining multi-level association rules (Han and Fu (1995)) does not consider an instantiated class object as a pattern and does not consider the attributes of such instantiated objects as pattern(s). For example, the milk, 1%milk, and Dairyland1%milk could be classes and have their attributes, such as Milk:{*protein*, *calories*}. 1% Milk:{ *protein*, *calories*, *fat*}. Dairyland1%milk :{ *protein*, *calories*, *fat*, *brand*}. The attributes such as *protein*, *calories*, *fat*, or *brand* could be mined for frequent patterns. The OR-FP algorithm (Kuba and Popelinsky (2003)) mines instantiated objects and the attributes of such objects for frequent patterns using the query/mining class and the objects in the subclasses of the query/mining class. In the milk example above, the OR-FP algorithm can consider milk as the query/mining class. The instantiated mining class object(s) could be frequent patterns, and the instantiated subclass objects such as 1%milk, Dairyland1%milk, along with attributes such as *protein*, *calories*, *fat*, or *brand* could also be frequent patterns. However, the OR-FP algorithm does not specify at which hierarchy such patterns are frequent.

To answer the previously stated queries in this paper which were labeled as query 4 and query 5 (queries for mining frequent patterns in a transactional table), an algorithm is required which can mine the attributes of *Computer*, *Laptop*, and the *Deskto*p class, and in addition to also specify the hierarchy level at which the pattern is frequent at.

1.5 Hierarchical Frequent Pattern

As discussed in section 1.2, frequent patterns are itemsets that appear in a data set with a frequency (also called support) of not less than a user-specified threshold (also called minimum support). Therefore, the patterns are itemset, *I*, which is a set of items. For example, an itemset could be <*a*, *b*, *d*>, which means that itemset < *a*, *b*, *d* > are frequent patterns. The TidFP algorithm (Ezeife and Zhang (2009)) proposed a method that mines for frequent pattern(s) along with a transaction id list. Therefore, in TidFP, the frequent patterns are not just itemsets, but a combination of itemsets and their transaction id lists, *TI*. *T* is a list of transaction ids, and *I* is a set of items. For example, an itemset with transaction ids could be <T_1, T_2, T_4, *a*, *b*, *d*>, which means that frequent pattern <*a*, *b*, *d*> appears in transaction T_1, T_2, and T_4. In this thesis, we introduce a new term, called *hierarchical frequent pattern*. Because we are looking for the patterns that at different class hierarchy levels, we need to specify which hierarchy level the pattern belongs to. We will still mine the frequent patterns by their transaction ids. Therefore, our hierarchical patterns are represented by the forms of <Tidlist, itemset, class_i> (*TIC*). *TI* is the frequent patterns with transaction ids, same as in the TidFP algorithm. *C* is the class hierarchies that the pattern belongs to. For example, a hierarchical frequent pattern could be <T_1, T_2, T_4, *a*, *b*, *d*, computer/IBM>, which means that the frequent pattern <*a*, *b*, *d*> appears in transaction T_1, T_2, T_4 and belongs to class hierarchy IBM computer.

Definition 4 (Frequent patterns with specifying class hierarchy): Hierarchical Frequent Pattern, *HFP*, is represented in the format of <Tidlist, itemset, $class_i$> and is used to indicate in which transactions and at which class hierarchy a frequent pattern appears.

Example 1.6 (hierarchical frequent pattern): Show a hierarchical frequent pattern of computer object database that consists of transaction ids (Tidlist), itemset, and hierarchy.

Solution 1.6: <*1*, *3*, *4*, *2GHz, 2G*, laptop/computer/IBM>. 1,3,4 are transaction ids(Tidlist), 2GHz, 2GB are itemsets, and laptop/computer/IBM is the hierarchy.

1.6 Thesis Contributions

This thesis proposes a series of methods for mining frequent patterns from multiple data sources and multiple tables in object-oriented model which include joining object database tables, integrating multiple data sources, and mining hierarchical frequent patterns.

This thesis argues that: 1) it is beneficial to capture multiple real life databases as object-oriented databases and 2) to develop techniques to mine such multiple object-oriented databases at different levels of inheritance hierarchy.

The following are main contributions of this thesis:

1. Define an object-oriented class model which has class attributes and class methods connected by (type, super_type) foreign keys relationship and database

schema as $C_i = (K, T, S, A, M, O)$, where K is object id, T is class type, S is super type, A is a set of attributes, M is a set of class methods, and O is a set of class objects.

2. Define the method called Object-Oriented Join (*OOJoin*) which joins super table C_{super} and sub class table C_{sub} by selecting the tuples which have distinct object id (K), Type (T), and Super_type (S), such that $C_{super}.K$ and $C_{sub}.K$ from the result of $C_{super} \bowtie C_{sub}$ where($C_{super}.T = C_{sub}.T$ or $C_{super}.T = C_{sub}.S$).
3. Define the new term, hierarchical frequent patt ern, HFP, formed as <Tidlist, Itemset, Hierarchy>, where Tidlist is a set of object id K, Itemset is a set of class attributes A, and Hierarchy is a set of classes, C_i (*class$_i$*). Hierarchical frequent pattern specifies at which hierarchy level the pattern is frequent and is an extension of the TidFP's pattern <Tidlsit, itemset>.
4. Propose an algorithm called MineHFPs that mines Hierarchical frequent patterns to answer frequent pattern mining queries and specify at which hierarchy level the pattern is frequent by traversing multiple database hierarchy tree (MHTree) with the 1-itemset candidate patterns and transaction IDs.
5. Propose a new method called *oomap-gen join* which is an extension of *map-gen join* used in TidFP algorithm for generating k-itemsets candidate patterns during the process of MineHFP algorithm to reduce the number of k-itemsets candidate patterns and avoid unnecessary intersecting of transaction ids by indexing the patterns by two position codes according to inheritance hierarchy, start position and end position and checking the position code before generating k-itemsets candidate patterns.

2. PREVIOUS/RELATED WORK

2.1 Frequent Pattern Mining

The early research work of frequent pattern mining is the Apriori algorithm (Agrawal and Srikant (1994)). Apriori algorithm generates the candidate itemsets and check the support of them by scanning the dataset. Han and et al. (2004) proposed the FP-tree algorithm, and algorithm that projects the dataset into a compressed tree structure so that it avoids multiple database scanning and candidate generation. Ezeife and Zhang (2009) proposed an algorithm called TidFP that uses a bitmap to count the support of candidate itemset to avoid multiple database scanning and also generate transaction ids of each frequent patterns at the same time. The TidFP algorithm also mines multiple tables by applying set operations on transaction ids.

2.1.1 Apriori

The Apriori algorithm was proposed by Agrawal and Srikant (1994). The algorithm is used to find the frequent itemsets and association rule in a transaction database. The main idea of the Apriori algorithm is generating candidate itemsets by apriori-join and scanning the database to count the support for each candidate. The large itemset will be the itemsets whose support count is equal to or greater than a given minimum support (*min_supp*) and considered as frequent itemset. Given a transaction database *D*, $\{I_1, I_2, I_3, I_4, I_5\}$ is a set of items. Table 2.1 is an example of a transaction database. From there, it is known that items I_1, I_2, I_5 appear in transaction 1. The task is to find all frequent itemsets whose support frequencies are equal to or greater than a given minimum support.

Transaction ID (TID)	Items
1	I_1, I_2, I_5
2	I_2, I_4
3	I_2, I_3
4	I_1, I_2, I_4

Table 2.1 Transaction Database

For instance, a given minimum support (*min_supp*) is 50%, all itemsets that appear in two or more than two transactions need to be found as frequent or large itemsets. The Apriori algorithm will first find frequent 1-itemset. All I_1, I_2, I_3, I_4 ,I_5 are candidate 1-itemset. From scanning the database (Table 2.1.1), it is known that I_1 appears in transactions 1 and 4. Its support count is 2. I_2 appears in all 4 transactions. Its support count is 4. I_4 appears in transaction 2 and 4. Its support count is 2. I_3 and I_5 only appear in one transaction. Therefore, the large itemsets are I_1, I_2, I_4. Next, candidate 2-itemsets need to be generated by applying an apriori-gen join. The apriori-gen join of large itemset L_i with L_i joins every itemset k of first L_i with every itemset n of second L_i where n > k and first (I-1) members of itemsets k and n are the same. In this example, I_1 will join I_2 and I_4. I_2 will join I_4, but I_1 will not join I_1, and I_2 will not join I_2. Candidate 2-itemsets are I_1I_2, I_1I_4 and I_2I_4. Support count of these three candidate 2-itemsets need to be checked by scanning the transaction database. I_1I_2 and I_2I_4 are large 2-itemsets, since their support counts are 2 which equals to minimum support. Candidate 3-itemsets will be generated by large 2-itemsets that is $I_1I_2I_4$. The Support count of $I_1I_2I_4$ is 1 which is less than minimum support. Therefore, there is no large 3-itemsets. The algorithm will halt, since the large itemset is an empty set. The problem with the Apriori algorithm is that large candidate generation and multiple scanning of a database will be costly in CPU time.

2.1.2 FP-Tree

Apriori algorithms use the candidate set generation and test approach. Apriori algorithm needs multiple scanning of a database, so it is very costly, especially when the dataset is large and minimum support is low. Han and *et al.* (2004) proposed a frequent-pattern tree (FP-tree) structure that projects the database to a compressed version to represent the frequent items and stores it in a prefix tree in descending order with their supports. The FP-tree is then mined using the FP-growth mining method. Table 2.1 will be used as an example to demonstrate the FP-growth method.

1. In the first scan, the frequent 1-itemsets are obtained with their support count. Keeping the same minimum support count of 2 as we used in previous section. The set obtained in first scan is stored in descending order of the support count denoted by *L*.

   $L = [I_2{:}7, I_1{:}6, I_3{:}6, I_4{:}2, I_5{:}2]$.
2. After the first scan, the FP-tree construction process begins with the creation of the root node labeled 'null'.
3. After this the database is scanned for the second time, all of the items in each transaction are processed in descending support count order and a branch is created for each transaction.
4. Transaction T1 has three items "I_1, I_2, I_5". Its L order would be "I_2, I_1, I_5". The tree branch created for this transaction is shown in Figure 2.1.

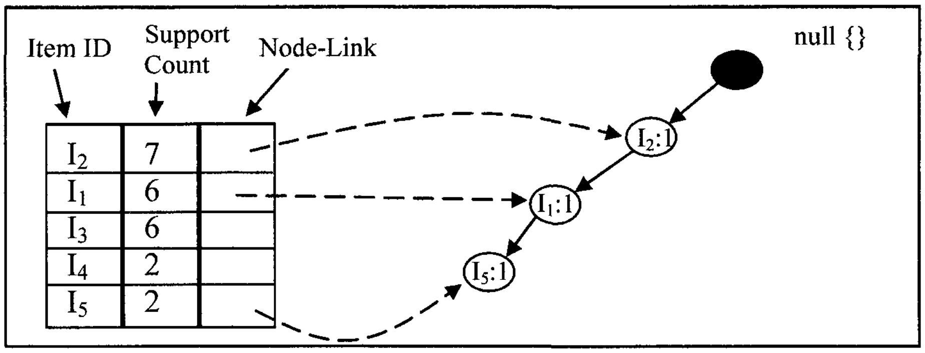

Figure 2.1 Tree branch for T1

5. This completes the transformation of first transaction from the database to one of the branches of the FP-tree.

6. Second transaction T2 consists of two items "I_2, I_4". Its L order would be "I_2, I_4". I_2 will be connected to the root and I_4 will be connected to the node I_2. The support count for I_2 will be incremented by 1 since this new branch shares the common prefix (I_2). Hence a new node I_4 will be created as a child note of I_2, this is shown in Figure 2.2.

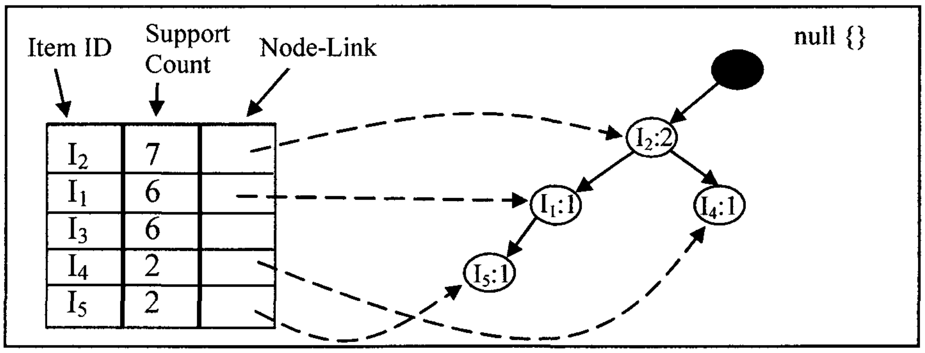

Figure 2.2 Tree branch for T2

7. The algorithm will run for all the transactions in the database and the resulting tree looks as shown in Figure 2.3.

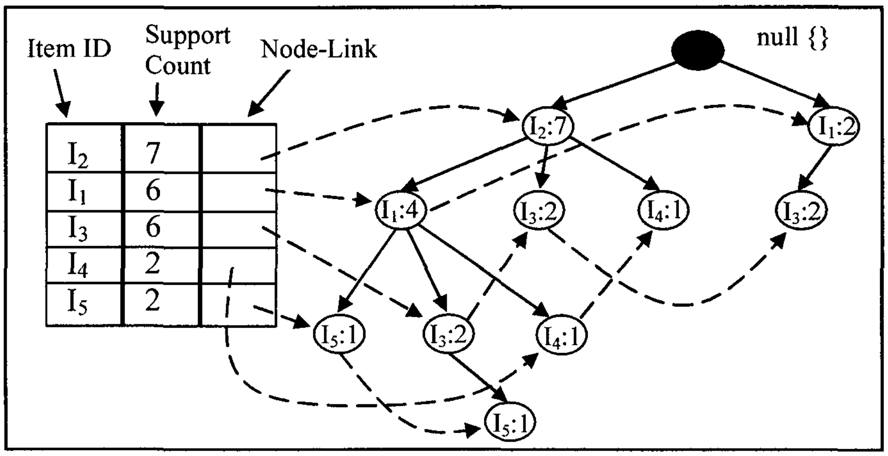

Figure 2.3 Complete FP tree

8. Along with the FP-Tree data structure, this algorithm also maintains an item header table. Each item from that table points to its occurrences in the FP-Tree using node-links.

9. Once the FP-Tree construction is complete, the mining process starts with the construction of the conditional pattern base from each frequent 1-itemset. Let us mine frequent patterns for I_3.

10. I_3 has two branches in its conditional FP-tree as shown in Figure 2.4, which generates the following set of patterns: $\{I_2\ I_3: 4, I_1\ I_3:2, I_2\ I_1\ I_3:2\}$.

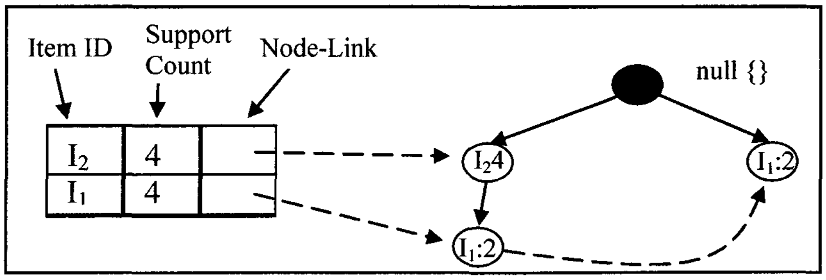

Figure 2.4 I_3 conditional FP Tree

Complete mining of FP-tree is shown in table 2.2.

Item	*Conditional pattern base*	*Conditional FP-tree*	*Frequent patterns generated*
I_5	$[(I_2\ I_1:1), (I_2\ I_1\ I_3:1)]$	$(I_2:2, I_1:2)$	$I_2\ I_5:2, I_1\ I_5:2, I_2\ I_1\ I_5:2$
I_4	$[(I_2\ I_1:1),(I_2:1)]$	$(I_2:2)$	$I_2\ I_4:2$
I_3	$[(I_2\ I_1:2),(I_2:2),(I_1:2)]$	$(I_2:4,I_1:2),(I_1:2)$	$I_2\ I_3:4, I_1\ I_3:2, I_2\ I_1\ I_3:2$
I_1	$[(I_2:4)]$	$(I_2:4)$	$I_2\ I_1:4$

Table 2.2 Complete mining result of FP-tree

2.1.3 TidFP Algorithm

Ezeife and Zhang (2009) proposed the TidFP algorithm which mines the frequent patterns with transaction ids. Mining patterns with transaction ids contributes in two aspects. First, it improves the efficiency of the mining process. Second, it mines more informative patterns not only from one database table, but also from multiple related tables to answer more complex queries.

The TidFP algorithm proposed a *map-join* algorithm. In Apriori algorithm, the *ap-gen* join of a large itemset L_i with L_i joins every itemset k of first L_i with every itemset n of second L_i where n > k and first (I-1) members of itemsets k and n are the same. The *map-gen join* performs the same operation on join of the itemset, however, it also intersects the transaction id lists of candidate itemsets. Table 2.1 will be used as an example of a transactional database. The first step of TidFP scans the database once and obtains all 1-item candidate itemsets with their transaction IDs in the format of a list a transaction IDs and itemset. In the table 2.1, all 1-item candidate itemsets are:

<1, 4> I_1, <1, 2, 3, 4>I_2, <3>I_3 , <2,4>I_4, <1>I_5

The support count of the 1-item itemset will be performed by counting the number of transaction ids of every item. If we are looking for the frequent pattern having a minimum support of 50%, the large 1-item itemsets are:

L1 = {<1, 4> I_1, <1, 2, 3, 4>I_2, <2, 4>I_4}

Candidate 2-item itemsets will be found by performing L1 *map-join* L1: The parts of the itemset will be obtained the same as ap-gen join and the transaction ids will be derived by intersecting transaction ids. For example,

C2 = {<1, 4> I_1I_2, <4> I_1I_4, <2, 4> I_2I_4}

The part representing transaction ids will be computed by counting the number of ids in a transaction id list. Because we are looking for the frequent pattern having a minimum support of 50%, the number of ids having less than 2 (out of 4) will be discarded. We obtain the large 2-item itemsets:

L2 = {<1, 4> I_1I_2, <2, 4> I_2I_4}

Candidate 3-item itemsets will be found by performing a *map-join*:

C3 = {<4> $I_1I_2I_4$}

By counting the transaction ids of $I_1I_2I_4$, we can determine that itemset $I_1I_2I_4$ is not a large itemset. Finally, we obtain the large itemsets with their transaction ids:

L = {<1, 4> I_1, <1, 2, 3, 4>I_2, <2, 4>I_4, <1, 4> I_1I_2, <2, 4> I_2I_4}

TidFP algorithm only needs to scan the database once and intersecting the id lists is performed by a bitmap operation, which significantly improves the efficiency of discovering frequent patterns.

The TidFP algorithm can also mine more complex knowledge from multiple database tables. For example, table 2.3 and table 2.4 are drug/side effects table and patient/drugs table.

Tid (Drug)	Items (Side Effect)
D1	1 3 4
D2	2 3 5
D3	1 2 3 5
D4	2 5

Table 2.3 Example Drug/Side Effects Database Records

Patient	Drug
P1	D1 D2
P2	D1 D2 D3
P3	D3 D4
P4	D1 D2 D4

Table 2.4 Example Patient/Drugs Database Records

The frequent patterns are discovered with their transaction ids in both tables, and then we can apply set operation such as intersection, union, or minus, which is able to answer queries as:

1. How many people have various patterns and frequent patterns of adverse effects given minimum 50% total occurrence?

Solution: Mining frequent patterns with transaction ids from the Patient/Drugs database records and mining frequent patterns with transaction ids from the Drug/Side Effects database records. Intersect the foreign keys of frequent patterns with transaction ids.

2. How many people use frequent combinations of products having minimum total occurrence of 50%?

Solution: Mine frequent patterns with transaction ids from the Patient/Drugs database records and then count the distinct ids.

3. Which drugs have dangerous combinations of adverse effects?

Solution: Mining the frequent patterns with transaction ids from the Drug/Side Effects database records, and then obtain the ids.

2.2 Mining Distributed Databases

This section reviews some of the algorithms for mining association rule from distributed databases. These algorithms are based on Apriori algorithm and apply parallel and distributed computing techniques. Also, these studies focus on less candidate generation and less message exchange over the distributed system.

2.2.1 Count Distribution (CD)

The algorithm CD was proposed by Agrawal and Shafer (1996). In the CD algorithm, every node generates and counts its own local itemsets, and then broadcast its itemsets with count to all other nodes. Every node will have identical candidate itemsets with the same order. Therefore, only the count of each itemset needs to be broadcast over the network. Table 2.5 is an example of distributed database. Consider mining the global frequent itemsets with a minimum support of 50% over three sites and 12 transactions in total.

TID	*Items*
100	*a b e*
200	*b d*
300	*b c*
400	*a b d*

Site 1

TID	*Items*
500	*a b f*
600	*b d*
700	*b c*
800	*a b d*

Site 2

TID	*Items*
900	*a b e*
1000	*b d*
1200	*b c*
1300	*a b d*

Site 3

Table 2.5 Distributed Databases

Every site scans the database and counts for 1-itemset locally, as shown in Table 2.6.

Itemset	*Count*
a	2
b	4
c	1
d	2
e	1

Site 1

Itemset	*Count*
a	2
b	4
c	1
d	2
f	1

Site 2

Itemset	*Count*
a	2
b	4
c	1
d	2
e	1

Site 3

Table 2.6 1-itemsets

Every site broadcast its own 1-itemsets and their local counts, and then every site will have identical candidate itemsets, as shown in Table 2.7 which are summed at each site to get these global counts.

Itemset	*Count*
a	6
b	12
c	3
d	6
e	2
f	1

Site 1

Itemset	*Count*
a	6
b	12
c	3
d	6
e	2
f	1

Site 2

Itemset	*Count*
A	6
B	12
C	3
D	6
E	2
F	1

Site 3

Table 2.7 1-itemset and counts

Since the minimum support is 50%, the candidates whose global counts are less than 6 will be eliminated. The global large 1-itemsets is shown in Table 2.8.

Large Itemset	*Count*
a	6
b	12
d	6

Site 1

Large Itemset	*Count*
a	6
b	12
d	6

Site 2

Large Itemset	*Count*
a	6
b	12
d	6

Site 3

Table 2.8 Large 1-itemset and counts

2-item candidates will be generated from every local site by apriori-gen join and count will be calculated locally, as shown in Table 2.9.

Large Itemset	***Count***
ab	2
ad	1
bd	2

Site 1

Large Itemset	***Count***
ab	2
ad	1
bd	2

Site 2

Large Itemset	***Count***
ab	2
ad	1
bd	2

Site 3

Table 2.9 2-itemset and counts

Since all candidates are identical and same ordered in every site, only a count of each itemset needs to be broadcasted, as shown in Table 2.10.

Large Itemset	***Count***
ab	6
ad	3
bd	6

Site 1

Large Itemset	***Count***
ab	6
ad	3
bd	6

Site 2

Large Itemset	***Count***
ab	6
ad	3
bd	6

Site 3

Table 2.10 Large 2-itemset and counts

From table 2.10, it can be seen that *ad* is less than 6 and should be dropped and large 2-itemsets will be *ab* and *bd*. Then candidate 3-itemsets will be generated by apriori-join locally on every site, which is *abd*. Count of *abd* is 1 on every site. After broadcasting count of *abd*, the global count of *abd* is found to be 3. Therefore *abd* is not frequent. Large 3-itemsets turned to be an empty set and algorithm will exit.

The CD algorithm will actually hash the itemsets and do the support counting by a vector summation in order to save time for comparing and matching the itemsets. It only broadcasts the count of each itemset, so that the message exchange over the network is small.

2.2.2 Distributed Mining Association Rules (DMA)

The DMA algorithm was proposed by Cheung, Ng, and Fu (1996). The authors claim that in the algorithm CD, candidates are generated in every local site redundantly. DMA can reduce candidate generation by generating candidate only from heavy itemsets and itemsets that is not locally large will be pruned away locally but still could be locally large in other sites. DMA proves that if an itemset is globally large, then there exists a site, where the itemset is locally large, also called local heavy itemset. Every site only generates candidate itemsets locally from local heavy itemsets and broadcasts to other sites to count the supports. Table 2.11 is an example of distributed databases. Consider to mine the global frequent itemsets with minimum support 50% over three sites, 12 transactions in total.

TID	*Items*
100	*a b e*
200	*b d e*
300	*b c e*
400	*a b d*

Site 1

TID	*Items*
500	*a b f*
600	*b d*
700	*b c*
800	*a b d*

Site 2

TID	*Items*
900	*a b e*
1000	*b d*
1200	*b c*
1300	*a b d*

Site 3

Table 2.11 Distributed Databases

Every site scans the database and counts for 1-itemset locally, as shown in Table 2.12.

Itemset	*Count*
a	2
b	4
c	1
d	2
e	3

Site 1

Itemset	*Count*
a	2
b	4
c	1
d	2
f	1

Site 2

Itemset	*Count*
a	2
b	4
c	1
d	2
e	1

Site 3

Table 2.12 1-itemsets

Every site only keeps its heavy 1-itemsets. In this case, every itemset that has a support count equal to or greater than 2 in its local site is considered as a heavy itemset, as shown in Table 2.13. Every site will also have to check its itemsets' global support count on other sites.

Heavy Itemset	*Count*
a	2
b	4
d	2
e	3

Site 1

Heavy Itemset	*Count*
a	2
b	4
d	2

Site 2

Heavy Itemset	*Count*
a	2
b	4
d	2

Site 3

Table 2.13 Heavy 1-itemsets

Candidate 2-itemsets are generated from heavy 1-itemsets locally, as shown in Table 2.14. Every itemset at its own site needs to check the global support count with other sites. However, only local heavy 2-itemsets will be kept to generate 3-itemset, as shown in Table 2.15.

Large Itemset	*Count*
ab	2
ad	1
ae	1
bd	2
be	3
de	1

Site 1

Large Itemset	*Count*
ab	2
ad	1
bd	2

Site 2

Large Itemset	*Count*
ab	2
ad	1
bd	2

Site 3

Table 2.14 2-itemsets

Large Itemset	*Count*
ab	2
bd	2
be	3

Site 1

Large Itemset	*Count*
ab	2
bd	2

Site 2

Large Itemset	*Count*
ab	2
bd	2

Site 3

Table 2.15 heavy 2-itemsets

All 3-itemsets will be generated locally from the heavy 2-itemsets, as shown in Table 2.16. Then it can be seen that none of the 3-itemset sets are heavy sets locally and large sets globally. The algorithm will terminate.

Large Itemset	***Count***
abd	1
abe	1
bde	1

Site 1

Large Itemset	***Count***
abd	1

Site 2

Large Itemset	***Count***
abd	1

Site 3

Table 2.16 heavy 3-itemsets

The DMA algorithm also uses a polling site to reduce the message exchange over the network. The polling site will avoid double message exchange during the process of global support counting. DMA generates less candidates than CD. However, CD only needs to broadcast the count of each itemset to other sites, because every site maintains identical candidate itemsets. On the other hand, DMA has to broadcast itemsets and their counts to other sites.

2.3 Mining Multiple-level Association Rules

Han and Fu (1995) claimed that previous association rule mining focused on mining from the single concept level. However, finding rules from a multiple concept level is also very useful. For example, finding rules such as 80% of customers that purchase milk may also purchase bread. It could be informative to also show that 75% of people buy wheat bread if they buy 2% milk. The authors also stated that large support is more likely to exist at a higher concept level, such as milk and bread, rather than at lower concept levels, such as a particular brand of milk and bread. If someone wants to find strong association rules at lower concept levels, the minimum support must be reduced. This action will result in finding some uninteresting rules, such as "toy->2%milk". To address these problems, a concept hierarchy needs to be created and every item will be encoded followed by the concept hierarchy, and also the different minimum support threshold should be used at mining different level of concept hierarchies. A data mining query is usually in relevance to only a portion of the transaction databases. For example, only food section is considered in a mining procedure. Figure 2.5 is a portion of concept hierarchy of food.

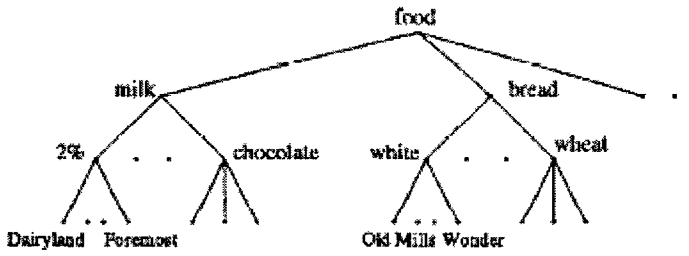

Figure 2.5 concept hierarchy

For example, "2% Foremost milk" is encoded as "112". Follow the concept hierarchy, the digit "1" represents milk at level one, the second digit "1" represents 2% milk at level 2, and the third digit "2" represents "Foremost" milk product at level three. By the same encoding schema, a transaction database will be encoded as Table 2.17

TID	Items
T1	{111, 112, 211, 221}
T2	{111, 211, 222, 323}
T3	{112, 122, 221, 411}
T4	{111, 121}
T5	{111, 122, 211, 221, 413}
T6	{211, 323, 524}
T7	{323, 411, 524, 713}

Table 2.17 Encoded transactions

Mining of finding frequent patterns begins from level one of a concept. In this example, "1**" represents all milk products and "2**" represents all bread products. Suppose we wish to find all the frequent itemsets at each concept level.

We start looking for the patterns at level-1 and minimum support is 4. Because the mining level is 1, for all items in Table 2.17, we only keep the first digit of the item. The table 2.17 should be represented as table 2.18.

TID	Items
T1	{1**, 1**, 2**, 2**}
T2	{1**, 2**, 2**, 3**}
T3	{1**, 1**, 2**, 4**}
T4	{1**, 1**}
T5	{1**, 1**, 2**, 2**, 4**}
T6	{2**, 3**, 5**}
T7	{3**, 4**, 5**, 7**}

Table 2.18 Encoded transactions

Level-1 large 1-itemsets is shown in Table 2.19 and Level-1 large 2-itemsets is shown in table 2.20.

Itemset	Support
{1**}	5
{2**}	5

Table 2.19 level-1 large 1-itemset

Itemset	Support
{1**, 2**}	4

Table 2.20 level-1 large 2-itemset

If an itemset is not large at a higher concept level, it must not be large at a lower concept level. Therefore, other itemsets which are not large at concept level one need to be filtered out. Table 2.21 is the filtered transaction.

TID	Items
T1	{111, 112, 211, 221}
T2	{111, 211, 222, 323}
T3	{112, 122, 221}
T4	{111, 121}
T5	{111, 122, 211, 221}
T6	{211}

Table 2.21 Encoded transactions

After large itemsets are found at level one, mining needs to be processed at level-two. The third digit will be ignored. The item will be represented as 11*, 12*, etc. The minimum support is changed to 3 at this level.

Itemset	Support
{11*}	5
{12*}	4
{21*}	4
{22*}	4

Table 2.22 level-2 large 1-itemset

Itemset	Support
{11*, 12*}	4
{11*, 21*}	3
{11*, 22*}	4
{12*, 22*}	3
{21*, 22*}	3

Table 2.23 level-2 large 2-itemset

Itemset	Support
{11*, 12*, 22*}	3

Table 2.24 level-2 large 3-itemset

Same as the process at Level one mining, large 1-itemsets will be found first, as shown in Table 2.22, then large 2-itemsets, as shown in Table 2.23, and large 3-itemsets, as shown in Table 2.24.

The minimum support of level-three is also 3. By the same process of level-one and level-two, the frequent itemsets are found as shown in Table 2.25 and Table 2.26

Itemset	Support
{111}	4
{211}	4
{221}	3

Table 2.25 level-3 large 1-itemset

Itemset	Support
{111, 211}	3

Table 2.26 level-3 large 2-itemset

2.4 Frequent pattern mining in object-oriented model

This section introduces several methods that mine frequent patterns and association rules from an object-oriented model or from object-oriented databases. Fortin and Liu (1996) proposed the method that mines multi-level association rules in object-oriented model. The method can flexibly combine multiple multi-level concept hierarchies for mining more informative and refined knowledge from the relational databases. Han and et al. (1997) proposed Generalization-based frequent pattern mining approach that generalizes the complex data objects from object-oriented databases, including object identifier, single attribute values, structured data and etc. to mine the frequent patterns and association rules. Kuba and Popelinsky (2003) proposed an algorithm called OR-FP that mines the frequent patterns as class objects and attributes from the query class table and its subclass tables in object-oriented database.

2.4.1 An Object-Oriented Approach to Multi-Level Association Rule Mining

Fortin and Liu (2005) claimed that an object-oriented approach is more flexible and more informative for mining multi-level association rule for a single concept hierarchy and across multiple concept hierarchies.

The authors stated the drawbacks of relational database tables.

1) Weak item representation. For example, in an item table, there is an attribute called "expiry_date". All items needs to be added this attribute in a flat relational database table, but for some products the attribute "expiry_date" has no meaning.
2) Relational databases do not present the concept hierarchies. For example, milk is a type of food and apple is type fruit. This concept cannot be demonstrated in a relational database table. There must a domain-specified expert to manually specify it.
3) A multi-level hierarchy cannot be treated as a unit of concurrency control in relational system. A concurrent update may cause data loss.

To address the above problems, the authors proposed an object-oriented approach. The object-oriented mode can explicitly represent the concept of hierarchies. Unlike relational system in Han and etc. (1996), in object-oriented model, concept hierarchies do not need to be predefined by a domain expert. Instead, these concept hierarchies can be dynamically adjusted by user queries. High levels concept hierarchies are captured explicitly and uniformly in the form of classes or objects in classes, and lower level of concept hierarchies can be adjusted dynamically upon request. For example, milk and bread are the subclasses of food. Both Diaryland skim milk<Diaryland milk < skim milk < milk and Diaryland skim milk < skim milk < Diaryland milk < milk are valid hierarchies. Furthermore, the concept hierarchies can be build not only on categories but also on other information. For example, the hierarchy can be built based on promotion and on sale information, as shown in Figure 2.6.

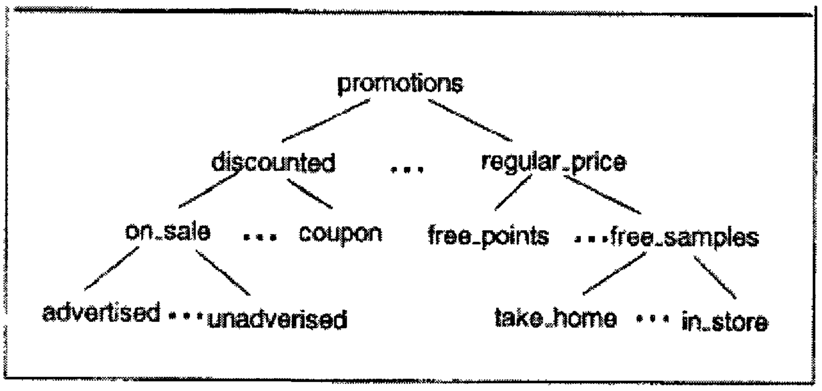

Figure 2.6 concept hierarchy

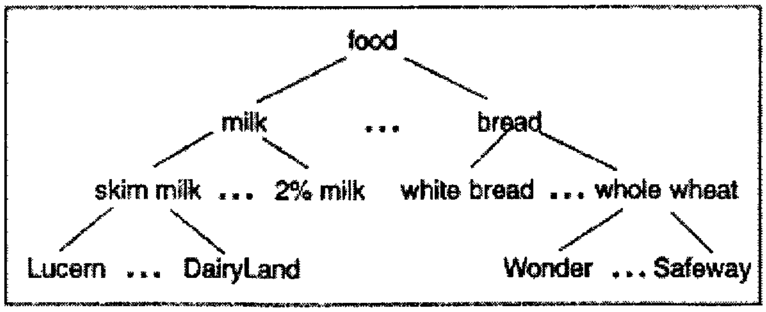

Figure 2.7 concept hierarchy

Authors also proposed an adaptive encoding scheme. Suppose we wish to find a rule relating specific milk products to bread. A specific milk product needs to be encoded.

Follow the concept hierarchy as Figure 2.7. A DairyLand skim milk can be encoded as 1,1,1,2 which is encoded at each of the four conceptual levels. Bread is encoded as 1,#,#,2. "#" is the dummy value that makes bread and a specific milk be at the same encoding level. Because a specific milk product and bread are encoded at the same level, mining can be processed in the same way as in Han and etc. (1996) (See section 2.5.2). Adaptive encoding scheme can also encode the items on different hierarchies and conjunct the code to discover more complex knowledge. For example, an item is encoded as "1,2,1,3" and "1,4,5,1". "1,2,1,3" is encoded from a product category hierarchy and "1,4,5,1" is encoded from "on sale" hierarchy. Milk is encoded as "1,2 and on sale is encoded as "1,4". Conjunct two code to result conjunctive item "11,24,15,31" where "11" represents food & promotion. "24" represents milk & on sale. Thus a user query might be "find me all rules invoking milk items at level 4, bread items and level 2, and items which are both cheese and on sale".

2.4.2 Mining Frequent Pattern in Object-oriented Data (OR-FP)

Kuba and Popelinsy (2003) proposed an algorithm of mining frequent patterns in object-oriented data, called OR-FP. The authors claimed that frequent patterns in object-oriented data are the patterns that cover at least a given minimum number of objects. This research work also introduced the term of frequent patterns in object-oriented data, that is frequent patterns are considered as objects. The simplest pattern is X:T, where X is a variable representing an object and T is the type of this object. The patterns in object-oriented model will be extended depending on the last variable of the existing patterns. The last variable of existing pattern X_i and its type is T_i. $X_i : T_i$ could be simple type, class type, or collection type. A simple type could be a *String* Type. For example, a class *Person,* whose attributes last_name is a *string* type. A class type variable will be extended with attributes and with every subclass of the given class. For example, if the last variable whose type is class *Person,* the pattern should be extended by every attributes of *Person* class, such as *name*, *salary*, or *age*. The pattern also should be extended by all subclasses of the Person class, such as class *Actor* and class *Director*. A collection type is a set. For example, the type of attribute *acts_in* of *Actor* class is a set of movies. The frequent patterns in object-oriented data are the patterns that cover at least a given minimum number of objects. The task of OR-FP is to find frequent patterns for a given query class. If a pattern is supposed to be frequent, it covers at least a given minimum number of objects of query class and its subclasses. An example for an object-oriented database schema of cinema is given as follow:

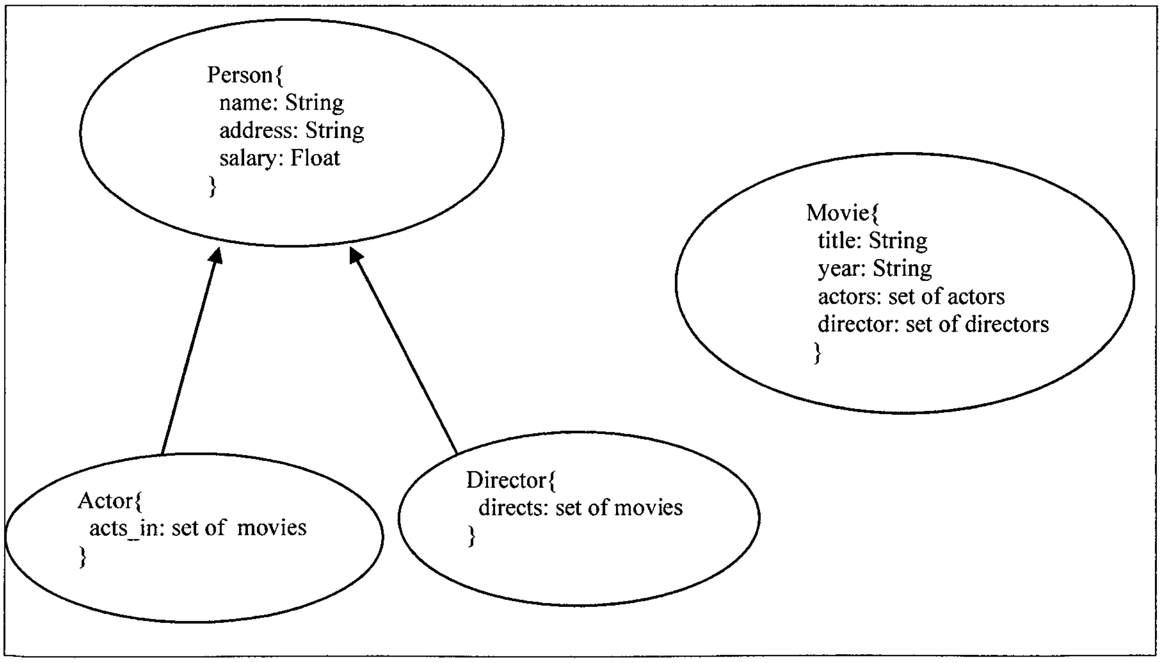

Figure 2.8 class inheritance hierarchy of Person, Actor, Director and Movie

In Figure 2.8, there are four classes, *Person*, *Movie*, *Actor* and *Director*. The *Actor* class and *Director* class are subclasses of the Person class. The data in the object-oriented database is represented as:

o_i*: class = {attribute*$_1$*, attribute*$_2$*, ..., attribute*$_n$*}*
Small part of data used in the example is given as follows:

o1: Person = {'Smith', 'Canada', 16000}
o2: Actor = {'John', 'Canada', 12000}
o3: Actor = {'William', 'Canada', 15000, {o6}}
o4: Actor = {'Mike', 'US', 18000, {o6}}
o5: Director = {'Steven', 'US', 25000, {o6}}
o6: Movie = {'prostriziny', 1980, {o1, o2}, o5}

The OR-FP algorithm mines the frequent pattern in the mining class and also the frequent patterns in the subclasses of the mining class. In the above example of OR-FP, we are looking for the frequent patterns that have a minimum support of 3. The patterns that cover at least 5 objects are considered frequent patterns. The query/mining class is the *Person* class. Therefore, the inputs of the algorithm are: the movie object-oriented database, the query/mining class *Person*, and the minimum support level of 3.

The candidate patterns start from the query/mining class *Person*. The candidate patterns are a list of *Person* objects, {X_0*:Person*}. Actor objects and Director objects are also Person objects. There are 5 Person objects in total. The pattern <X_0:Person> is a frequent pattern. <X_0:Person> should be extended by the attributes of the *Person* class. For example there may be 3 persons' whose addresses is "Canada", and 4 persons' whose salary is within the range (15000, 20000). The new patterns are generated, <X_0:Person.address=X_1:String="Canada"> and <X_0:Person.salary= X_2:Float=(15000,2000)>. Also, the 2-itemset pattern <X_1:Person.address:String="Canada", X_2:Person.salary:Float=(15000, 20000)>. The class Person has subclasses Actor and Director. Pattern X_0:Person should be extended by the Actor class and the Director class and check their supports. There are 3 Actor objects and only 1 Director object. Therefore, the new pattern should be generated, <X_0:Person, X_0:Actor>.

The class actor has the attributes. The pattern should be extended by the attributes of the class Actor. For example there are 3 actors' whose salary is in the range (15000, 2000). The new generated pattern is <X_0:Person, X_0:Actor.salary: Float = (15000,2000)>. The Actor class does not have any subclasses, so it will not be further extended.

3. Mining Object-Oriented Multiple Databases

As discussed in section 1.4, our goal is to define a series of methods that answer *frequent pattern mining* queries from multiple related tables (related by class hierarchies) and the database table which is integrated by multiple data sources. In this thesis, we define the object-oriented class model and a set of class methods for various classes. These class methods are able to integrate multiple data sources (updating the *Root* class table), join object tables, and answer *frequent pattern mining* queries. The object-oriented class model is defined as follows:

```
Root{
    a set of transaction attributes A_t //including super_type and all physical attributes of C_t
    private void InsertTransactions;
    private set MineRootFPs;
    public set OOJoin;
}
C_t{
    a set of physical attributes A_t
    private set MineClassFPs;
}
```

Class C_t has a set of physical attributes which are the properties of the class C_t. In the example of an object orientated database representing a computer, the database could be represented by a class called *Computer,* and the physical attributes of the computer could be the "CPU", "RAM", and "Hard_dirve". The *Laptop* class could be an extension of the *Computer* class and its attributes could be things like the "Screen_size" and "Battery_life". We define *transactions attributes* to be composed of two parts. The first part is the set of super_t which is a super_type of class C_t. The second part is comprised of all the physical attributes of class C_t. The *Root* class embodies the *transaction attributes*.

The private method *InsertTransactions* of the *Root* class is used to insert transactions into the Root table and can only be called by the class *Root*. As discussed in section 1.2, in the example of a computer object database, the Root table is a sales transaction table which records the purchase of computers from all the databases. The private method *MineRootFPs* of the *Root* class is used to discover the *hierarchical frequent patterns* (HFPs) in the Root table and this method can also only be called from the class Root. This method is able to answer the queries labeled as query 4 and query 5 in section 1.4. *OOJoin* is a public method that joins a supper class table and a sub class table, and can be invoked by other classes. The private method *MineClassFPs* is used to mine the frequent patterns of a specific class in each database. For example if the *MineClassFPs* is called from a derivative class like *Laptop*, it will mine the *Laptop* class and in addition it will also mine the *Computer* class because the *Laptop* class derives from the *Computer* class. This is accomplished via calling of the public method *OOJoin*. However if the *MineClassFPs* method is invoked in the class *Computer*, it will only mine the *Computer* class. Thus, *MineClassFPs* is able to answer query 1 or query 2 in section 1.4. Section 3.2 will discuss the *MineClassFPs* method of class C_t in more detail. This method mines frequent patterns in each class for each database. In section 3.2 the paper will also give the detailed algorithm for *OOJoin*. Section 3.3 will discuss the private methods *InsertTransactions* and *MineRootFPs* of the class Root.

3.1 Problems Addressed in Mining Multiple Object-oriented Databases

1. Frequent pattern mining algorithms, such as *Apiori* (Agrawal and Srikant (1994)) and *FP-tree* (Han and *et al.* (2004)), can only mine the frequent patterns from a single database table. They cannot discover the frequent patterns from multiple tables and multiple data sources. Also, they cannot discover the patterns at different class hierarchies, as the inputs of these algorithms are simple transactional database tables, there are no class hierarchies.

2. The TidFP algorithm proposed by Ezeife and Zhang (2009) mines frequent patterns first, generating frequent patterns with their transaction ids (called TidFp), then applying set operations on the TidFps to answer frequent pattern related queries across multiple database tables. The TidFP algorithm does not mine frequent patterns in object-oriented multiple databases nor does it specify the hierarchy levels that patterns belong to.

3. Existing work, such as Mining Multi-level Association Rule (Han and Fu (1995)) and Object-Oriented Approach to Multi-Level Association Rule Mining (Fortin and Liu (1996)) replace the patterns by another pattern in a higher or a lower hierarchy level and discover frequent patterns at different concept hierarchy levels. However, these algorithms do not take object databases as inputs and do not consider the instantiated objects or object attributes as patterns.

4. The OR-FP algorithm takes an object-oriented database as input and mines instantiated class objects and attributes of objects as frequent patterns. However, it does not mine multiple object databases and does not specify at which hierarchy level patterns are frequent.

5. The database tables that represent classes at different hierarchies representing different databases need to be joined in an object-oriented model.

6. The TidFP algorithm is an efficient algorithm, because it counts the support of every candidate pattern by intersecting transaction ids, so that it avoids multiple scanning of a database. However, generating candidate patterns is costly. Also, when the dataset is large, intersecting transaction ids and counting intersected transaction ids are also expensive processes. Reducing the candidate itemset generation and avoiding unnecessary support counting are important.

3.2 Mining Frequent Patterns for Each Class

This section discusses the private method *MineClassFPs* of class C_i which mines frequent patterns in each class. Section 3.2.1 gives main algorithm and process flow of *MineClassFPs*. Section 3.2.2 gives the detailed algorithm for *OOJoin* with examples.

3.2.1 Algorithm and Process Flow for Mining Frequent Patterns for Each class

The private method *MineClassFPs* outputs a set of class attributes as frequent patterns. The algorithm of *MineClassFPs* is provided in Figure 3.1

```
Algorithm MineClassFPs($C$, $CS_i$, $s\%$)

Input: class table $C$ //the class table to be mined,
        class tables $CS_i$ // a set of super class tables of $C$, i = 1,2...k. $CS_2$ is superclass
                        of $CS_1$, $CS_k$ is the superclass of $CS_{k-1}$
        minimum support $s\%$.

Other variables: Joined class table $T$

Output: A set of frequent patterns FPs.

Begin

  1.0 JoinClasses ($C$, $CS_i$)
      1.1 $T = C$;
      1.2 if($CS_i \neq$ NULL) // C has super classes.
          1.2.1 For each superclass table $CS_i$
                1.2.1.1 $T$ = OOJoin($CS_i$, $T$); // call OOJoin to join subclass and
                                                   superclass
          End for
      End if
  2.0 TidFP($T$, $s\%$);

End
```

Figure 3.1 Algorithm for method MineClassFPs

The class method *MineClassFPs* can be used to answer query 1 or query 2 in section 1.4. Query 1 is used to mine the frequent patterns (computer specifications) for *Computer* class. The *Computer* class does not have any superclass. The algorithm of JoinClasses will only execute step 1.1. The joined class table is the *Computer* class table. Then the algorithm of *MineClassFPs* goes to step 2.0, and then the *MineClassFPs* method calls the TidFP algorithm to mine the frequent patterns. Query 2 is used to mine the frequent patterns (Laptop specifications) for the *Laptop* class. The *Computer* class is a super class for the *Laptop* class. Because the *Laptop* class has a superclass, the sub-algorithm *JoinClasses,* will execute step 1.1 and step 1.2. Step 1.2 needs to invoke the *OOJoin* algorithm to join the table of the superclass (Computer) and the table of the subclass

(Laptop). *OOJoin* will be discussed in detail in section 3.2. Then the *MineClassFPs* algorithm goes to step 2.0 to call the TidFP algorithm to mine for the frequent patterns. Figure 3.2 is the process flow of mining each class.

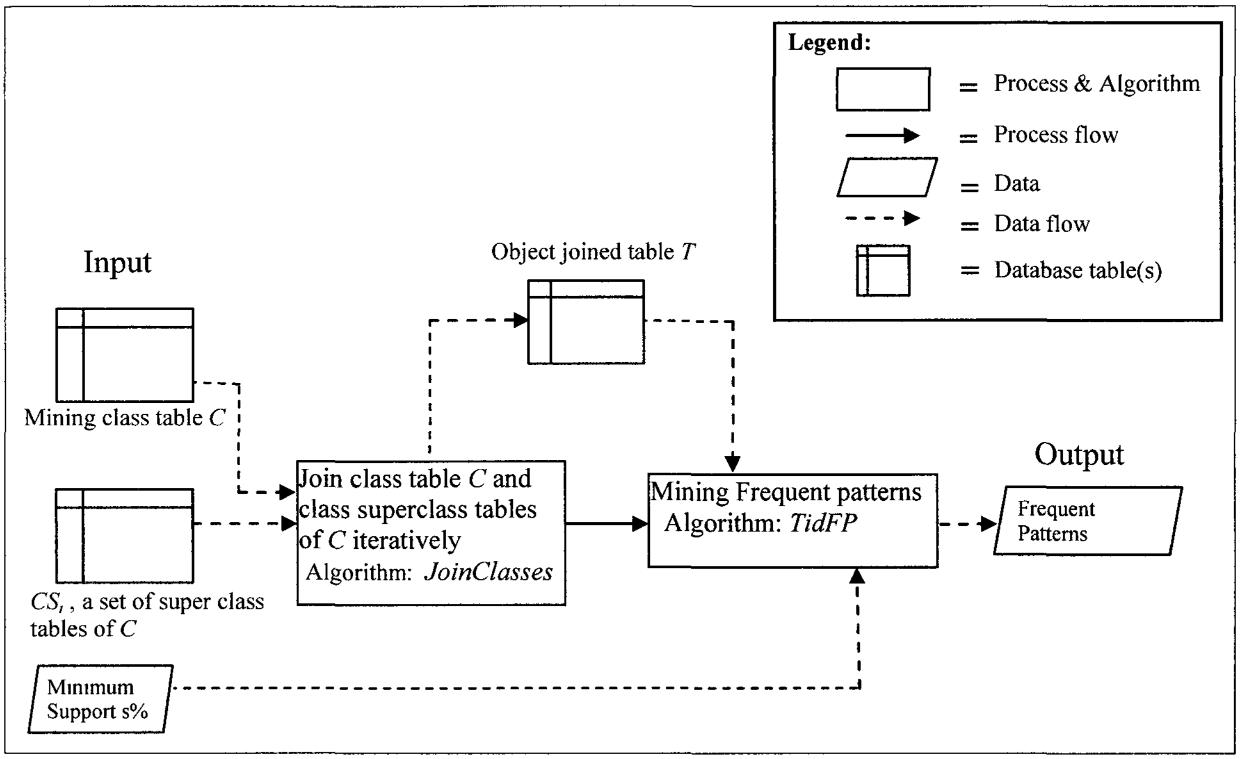

Figure 3.2 Process flow for mining FPs for each class

In Figure 3.2, the inputs are the mining class *C* (the class table to be mined), a set of super class tables CS_t of *C*, and the minimum support *s%*. The private method *MineClassFPs* calls algorithm *JoinClasses* which applies the *OOJoin* algorithm to join the mining class *C* and all super class tables CS_t of *C* iteratively to obtain an object joined table *T*, and then call the TidFP algorithm with a specific minimum support of *s%* to mine the frequent patterns on the object joined table *T*.

3.2.2 Object-Oriented Join (*OOJoin*)

OOJoin was discussed previously in section 1.2. This section shall explain how the

OOJoin algorithm functions to join a super class table and a subclass table in an object-oriented model.

The first step of the *OOJoin* algorithm is to cross product every tuple in the super class table and the sub class table. The resulting tuples from the cross product operation contain all the attributes of the superclass (Computer) and the subclass (Laptop), including a number of keys (the primary keys and the foreign keys). The superclass keys are: the primary key of the superclass (K_1); the first foreign key which is the *type* for the superclass (T_1), and the second foreign key which is the *super_type* for the superclass (S_1). The keys representing the subclass in the tuple are: the primary key of the subclass (K_2); the first foreign key of the subclass which is the *type* for the superclass (T_2), and the second foreign key which is the *super_type* for the subclass (S_2).

The second step is to discard certain tuples from the result of the cross product operation. For each tuple, the foreign key T_1 is compared with foreign key T_2. If T_1 matches T_2, or T_1 matches S_2 then the tuple will be kept, else the tuple will be discarded.

The third step in the algorithm further prunes the list of tuples. The first tuple is always kept. Two lists are created, each list is a list of primary keys. The first list will be reffered to as $List_1$ and it is used to store the K_1 primary keys and the other list shall be reffered to as $List_2$ and it is used to store the K_2 primary keys. For each tuple, starting with the second tuple, we first check if K_1 of the current tuple is already in $List_1$. If it is, then this tuple will be discarded. Else, if K_1 is not already in $List_1$, then we check if K_2 is already in *List*$_2$. If it is, then the tuple will be discarded, else the tuple is kept.

The *OOJoin* algorithm is shown in Figure 3.3

```
Algorithm OOJoin($C_{super}$ , $C_{sub}$)

Input: Super class table $C_{super}$, Sub class table $C_{sub}$
Other variables:
$T_c$: set that contains result of cross product of two class tables, initialized as empty
The superclass primary key $K_1$, the superclass foreign keys $T_1$ and $S_1$.
The subclass primary key $K_2$, the subclass foreign keys $T_2$ and $S_2$.
$T_t$: set that contains tuples selected by constraints $C_{super}$. $T_1$ = $C_{sub}$. $T_2$ or $C_{super}$. $T_1$ =
$C_{sub}$. $S_2$), initialized as empty.
$T_d$: to store output tuples, initialized as empty.
$List_1$: set of IDs of super class table, initialized as empty.
$List_2$: set of IDs of sub class table, initialized as empty.

Output: A set of tuples of objects $T_d$.

Begin

 1.0 $T_c = C_{super} \times C_{sub}$.

 2.0 $T_t$ = select from $T_c$ where ($C_{super}$. $T_1$ = $C_{sub}$. $T_2$ or $C_{super}$. $T_1$ = $C_{sub}$. $S_2$)

 3.0 select a set of distinct tuples $T_d$ from $T_t$;
     3.1 insert the first tuple $t_1$ of $T_t$ into $T_d$;
     3.2 insert object id of superclass part in $t_1$ into $List_1$;
     3.3 insert object id of subclass part in $t_1$ into $List_2$;
     3.4 For each tuple $t_x$ left in the $T_t$
            3.4.1 If ($K_1$ does not exist in $List_1$ && $K_2$ in $t_1$ does not exist in $List_2$)
                  3.4.1.1 Insert $t_x$ into $T_d$;
                  3.4.1.2 Insert $K_1$ in $t_x$ into $List_1$;
                  3.4.1.2 Insert $K_2$ in $t_x$ into $List_2$;
            End if
      End for

End
```

Figure 3.3 algorithm OOJoin

The "Computer" table and "Laptop" table are shown in Table 1.1 and Table 1.2 respectively. The following is a detailed example of the operations of the *OOJoin* algorithm on Table 1.1 and Table 1.2. The result is a set of joined instantiated objects of "Computer" and "Laptop".

Table 3.1 shows the result of step 1.0 of the *OOJoin* algorithm. The tuples shown are the cross product of the super class table "Computer" (Table 1.1) and the subclass table "Laptop" (Table 1.2).

ID	Type	Super	CPU	RAM	Hard Drive	Comp Name	ID	Type	Super	Screen Size	Battery Life
comp1	Laptop	Computer	2GHz	2G	250G	Idea laptop	lapt1	Ideapad laptop	Laptop	15"	3hrs
comp2	Laptop	Computer	2GHz	2G	320G	Idea laptop	lapt1	Ideapad laptop	Laptop	15"	3hrs
comp3	Laptop	Computer	3GHz	4G	350G	Think laptop	lapt1	Thinkpad laptop	Laptop	15"	3hrs
comp4	Desktop	Computer	3GHz	4G	500G	Wk station	lapt1	Work station	Laptop	15"	3hrs
comp5	Desktop	Computer	3GHz	4G	500G	Wk station	lapt1	Work station	Laptop	15"	3hrs
comp6	Desktop	Computer	3GHz	4G	500G	IBM PC	lapt1	desktop	Laptop	15"	3hrs
comp1	Laptop	Computer	2GHz	2G	250G	Idea laptop	lapt2	Ideapad laptop	Laptop	15"	3hrs
comp2	Laptop	Computer	2GHz	2G	320G	Idea laptop	lapt2	Ideapad laptop	Laptop	15"	3hrs
comp3	Laptop	Computer	3GHz	4G	350G	Think laptop	lapt2	Thinkpad laptop	Laptop	15"	3hrs
comp4	Desktop	Computer	3GHz	4G	500G	Wk station	lapt2	Work station	Laptop	15"	3hrs
comp5	Desktop	Computer	3GHz	4G	500G	Wk station	lapt2	Work station	Laptop	15"	3hrs
comp6	Desktop	Computer	3GHz	4G	500G	IBM PC	lapt2	desktop	Laptop	15"	3hrs
comp1	Laptop	Computer	2GHz	2G	250G	Idea laptop	lapt3	Ideapad laptop	Laptop	17"	3 5hrs
comp2	Laptop	Computer	2GHz	2G	320G	Idea laptop	lapt3	Ideapad laptop	Laptop	17"	3 5hrs
comp3	Laptop	Computer	3GHz	4G	350G	Think laptop	lapt3	Thinkpad laptop	Laptop	17"	3 5hrs
comp4	Desktop	Computer	3GHz	4G	500G	Wk station	lapt3	Work station	Laptop	17"	3 5hrs
comp5	Desktop	Computer	3GHz	4G	500G	Wk station	lapt3	Work station	Laptop	17"	3 5hrs
comp6	Desktop	Computer	3GHz	4G	500G	IBM PC	lapt3	desktop	Laptop	17"	3 5hrs

Table 3.1 result of "Computer" table cross product "Laptop" table

Table 3.2 shows the result of step 2.0 of the *OOJoin* algorithm. The remaining tuple list is smaller since the only tuples kept are the tuples where the keys T_1 and T_2 are matched, or T_1 is matched with S_2

ID	Type	Super	CPU	RAM	Hard Drive	Comp Name	ID	Type	Super	Screen Size	Battery Life
comp1	Laptop	Computer	2GHz	2G	250G	Idea laptop	lapt1	Ideapad laptop	Laptop	15"	3hrs
comp2	Laptop	Computer	2GHz	2G	320G	Idea laptop	lapt1	Ideapad laptop	Laptop	15"	3hrs
comp3	Laptop	Computer	3GHz	4G	350G	Think laptop	lapt1	Thinkpad laptop	Laptop	15"	3hrs
comp1	Laptop	Computer	2GHz	2G	250G	Idea laptop	lapt2	Ideapad laptop	Laptop	15"	3hrs
comp2	Laptop	Computer	2GHz	2G	320G	Idea laptop	lapt2	Ideapad laptop	Laptop	15"	3hrs
comp3	Laptop	Computer	3GHz	4G	350G	Think laptop	lapt2	Thinkpad laptop	Laptop	15"	3hrs
comp1	Laptop	Computer	2GHz	2G	250G	Idea laptop	lapt3	Ideapad laptop	Laptop	17"	3 5hrs
comp2	Laptop	Computer	2GHz	2G	320G	Idea laptop	lapt3	Ideapad laptop	Laptop	17"	3 5hrs
comp3	Laptop	Computer	3GHz	4G	350G	Think laptop	lapt3	Thinkpad laptop	Laptop	17"	3 5hrs

Table 3.2 result of selected rows

Step 3.0 of the *OOJoin* algorithm produ ces the final result table (Table 3.3). Th e algorithm always keeps the first tuple. Extract the ID of super class and ID of sub class of first tuple and store them in two sets super class ID set and sub class ID set, respectively: {comp1}, {lapt1}. For the rest of tuples in Table 3.2, start from the second tuple, check if the ID of sub class exists in the sub class ID set. In this case, the id of sub class is "lapt1" and "lapt1" is already in the set, so the second tuple should not be selected. The id of sub class is also "lapt1" of third tuple, so third tuple should not be selected either. The id of sub class is "lapt2" of Fourth tuple, so we need to check the id of super class, which is "comp1". "comp1" already exists in the set, so the fourth tuple should not be selected. The id of sub class is "lapt2" of fifth tuple, so we need to check the id of super class, which is "comp2". The fifth tuple should be selected and stored in the result table (Table 3.3). Then we store the super class ID and sub class ID of fifth tuple into the ID sets: {comp1, comp2}, {lapt1, lapt2}. By the same method we can obtain Table 3.3

ID	Type	Super	CPU	RAM	Hard Drive	Comp Name	ID	Type	Super	Screen Size	Battery Life
comp1	Laptop	Computer	2GHz	2G	250G	I laptop	lapt1	Ideapad Laptop	Laptop	15"	3hrs
comp2	Laptop	Computer	2GHz	2G	320G	I laptop	lapt2	Ideapad Laptop	Laptop	15"	3hrs
comp3	Laptop	Computer	3GHz	4G	350G	T laptop	lapt3	Thinkpad Laptop	Laptop	17"	3 5hrs

Table 3.3 Result table of OOJoin

OOJoin can be used to join the "Computer" and "Desktop" tables in the same way that the "Computer" and "Laptop" tables were joined.

3.3 Mining Frequent Patterns in Transaction (*Root*) Table

In section 3.2, we defined the class method *MineClassFPs* for mining *frequent patterns* for every class C_i in each object database. In this section, we define two methods *InsertTransactions* and *MineRootFPs* belonging to class *Root*. The method *InsertTransactions* is used to update the Root table and the method *MineRootFPs* is used to mine *hierarchical frequent patterns* (HFPs) in the Root table. Section 3.3.1 will introduce some definitions and algorithms that will be used in the above two methods. Section 3.3.2 will give the process flow diagram of the methods *InsertTransactions* and *MineRootFPs*. Section 3.3.3 will provide the details of the *InsertTransactions* method. Section 3.3.4 will provide the details of the *MineRootFPs* method.

3.3.1 Definitions and Algorithms used by Methods of *Root* Class

This section will introduce some definitions and algorithms that will be used in the methods *InsertTransactions* and *MineRootFPs* of the class *Root*. Section 3.3.1.1 will introduce Inheritance Hierarchy Tree (slightly different from the MHTree induced in section 1.2). Section 3.3.1.2 will discuss pre-order traversal and the position coding method used in the PLWAPLong algorithm proposed by Ezeife, Saeed, and Zhang (2009). Section 3.3.1.3 will discuss the *map-gen join* algorithm used in the TidFP algorithm proposed by Ezeife and Zhang (2009). Section 3.3.1.4 will introduce the *oomap-gen join* algorithm, an extended version of *map-gen join* which is suitable for mining object-oriented data.

3.3.1.1 Inheritance Hierarchy Tree

As we discussed in ***Definition 1***, in section 1.2, a multiple database class inheritance hierarchy can be represented in a tree structure called MHTree as shown in Figure 1.3 in section 1.2. However, the class inheritance hierarchy in each database can also be represented in a tree structure.

Definition 5 (structure of inheritance hierarchy): Inheritance Hierarchy Tree, HTree, is a tree structure representation of inheritance hierarchy ***H***(class and superclass relationships). For example, class inheritance hierarchy of three classes "Computer", "Laptop", and "Desktop" in one database shown in Figure 3.4.

Example 3.1 (show s a tree structure of inheritance hierarchy): given an inheritance hierarchy (class and superclass relationships) for 3 classes "Computer", "Laptop", and "Desktop": (laptop, computer), (desktop, computer), show an HTree.

Solution 3.1: Figure 3.4 shows the Inheritance Hierarchy Tree, HTree, for 3 classes "Computer", "Laptop", and "Desktop".

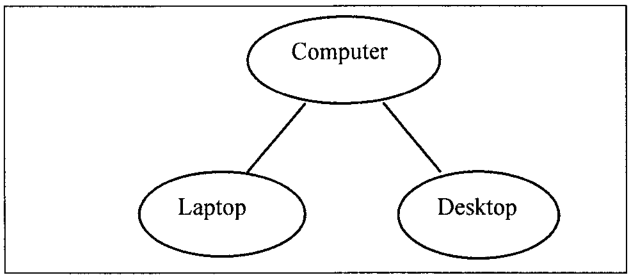

Figure 3.4 HTree for 3 class inheritance hierarhcy

The algorithm for creating the inheritance hierarchy tree (HTree) is given in Figure 3.5.

```
Algorithm CreateTree ($H$)

Input: inheritance hierarchy $H(class_i, super_i)$, subclass and superclass relationships,
sorted from higher hierarchy to lower hierarchy
Output: HTree that represents the class inheritance hierarchy in one database
Other variable: pointer nodePtr, // a node pointer variable points to the node of tree
                pointer Root // a node pointer variable points to the root of tree
Sub algorithm: CreateNode (nodePtr)// traverse existing part of the tree to find
matches and create new node

Begin

0.0 For each pair in $H(class_i, super_i)$ $i$ = 1,2...n do
      0.1 if ($i$ = 1)
          1.1.1 Create node and label it as $super_i$;
          1.1.2 Root points to node $super_i$
          1.1.3 Create node, label it as $class_i$, and set its parent as $super_i$;
       End if
       1.2 else
          1.2.1 nodePtr points to node Root
          1.2.2 CreateNode(nodePtr)
              1.2.2.1if (nodePtr != null)
                 1.2.2.1.1if ($super_i$ matches nodePtr->label)
                     1.2.2.1.1.1 Create new node, $class_i$, and set its parent as
                                 *nodePtr;
                     1.2.2.1.1.2 if (currentNode has a leftmost child)
                                    Set new node as right sibling of rightmost
                                    child of currentNode
                                 else
                                    Set new node as leftmost son of currentNode
                       End if
                 End if
                 1.2.2.1.2 CreateNode(nodePtr->left most child);
                 1.2.2.1.3 CreateNode(NodePtr ->right sibling);
              End if

End for
```

Figure 3.5 Algorithm for Creating Inheritance Hierarchy Tree

3.3.1.2 Pre-order Traversal and the Position Coding method

Pre-order traversal of a tree means recursively traverse the tree by first visiting (printing) the root node (N), followed by visiting the left subtree (L), and finally visiting the roght subtree (R).

***Example** 3.2* (pre-order traverse): Given a tree such as Figure 3.4, shows a process of pre-order traversal.

Solution 3.2: In Figure 3.4 node "Computer" is the root node. Node "Laptop" is the leftmost child of node "Computer", and the node "Desktop" is the right sibling of node "Laptop". The Pre-order traversal algorithm will visit node "Computer" first. Node "Computer" has a leftmost child, the node "Laptop". The algorithm then will visit the "Laptop" node. Node "Laptop" has no leftmost child. The algorithm will visit the right sibling child "Desktop".

The algorithm of pre-order traversal is show in Figure 3.6.

```
Algorithm PreOrderTraverse (root)

Input: root node of the tree T to be traversed.

Output: traversed tree T

Other variables: pointer nodePtr

Begin

      1.0 nodePtr = T.root;
      2.0 if (nodePtr != Null)
            2.1 PreOrderTraverse (nodePtr->leftmost child);
            2.2 PreOrderTraverse (nodePtr->right sibling);
       End if

End
```

Figure 3.6 Algorithm for pre-order traversal

In the *PLWAPLong* algorithm (Ezeife, saeed and Zhang(2009)), two position codes, start position and end position (tow integer numbers) are assigned to every node of the tree to distinguish the position of the nodes in the tree. Position codes are assigned by pre-order traversing the tree.

Example 3.2 (pre-order traverse): Given a tree such as Figure 3.4, each node of the tree is assigned two position codes, the start position and the end position by pre-order traversal.
Solution 3.2: In Figure 3.4 node "Computer" is the root node. Node "Laptop" is the leftmost child of the node "Computer", and the node "Desktop" is the right sibling of the node "Laptop". The Pre-order traversal algorithm will visit the "Computer" node first and will assign a start position for the node of "0". The algorithm then will visit the leftmost child node of the "Computer" node, node "Laptop", and will assign to the node a start position of "1". Node "Laptop" has no leftmost child and will be assigned an end position of "2". The algorithm will visit the right sibling child "Desktop" and it will assign a start position of "3" to the node "Desktop". Node "Desktop" has no leftmost child and it will be assigned an end position of "4". The algorithm will traverse back to assign an end position of "5" to the root node "Computer". Figure 3.7 shows the position codes assigned to the nodes of the tree.

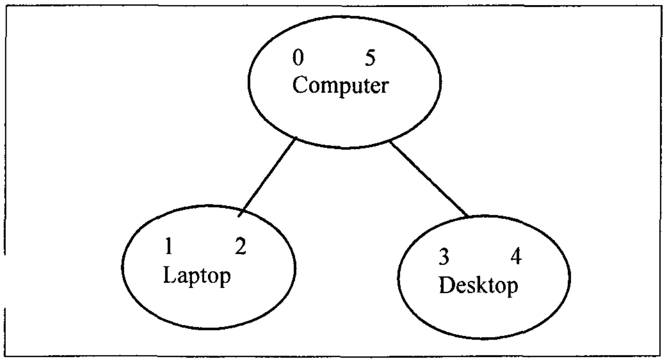

Figure 3.7 Position codes assigned Tree

The algorithm for assigning position codes is shown in Figure 3.8.

```
Algorithm AssignPosCode (T, root)

Input: tree to be assigned position codes T, root node of T.

Output: position codes assigned tree $T_p$

Other variables: pointer currentNode, posNumber initialized as 0, Boolean variable
forward initialized as True, backward initialized as False.

Begin
      1.0 root->startPosition = posNumber;
      2.0 currentNode = root->leftmost son;
      3.0 while(currentNode != root)
            3.1 posNumber incremented by 1;
            3.2 if (forward = True and backfward = False)
                  3.2.1if (currentNode->leftmost son != NULL)
                              3.2.1.1 currentNode->startPosition = posNumber;
                              3.2.1.2 currentNode = currentNode->leftmost son;
                  End if
                  3.2.2 else
                              3.2.2.1 currentNode->startPosition = posNumber;
                              3.2.2.2forward = false;
                              3.2.2.3 backward = true;
                   End else
            End if
            3.3else
                  3.3.1 if (forward = False and backward = True)
                        3.3.1.1 if(currentNode->right sibling == NULL)
                                    3.3.1.1.1 currentNode->endPosition = posNumber;
                                    3.3.1.1.2 currentNode = currentNode->parent;
                        End if
                         3.3.1.2 else
                                    3.3.1.2.1 currentNode->endPosition = posNumber;
                                    3.3.1.2.2 currentNode = currentNode->right sibling;
                                    3.3.1.2.3forward = True;
                                    3.3.1.2.4 backward = False;
                         End else
            End else
End
```

Figure 3.8 Algorithm for Assigning position codes

In this thesis, we borrowed the idea of assigning position codes and use it to represent the levels of inheritance hierarchy. This will be discussed in detail in section 3.3.1.4.

3.3.1.3 *map-gen join* of TidFP algorithm

One of the main techniques used in the TidFP algorithm is *map-gen join.* According to Ezeife and Zhang (2009), map-gen join is: For each pair of itemsets *M* and *P* $\in FC_k$ where each FC_k itemset has the two parts "< itemset, transaction id list >", the following three conditions have to be satisfied: *M* joins with *P* to get itemset $M \cup P$ if the following conditions are satisfied.

(a) itemset M comes before itemset P in FC_k,

(b) the first k-1 items in M and P (excluding just the last item) are the same,

(c) the transaction id list of the new itemset $M \cup P$ represented as $TidM{\cup}P$ is obtained as the intersection of the Tid lists of the two joined k-itemsets M and P and thus, $Tid_{M \cup P} = Tid_M \cap Tid_P$.

Example 3.3 (generate 2-itemset candidate patterns by *map-gen join*): Given a set of 1-itemset patterns with their transaction id lists. <1, 2, 3, *a*>, <1, 2, *b*>, <1, 2, *c*>, <3, 4, *d*>, generate 2-itemsets candidate patterns by *map-gen join.*

Solution 3.3: The part of itemsets are: *a, b, c, d.* Combinations can be generated as *ab*, *ac*, *ad*, *bc*, *cd.* For *ab*, intersect the transaction id list of *a* and transaction id list of *b* and this results in <1,2>. Therefore, the pattern is <1, 2, *ab*>. By the same method, we obtain all 2-itemset candidate patterns <1, 2, *ab* >, <1, 2, *ac* >, <3, *ad* >, <1, 2, *bc* >, <None, *bd* >,

<None, *cd* >.

Example 3.4 (generate 3-itemset candidate pattern by map-gen join): Given a set of 2-

itemset patterns with their transaction id lists. <1, 2, *ab* >, <1, 2, *ac* >, <3, *ad* >, <1, 2, *bc* >, generate 3-itemsets candidate patterns by *map-gen join.*

Solution 3.4: The itemset parts are *ab*, *ac*, *ad*, *bc*. *ab* will be checked with the other itemsets *ac*, *ad*, and *bc*. The first item of *ab* is *a*, the first items of *ac*, and *ad* is also *a*, therefore a 3-itemset will be generated such as *abc* and *abd*. For *abc*, intersect the transaction id list of *ab* and the transaction id list of *bc* to obtain <1, 2>. This results in the pattern <1, 2, *abc*>. Then *ac*, will be checked with *ad*, *bc* and *cd*. *ad* will be check with *bc*, and *cd*. *bc* will be checked with *cd*. The 3-itemset candidate patterns are: <1, 2, *abc* >, <None, *abd* >, <None, *acd* >, and <None, *bcd* >.

3.3.1.4 *oomap-gen join*

According to Ezeife and Zhang (2009), *map-gen join* avoids scanning a database multiple times. It provides support for *support counting* by intersecting the transaction ids. This is much more efficient than the Apriori algorithm. However, when the pattern is sparse, generating k-itemset candidate patterns is still time consuming. When the dataset is large and contains hundreds of thousands or millions of transactions, intersecting transaction ids of every k-itemset candidate pattern is time consuming.

In this thesis, we extend the *map-gen join* of the TidFP algorithm so that it avoids unnecessary candidate pattern generation along the inheritance hierarchy. The new algorithm avoids having to intersect the transaction ids of unnecessary candidate patterns. The modified version of the algorithm shall be called *oomap-gen join.*

In object-oriented frequent pattern mining, the patterns are in fact the attributes of the class objects. Example 1.3 shows computer object database. In that example the *Computer* class object has the attributes of, "CPU", "RAM", and "hard_drive". "Screen_size" and "battery_life" are attributes of the *Laptop* class object which inherits from the *Computer* class. "Graphic" is attribute of the *Desktop* class object which also inherits from the *Computer* class.

The 1-itemset candidate pattern such as <2GHz> is "CPU" attribute of the *Computer* class and it can be joined with <17"> which is "screen size" attribute of the *Laptop* class and <256M> is the "Graphic" attribute of the *Desktop* class. Examples of generated 2-itemset candidate patterns are <2GHz, 17"> and <2GHz, 256M>. However, we should not generate the candidate pattern <17", 256M>, because the patterns <17"> and <256M> will not appear at the same transaction after we joined the class tables. The *map-gen join* unfortunately would generate the candidate pattern <17", 256M>. Once the candidate pattern is generated the *map-gen join* algorithm will intersect the transaction ids of <17"> and <256M>, and thus determine that the intersection of these two id lists is "None". The new algorithm is more efficient because it would not have generated the candidate pattern <17", 256M> and thus can avoid having to perform the intersection operation. In section 3.3.1.2, we used two position codes, the *start* position and the *end* position to represent levels of inheritance hierarchy. We will use these two position codes to index the patterns (attributes of classes) to judge which class the patterns belong to, so that we can find out that if two k-itemset patterns should be joined to generate an (k+1)-itemset candidate pattern before joining them. Therefore, we need to define a new format for a pattern. As discussed in section 1.5, the TidFP algorithm proposed the pattern in the format of <Tidlist, itemset>, and called this the *TI* pattern. We can also extend the pattern to the format of <Tidlist, itemset>(start_position, end_position), which is the same as the *TI* pattern except with two position codes (*start* position and *end* position). This new form will be called the *TIP* pattern. An example of a *TIP* pattern is <1,2,3, "2GHz" "2GB">(0, 5). "1,2,3" is the transaction ids list which means the pattern appears in transaction 1, 2, and 3. "2GHz" "2GB" is itemsets. (0, 5) are position codes. "0" is the start position and "5" is the end position.

As discussed in section 3.3.1.2, the two position codes (*start* position and *end* position) can represent and account for the levels of a class inheritance hierarchy. As shown in Figure 3.6, each of the three classes (three nodes in the tree) is assigned a *start* position and *end* position. The class *Computer* is assigned "0" and "5" for the start and end position values. The class *Laptop* is assigned "1" and "2" for the *start* and *end* position values. The class *Computer* is assigned "3" and "4" for the *start* and *end* position values. The class *Computer* is the superclass of class *Laptop*. We can see that the *start* position of the class *Computer* is smaller than *start* position of the class *Laptop* and that the *end* position of the class *Computer* is greater than the *end* position of the class *Desktop*. There is no superclass or subclass relationship between the class *Laptop* and the class *Desktop*. We can see that the *start* position of the class *Laptop* is smaller than the *start* position of the class *Desktop*. We can also see that the end position of the class *Laptop* is smaller than the *end* position of the class *Desktop*.

As we discussed in the previous paragraph in this section, before joining two k-itemset patterns to generate an (k+1)-itemset pattern, we should check the level of inheritance hierarchy the patterns belong to. The two position codes (*start* and *end* position) can be used to check the level of hierarchy. When mining object-oriented data, patterns are class attributes. The *start* and *end* position coding scheme allows for patterns to be indexed according to the level of hierarchy they belong to. A set of rules is used to decide whether the patterns should be joined. This set of rules is defined as follows:

Rule 1: Given two patterns P and M, which are both k-itemset patterns, if the *start* position of P is equal to the *start* position of M and also the *end* position of P is equal to the *end* position of M, then P and M can be joined to generate a (k+1)-itemset candidate pattern. With reference to Example 1.3 (describing a computer object database) and Figure 3.6 (describing the HTree and position codes assigned to the nodes of the tree), the patterns <2GHz> and <4G> can be determined to both belong to the class *Computer*, since both will be index by the start position "0" and end position "5". Therefore, they can be joined to generate the candidate pattern <2GHz, 4G>.

Rule 2: Given two patterns P and M which are both k-itemset patterns, if the *start* position of P is smaller than the *start* position of M and also the *end* position of P is greater than the *end* position of M, then P and M can be joined to generate a (k+1)-itemset candidate pattern. With reference to Example 1.3 (describing a computer object database) and Figure 3.6 (describing the HTree and position codes assigned to the nodes of the tree), the pattern <2GHz> can belong to the class *Computer* or *Laptop* but the pattern <3hrs> exclusively belongs to the class Laptop. The Pattern <2GHz> will be indexed by a *start* position "0" and *end* position "5". Pattern <3hrs> is indexed by *start* position “1” and *end* position “2”. Therefore, they can be joined to generate a candidate pattern <2GHz, 2hrs>.

Rule 3: Given two patterns P and M which are both k-itemset patterns, if the *start* position of P is greater than the *start* position of M and also the *end* position of P is smaller than the *end* position of M, then P and M can be joined to generate a (k+1)-itemset candidate pattern. With reference to Example 1.3 (describing a computer object database) and Figure 3.6 (describing the HTree and position codes assigned to the nodes of the tree), the pattern <256M> belongs to the class *Desktop* and the pattern <3GHz> belongs to the class *Computer*. Pattern <256M> will be indexed by *start* position “3” and *end* position “4”. Pattern <3GHz> be indexed by *start* position “0” and *end* position “5”. Therefore, they can be joined to generate the candidate pattern <256M, 3GHz>.

As we discussed in section 3.3.1.3, *map-gen join* will check if the first k-1items in two k-itemset patterns are matched. Then it generates a (k+1)-itemset pattern by appending the last item of the second pattern to the end of first pattern, and it also intersects the transaction id lists of the two patterns. The difference between *map-gen join* and *oomap-gen join* is that the *oomap-gen join* will check if the two patterns satisfy any one of the rules described above (*Rule 1*, *Rule 2*, or *Rule 3*). If any of the rules are satisfied then we check for a match of k-1 items of two k-itemset patterns, and if they are matched then we generate a new (k+1)-itemset pattern. Once the new (k+1)-itemset pattern is generated, we intersect the transaction ids and append the position codes of second pattern to the newly generated (k+1)-itemset pattern. We need to append the position code of the second pattern to the newly generated pattern because the position codes need to be checked for generating a (k+2)-itemset candidate pattern. The algorithm of *oomap-gen* is given in Figure 3.9.

```
Algorithm oomap_gen_join ($F_k$)

Input: a set of k-itemset patterns $F_k$
Output: a set of (k+1)-itemset candidate patterns in format of
          <transaction id list, itemset>(start_position, end_position).

Other variables: any two k-itemset pattern P and M in $F_k$, in the format of
          <transaction id list, itemset>(start_position, end_position).
          (k+1)-itemset candidate pattern generated by P and M, in format of
          <transaction id list, itemset>(start_position, end_position).

Begin

  1.0 For every two k-itemset patterns P and M in $F_k$, P comes before M
     1.1if ((start_position of  P = start_position of  M and
        end_position of P = end_position of M)
        or (start_position of P < start_position of M and
        end_position of P > end_position of M)
        or (start_position of P < start_position of M and
        end_position of P < end_position of M))

      1.1.1if (the first k-1 itemsets in M and P are the same)
          1.1.1.1 transaction id list of N = transaction id lsit of P $\cap$ transaction id
                  List of M;
          1.1.1.2 itemset of N = itemset of P append last item of M;
          1.1.1.3start_position of N = start_position of M;
          1.1.1.4 end_position of N = end_position of M;
      End if
  End if

End
```

Figure 3.9 Algorithm for oomap-gen join

For example, if a pattern P has the format of <1,2,3,4, 2GHz, 2G>(0, 5), then we can determine from this format that the pattern appears in transaction 1,2,3,4 and belongs to the class *Computer* which has a *start* position "0" and an *end* position "5". Another pattern M may have the format of <1,2,3,5, 2GHz, 3hrs>(1, 2). From pattern M we can determine that the pattern appears in transaction 1,2,3,5 and belongs to the class *Laptop* which has *start* position "1" and *end* position "2". In step 1.0, *oomap-gen join* algorithm will check the *start* position and *end* position of these two patterns. The *start* position of P is smaller than the *start* position of M and the *end* position of P is greater than the *end* position of M. Then the *oomap-gen join* algorithm goes to step 1.1. Pattern P and M are 2-itemset patterns, and the first item of both patterns is "2GHz". The algorithm goes to step 1.1.1, where the transaction id lists of the two patterns will be intersected and to get the result <1,2,3>, which is the transaction id list. After the algorithm goes to step 1.1.2, the new itemset shall be <2GHz, 2G, 3hrs>. Finally, the algorithm goes to step 1.1.3 and 1.1.4. The *start* and *end* position of the new pattern is "1" and "2". Therefore, the final form of the new pattern is <1,2,3, 2GHz, 2G, 3hrs>(1,2).

3.3.1.5 Transaction IDs Stored MHTree (TMHTree)

As defined in ***Definition 1***(multiple databases inheritance hierarchy tree) in section 1.2, multiple databases inheritance hierarchy can be represented in a tree structure called ***MHTree***. When the method *MineRootFPs* mines *hierarchical frequent patterns* (HFPs) in the *Root* table, the transaction ids need to be stored in the nodes of ***MHTree*** to provide the information necessary to describe at which level of inheritance hierarchy the transaction appears.

Definition *6* (transaction ids stored multiple databases inheritance hierarchy tree): Transaction ids stored in the nodes of a multiple database inheritance hierarchy tree (*MHTree*). Due to the transaction ids being stored in the node we rename the modified ***MHTree*** to be called ***TMHTree***.

The example for an *MHTree* is shown in Figure 1.3 of section 1.2. That *MHTree* stores the transaction ids of a Root table (transaction table). The Root table was shown in Table 3.4 of section 3.3.2. The resulting ***TMHTree*** is shown in Figure 3.10.

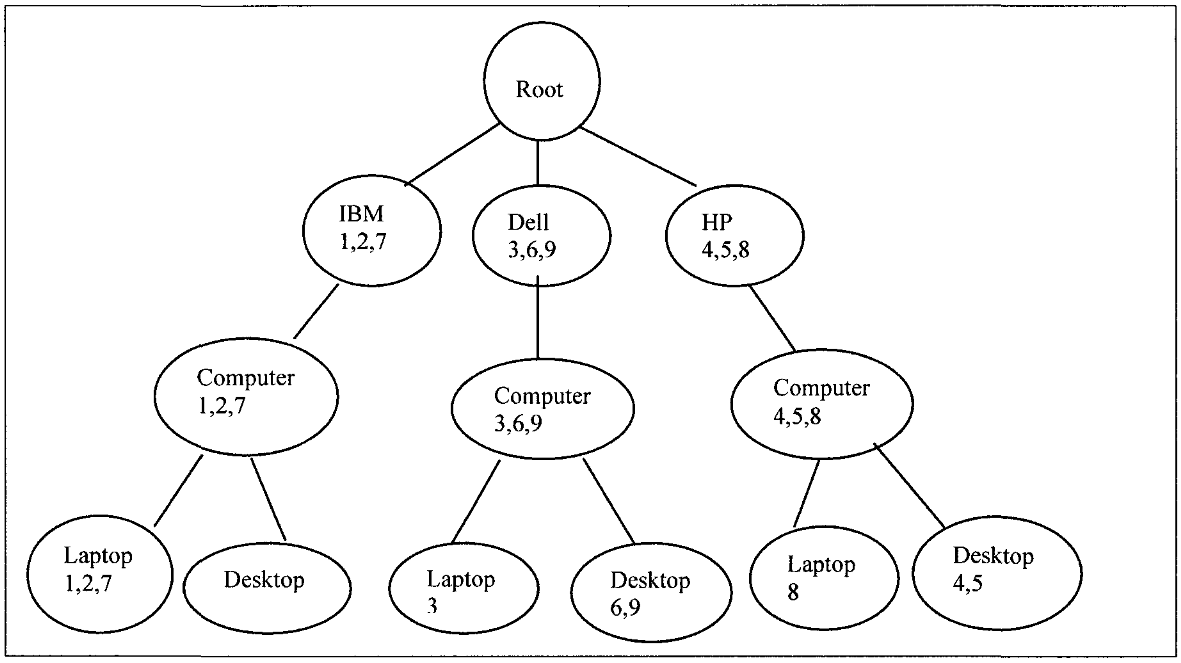

Figure 3.10 TMHtree transaction IDs stored MHTree

In Figure 3.10, integer numbers stored in the nodes of TMHTree are transaction ids.

3.3.1.6 Linkage Built TMHTree (LTMHTree)

In the PLWAP algorithm (Ezeife and Lu (2005)), the nodes in the tree with the same label can be connected by linkage and be accessed by a link header table.

Definition *7* (linkage built multiple databases inheritance hierarchy tree): after the linkage step the ***TMHTree*** is reffered to as the ***LTMHTree***. As an example, a TMHTree is shown in Figure 3.10 of section 3.3.1.5. Figure 3.11 provides an ***LTMHTree*** for three computer object databases, IBM, Dell, and HP with linkage built.

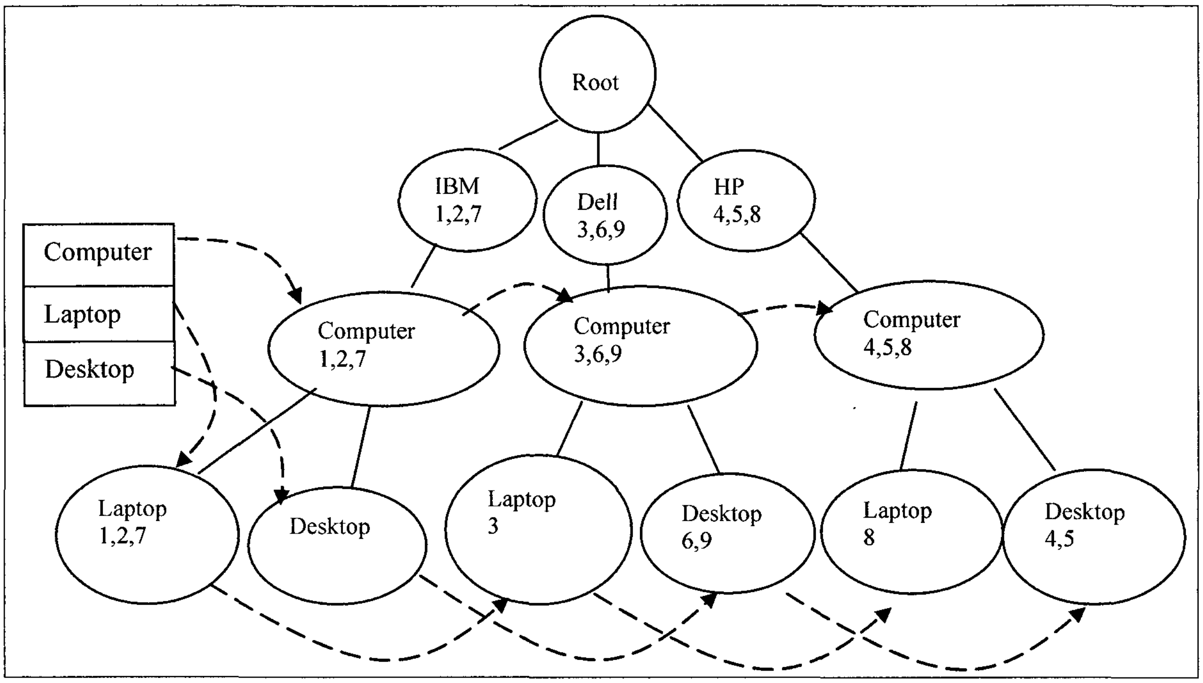

Figure 3.11 Linkage built LTMHTree (LTMHTree)

In Figure 3.11, the three rectangles which are labelled as “Computer”, “Laptop”, and “Desktop” represent the link header table. The link header table can access the nodes of the LTMHTree through the linkages. The detail of algorithm for building the linkage will be discussed in section 3.3.4.4.

3.3.2 Insert Transactions into *Root* table

As defined in section 1.2, the *Root* table is a transactional table which records the purchase transactions. When a purchase is made, a transaction needs to be inserted into the *Root* table. The main algorithm of the method *InsertTransactions* is shown in Figure 3.12.

```
Algorithm InsertTransactions ($H$, $C_t$, $Q$)

Input: Inheritance hierarchy $H$, //superclass and subclass relationship
        Class tables of purchased product $C_t$
        $CS_t$ //a set of superclass tables of $C_t$
        Purchase inquiry $Q$
Output: transaction inserted Root table
Other variables: Inheritance hierarchy Tree HTree, object joined table $T$

Begin

    1.0 CreateHTree($H$); //create inheritance hierarchy tree HTree

    2.0 AssignPosCode(HTree); //assign position codes to HTree

    3.0 IndexAttri(HTree, $C_t$); //index class attributes by two position codes

    4.0 JoinClasses($CS_t$, $C_t$); // join class tables and result a object joined table $T$
        4.1 $T = C_t$;
        4.2 if($CS_t \neq$ NULL) // C has super classes.
            4.2.1 For each superclass table $CS_t$
                    4.2.1.1 $T$ = OOJoin($CS_t$, $T$); // call OOJoin to join subclass and
                                                    superclass
            End for
        End if

    5.0 InsertTrans($Q$, $T$); // insert transaction into Root table

End
```

Figure 3.12 Algorithm for method *InsertTransactions*

Figure 3.13 is the process flow of the private method *InsertTransactions* belonging to the *Root* class.

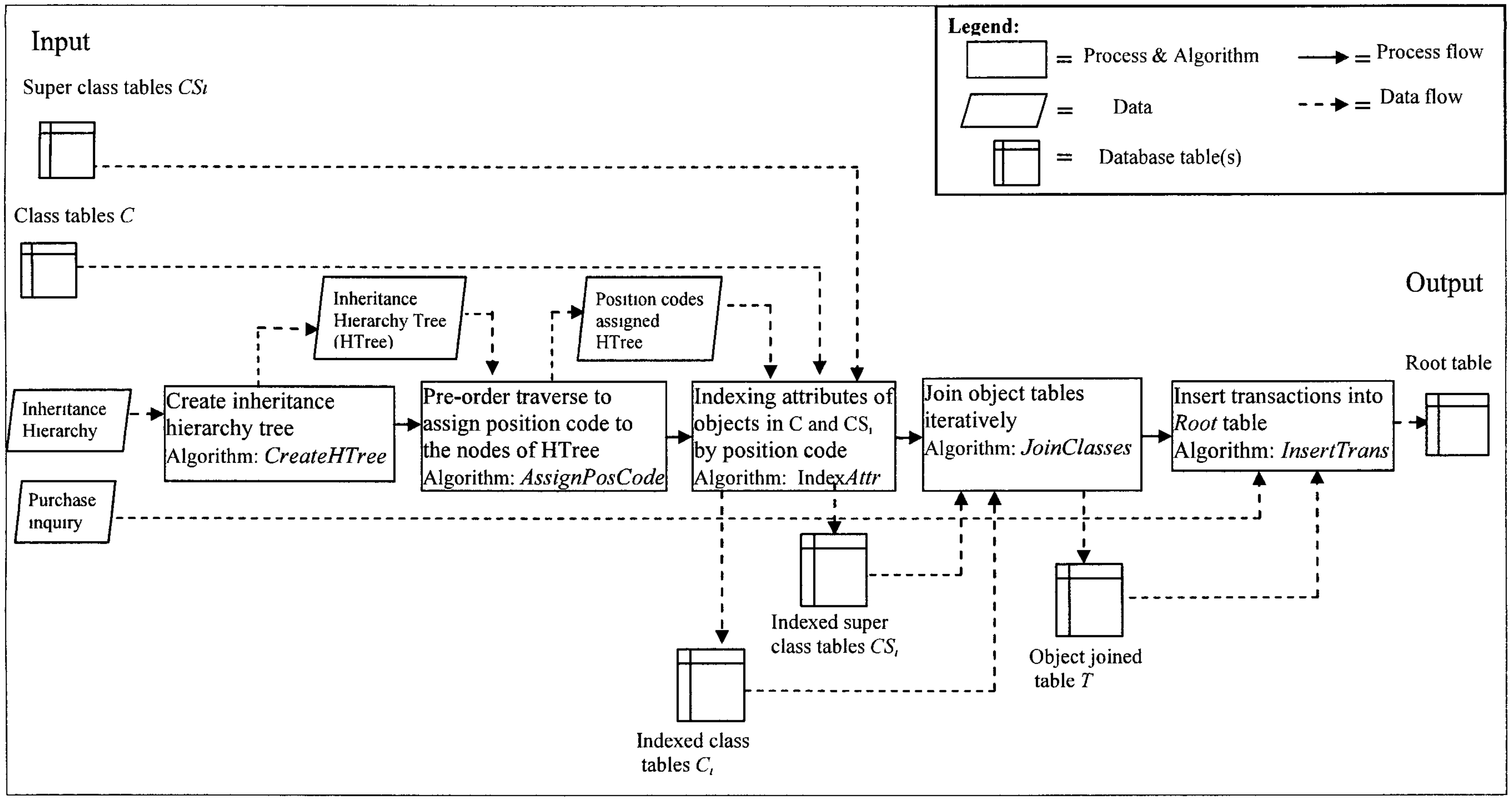
Input
Super class tables CS_i
Class tables C
Legend:
= Process & Algorithm
= Process flow
= Data
= Data flow
= Database table(s)
Output
Inheritance Hierarchy Tree (HTree)
Position codes assigned HTree
Inheritance Hierarchy
Create inheritance hierarchy tree Algorithm: *CreateHTree*
Pre-order traverse to assign position code to the nodes of HTree Algorithm: *AssignPosCode*
Indexing attributes of objects in C and CS_i by position code Algorithm: Index*Attr*
Join object tables iteratively Algorithm: *JoinClasses*
Insert transactions into *Root* table Algorithm: *InsertTrans*
Root table
Purchase inquiry
Indexed super class tables CS_i
Object joined table T
Indexed class tables C_i

Figure 3.13 Process Flow of private method *InsertTransactions* of Root Class

As shown in Figure 3.13, the private method *InsertTransactions* of the *Root* class consists of 5 steps:

1) Create the Inheritance hierarchy tree (HTree).

2) Use pre-order traversal to assign position codes to the nodes of the tree.

3) Index the attributes in the object database for each table.

4) Join the object table and its super class tables

5) Insert a transaction into *the Root* table.

For example, the private method *InsertTransactions* has the inputs: inheritance hierarchy *H*={(laptop, computer), (Desktop, Computer)}, a computer object database (refer to Example 1.3 of section 1.2), and a purchase inquiry of a laptop computer. The result of running the method is an updated Root table consisting of the new transaction.

Step1.0 of the algorithm will create an inheritance hierarchy tree (HTree), from H (as discussed in section 3.3.1.1). Step 2.0 will pre-order traverse the HTree to assign the position code values (as discussed in section 3.3.1.2). Step 3.0 will index the attributes of the computer object tables (Refer to Table1.1, Table 1.2 and Table 1.3 of section 1.2) by position codes in the HTree. The indexed object tables are shown in Table 3.4, Table 3.5, and Table 3.6.

Comp_id	Type	Super_type	CPU	RAM	Hard Drive
comp1	Laptop	Computer	2GHz (0, 5)	2G (0, 5)	250G (0, 5)
comp2	Laptop	Computer	2GHz (0, 5)	2G (0, 5)	320G (0, 5)
comp3	Laptop	Computer	3GHz (0, 5)	4G (0, 5)	350G (0, 5)
comp4	Desktop	Computer	3GHz (0, 5)	4G (0, 5)	500G (0, 5)
comp5	Desktop	Computer	3GHz (0, 5)	4G (0, 5)	500G (0, 5)
comp6	Desktop	Computer	3GHz (0, 5)	4G (0, 5)	500G (0, 5)

Table 3.4 Object Table of *Computer* Class in IBM DB

Lap_id	Type	Super_type	Screen_size	Battery_life
lapt1	Ideapad Laptop	Laptop (1, 2)	15" (1, 2)	3 hours (1, 2)
lapt2	Ideapad Laptop	Laptop (1, 2)	15" (1, 2)	3 hours (1, 2)
lapt3	Thinkpad Laptop	Laptop (1, 2)	17" (1, 2)	3.5 hours (1, 2)

Table 3.5 Objects table of Laptop class in IBM DB

Desk_id	Type	Super_type	Graphic
desk1	Work station	Desktop	256M (3, 4)
desk2	Work station	Desktop	256M (3, 4)
desk3	Desktop	Desktop	512M (3, 4)

Table 3.6 Objects table of Desktop class in IBM DB

Step 4.0 will apply *OOJoin* algorithm to join the superclass tables and subclass tables. The resulting table of the join is shown in Table 3.7

ID	Type	Super	CPU	RAM	Hard Drive	Comp Name	ID	Type	Super	Screen Size	Battery Life
comp1	Laptop	Computer	2GHz (0, 5)	2G (0, 5)	250G (0, 5)	I. laptop	lapt1	Ideapad Laptop	Laptop	15" (1, 2)	3hrs (1, 2)
comp2	Laptop	Computer	2GHz (0, 5)	2G (0, 5)	320G (0, 5)	I. laptop	lapt2	Ideapad Laptop	Laptop	15" (1, 2)	3hrs (1, 2)
comp3	Laptop	Computer	3GHz (0, 5)	4G (0, 5)	350G (0, 5)	T. laptop	lapt3	Thinkpad Laptop	Laptop	17" (1, 2)	3.5hrs (1, 2)

Table 3.7 Result table of OOJoin

Step 5.0 of the algorithm will insert a transaction into the *Root* table. As defined, the *Root* class primary key (transaction id) is an integer number. The class *Type* is the database name where the transaction came from. The class *Super_type* has the value of Root. The Root table also contains a set of attributes which are the attributes of the classes in each database and also contains the super types of these classes. The input "purchase inquiry" of the *InsertTrans* algorithm indicates the purchased object and which object database the purchase comes from.

The new purchase transaction is inserted into the Root table in such a form that attributes such as the transaction id, the Type (name of database where the transaction comes from), the *Super_type* (topmost object in the inheritance hierarchy), and $super_i$ of the object (the *Super_type* of the objects being joined), and also all the attributes of the joined classes.

The algorithm of *InsertTrans* is shown in Figure 3.14.

```
Algorithm InsertTrans(Q, T)

Input: purchase inquiry Q indicate the object purchased
       Joined object table T
Other variables: Root class table,
                 DBname // the name of the database the purchase comes from
                 Tid // transaction id, a sequence ID created by a DBMS
                 Super_i // super types of joined class tables
                 Attri_i // all attributes of joined class tables

Output: Root class table with inserted transaction.

Begin

    1.0 select purchased object from object joined table T;
    2.0 insert(Tid, DBname, "Root", Super_i, Attri_i) into Root.

End
```

Figure 3.14 Algorithm for InsertTrans

After calling the method *InsertTransactions* a few times, a sample Root table is shown in Table 3.8.

TID	Type	Super_type	$super_1$	$Super_2$	CPU	RAM	Hard Drive	Screen Size	Battery Life	Graphic
1	IBM	Root	computer	Laptop	2GHz (0, 5)	2G (0, 5)	250G (0, 5)	13" (1, 2)	3hrs (1, 2)	
2	IBM	Root	computer	Laptop	3GHz (0, 5)	4G (0, 5)	320G (0, 5)	14" (1, 2)	3hrs (1, 2)	
3	Dell	Root	computer	Laptop	2GHz (0, 5)	2G (0, 5)	350G (0, 5)	17" (1, 2)	3.5hrs (1, 2)	
4	HP	Root	computer	Desktop	3GHz (0, 5)	4G (0, 5)	500G (0, 5)			128M (3, 4)
5	HP	Root	computer	Desktop	3GHz (0, 5)	4G (0, 5)	500G (0, 5)			256M (3, 4)
6	Dell	Root	computer	Desktop	3GHz (0, 5)	4G (0, 5)	500G (0, 5)			512M (3, 4)
7	IBM	Root	computer	Laptop	2GHz (0, 5)	2G (0, 5)	300G (0, 5)	15" (1, 2)	3hrs (1, 2)	
8	HP	Root	computer	Laptop	3GHz (0, 5)	4G (0, 5)	160G (0, 5)	17" (1, 2)	3.5hrs (1, 2)	
9	Dell	Root	computer	Desktop	3GHz (0, 5)	4G (0, 5)	300G (0, 5)			256M (3, 4)

Table 3.8 indexed Root table

3.3.3 Mining Frequent Patterns in the *Root* Table

As discussed in section 1.4, directly applying one of the frequent pattern mining algorithms, such as the TidFP algorithm to mine the object attributes as frequent patterns of the Root class is not informative enough. An example of such a query is query 3 provided in section 1.4 (mining object attributes as frequent patterns for all computers purchased).

In order to discover the *Hierarchical Frequent Patterns* (HFPs) in the format of <Tidlist, itemsets, $class_i$>, queries like query 4 and query 5 of secion 1.4 (mining object attributes as frequent patterns while at the same time being able to indicate at which level the frequent patterns are significant at) requires a method such as *MineRootFPs*.

The main algorithm for the method *MineRootFPs* belonging to the *Root* class is given in Figure 3.15.

```
Algorithm MineRootFPs(MH, s%, Root)

Input: multiple database inheritance hierarchy MH, Root table, minimum support s%

Other variables: multiple database inheritance hierarchy Tree MHTree,
                 TMHTree, //Transaction ids stored MHTree
                 LTMHTree //Linkage built TMHTree,
                 set of k-itemset frequent pattern $F_k$;
                 set of k-itemset candidate pattern $C_k$;

Output: hierarchical frequent patterns HFPs in the format of
        <Tidlist, itemsets, $class_i$>.

Begin

    1.0 CreateMHTree(MH); //create multiple database inheritance hierarchy tree,
                                   MHTree
    2.1 StoreTidMHTree(MHTree, Root); //store transaction ids into MHTree and
                                            Obtain TMHtree
    2.2 GenOneCand(Root); //generate 1-itemset candidate patterns

    3.0 BuildLinkage(TMHTree); //build linkage of TMHTree and obtain
                                       TMHTree
    4.0 MineHFPs(LTMHTree, $C_k$, s%)
        4.1 $C_k$ = {1-itemset candidate patterns}
        4.2 $F_k$ = CheckMinS(MHTree, $C_k$, s%);
        4.3 if ($F_k$ is not empty)
            4.3.1 $C_{k+1}$ = oomap-gen-join($F_k$);
            4.3.2 k = k+1
            4.3.2 go to step 4.2

End
```

Figure 3.15 Algorithm for the method *MineRootFPs* of the *Root* class

Step 1.0 is creating a multiple database inheritance hierarchy tree (MHTree). Step 2.1 scans the entire table and stores the transa ction ids into the nodes of the MHTree. Concurently with Step 2.1, Step 2.2 generates the 1-itemset candidate patterns. Step 3.0

is building the linkage to MHTree, so that nodes of the MHTree can be easily accessed. Step 4.0 is mining for the *hierarchical frequent patterns* in the Root table. The detail of each step will be discussed in the Section 3.3.4.1, 3.3.4.2, 3.3.4.3, 3.3.4.4, and 3.3.4.5. The process flow of the private method *MineRootFPs* belonging to the *Root* class is shown in Figure 3.16.

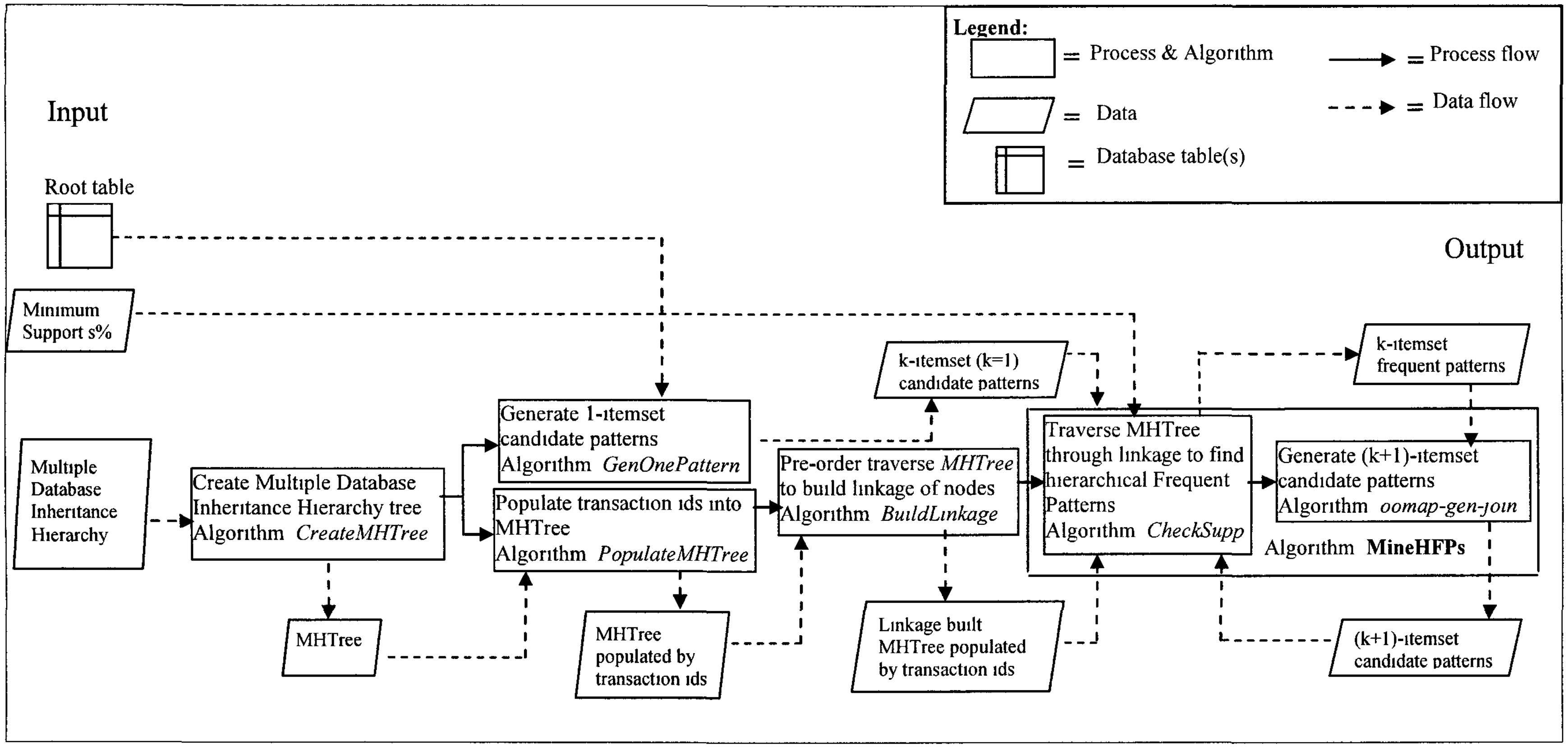
Legend:
= Process & Algorithm
= Process flow
= Data
= Data flow
= Database table(s)
Input
Output
Root table
Minimum Support s%
Multiple Database Inheritance Hierarchy
Create Multiple Database Inheritance Hierarchy tree Algorithm CreateMHTree
MHTree
Generate 1-itemset candidate patterns Algorithm GenOnePattern
Populate transaction ids into MHTree Algorithm PopulateMHTree
MHTree populated by transaction ids
k-itemset (k=1) candidate patterns
Pre-order traverse MHTree to build linkage of nodes Algorithm BuildLinkage
Linkage built MHTree populated by transaction ids
Traverse MHTree through linkage to find hierarchical Frequent Patterns Algorithm CheckSupp
k-itemset frequent patterns
Generate (k+1)-itemset candidate patterns Algorithm oomap-gen-join
Algorithm MineHFPs
(k+1)-itemset candidate patterns

Figure 3.16 Process Flow of private method *MineRootFPs* of Root Class

3.3.3.1 Create Multiple Databases Inheritance Hierarchy Tree

Step 1.0 of the method *MineRootFPs* belonging to the class *Root* is the algorithm of *CreateMHTree*. The Multiple databases Hierarchy Tree (***MHTree***) is retrieved from MH which is a set representing the relationship between each subclass and its respective superclass. The pairs from the MH set are sorted with respect to hierarchy level, from highest to lowest hierarchy. The algorithm will first create root node labelled as the "*super*" element of the first pair and the "*class*" element of the first pair is the leftmost child node of the root node. The element super in each pair from the set MH, will be first checked to see if it is already an established node in the MHTree that is being built. If the node does not yet exist, then the node is appended to the MHTree in its appropriate place with respect to its parent (hierarchy level). If the existing parent node to which the new node is about to be added does not have a child, the newly created node is set to be the leftmost child of the parent node. If the existing node has a leftmost child, then we find the rightmost child of the parent node and set the newly created node to be the new rightmost child. The algorithm of *CreateMHTree* is shown in Figure 3.17 and it is similar to the algorithm for creating the inheritance hierarchy tree in Figure 3.5.

```
Algorithm CreateMHTree (MH)

Input: multiple database class inheritance hierarchy MH(class_i, super_i), a set of
pairs, sorted from higher hierarchy to lower hierarchy
Output: MHTree that represents the class inheritance hierarchy in multiple
databases
Other variable: pointer nodePtr // a poniter variable points to the node of tree
Sub algorithm: CreateNode ( )// traverse existing part of the tree to find matches and
create new node

Begin

1.0 For each pair in MH(class_i, super_i) i = 1,2...n do
        1.1 if (i = 1)
            1.1.1 Create node and label it as Root;
            1.1.2 Create node, label it as class_i, and set its parent as Root;
        End if
        1.2 else
            1.2.3 nodePtr points to node Root
            1.2.4 CreateNode(nodePtr)
                1.2.4.1if (nodePtr != null)
                    1.2.4.1.1if (super_i matches nodePtr->label)
                        1.2.4.1.1.1 Create new node, class_i, and set its parent as
                                    *nodePtr;
                        1.2.4.1.1.2 if (currentNode has a leftmost child)
                                        Set new node as right sibling of rightmost
                                        son of currentNode
                                    else
                                        Set new node as leftmost son of currentNode
                        End if
                    End if
                    1.2.4.1.2 CreateNode(nodePtr->left most child);
                    1.2.4.1.3 CreateNode(NodePtr ->right sibling);
                End if
End for

End
```

Figure 3.17 Algorithm for creating multiple inheritance hierarchy tree (MHTree)

In Example 1.1, the ***MHTree*** will be obtained as Figure 3.18.

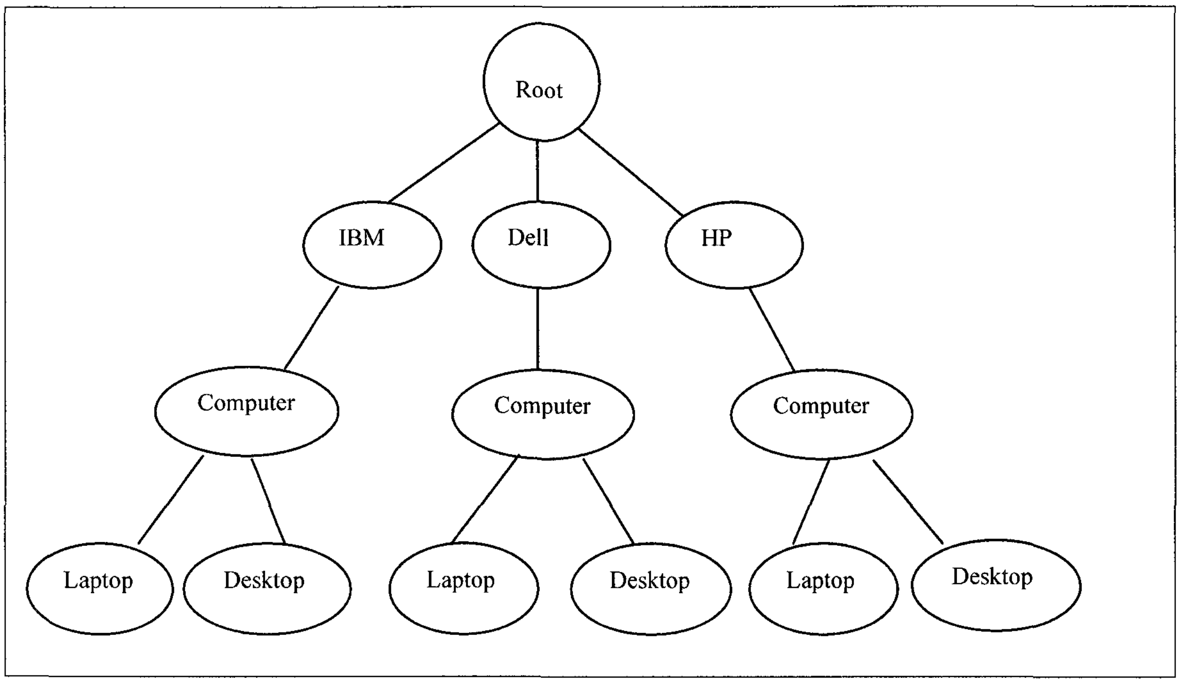

Figure 3.18 MHTree of computer object databases example

3.3.3.2 Store Transaction IDs into MHTree

Step 2.1 of the method *MineRootFPs* of the *Root* class is the algorithm of *StoreTidMHTree*. As discussed in section 1.4 and section 3.3.3, the Root table has the schema *Root(K, T, S, A, M, O)*. *T* is the *Type* of the *Root* class. When the Root table is a transaction table, *T* is the name of database where the transaction comes from. *A* is a set of attributes. *A* consists of super types (represented as $super_1$, $super_2$ ... $super_n$ in the *Root* table) and the physical attributes of all classes in the databases. In step 2.1, the algorithm scans the database table and extracts the attributes transaction ID, type (Database name), $super_1$, $super_2$ ... $super_n$ from the Root table. For every tuple in the Root table, the MHTree is traversed by the *StoreTidMHTree* algorithm. To insert the thransaction ids into the MHTree, we search the direct children of the root node for a match in the database name. Once the node is found with a matching database name, we insert the transaction id of the transaction record into the matching node. Then, we take each *super$_i$* that is found in that same transaction row, and we search the MHtree for the node representing that *super$_i$*. Once the node is found which represents the super element in the transaction record, the transaction id of that transaction record is inserted into that node.

The algorithm of PopulateMHTree() is shown in Figure 3.19.

```
Algorithm StoreTidMHTree(MHTree, Root)
Input: multiple databases inheritance hierarchy tree MHTree, Root table
Output: TMHTree//MHTree stored with transaction IDs
Other variables: pointer nodePtr //a node pointer
                 Tuple in Root table $t_i$

Begin

  1.0For each tuple $t_i$ in integrated Root table do
     1.1 nodePtr points to leftmost child node of Root node of MHTree;
     1.2 while(nodePtr != Null)
         1.2.1 if $t_i$.super_type matches nodePtr->label
               1.2.1.1 store $t_i$.transactionID in *nodePtr;
               1.2.1.2 break;
         End if
         1.2.2else
               1.2.2.1 nodePtr = nodePtr->right_slibling;
         End else
     End while
     1.3 nodePtr points to leftmost child of nodePtr;
     1.4 For each $t_i$.super$_j$ (j = 1, 2...n)
         1.4.1 nodePtr points to leftmost child of nodePtr;
         1.4.2 while(nodePtr != Null)
               1.4.2.1 if $t_i$.super matches nodePtr->label
                       1.4.2.1.1 store $t_i$.transactionID in *nodePtr;
                       1.4.2.1.2 break;
               End if
               1.4.2.2 else
                       1.4.2.2.1 nodePtr = nodePtr->right_slibling;
               End else
         End while

End
```

Figure 3.19 Algorithm of TraverseTree()

Table 3.4 (Root table) and Figure 3.18 (MHTree) show the sample input of the algorithm *StoreTidMHTree*. The first row has the attributes transaction id, database name, and $super_1 \ldots super_n$ As an example, the attributes of the first row of Table 3.4 would be <1, IBM, computer, laptop> and traversal of the MHTree will result in the MHTree being as shown in Figure 3.20. Transaction id is stored in node "IBM", "Computer", and "Laptop".

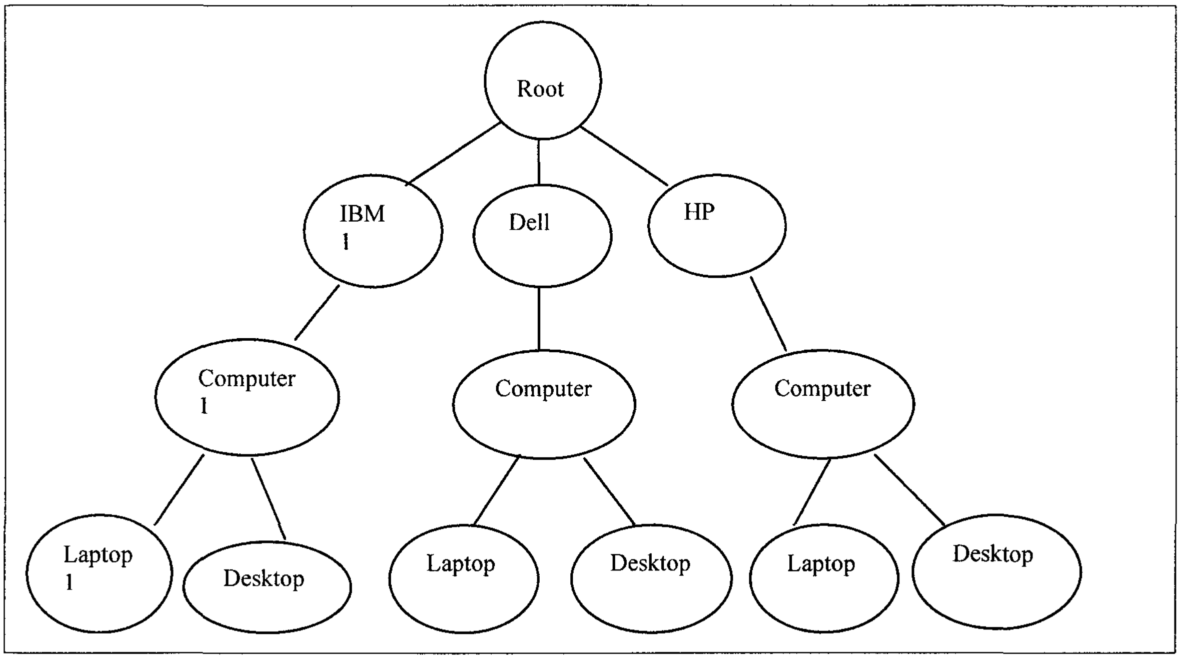

Figure 3.20 TMHTree store transaction ID 1

As an example, the attributes of the first row of Table 3.4 would be <2, IBM, computer, laptop> and traversal of the MHTree will result in the MHTree being as shown in Figure 3.21. Transaction id is stored in node "IBM", "Computer", and "Laptop".

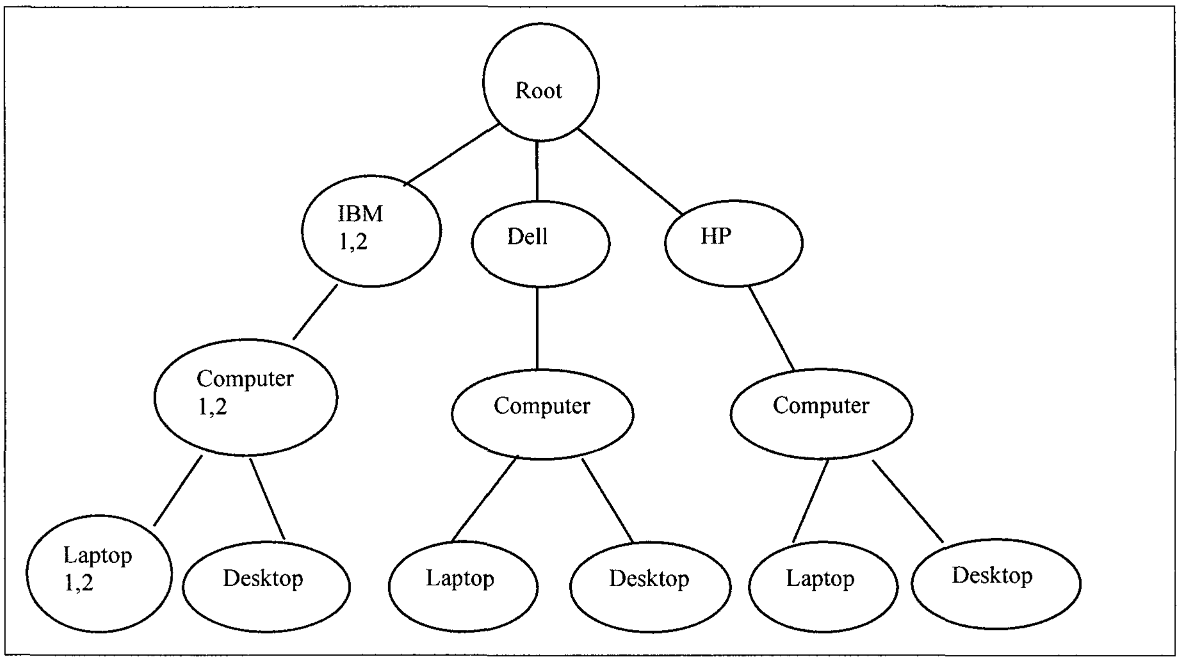

Figure 3.21 TMHTree store transaction ID 1, 2

After all the rows in Table 3.4 have been processed, the MHTree nodes will be populated with the appropriate transaction ids. The result obtained is shown in Figure 3.22.

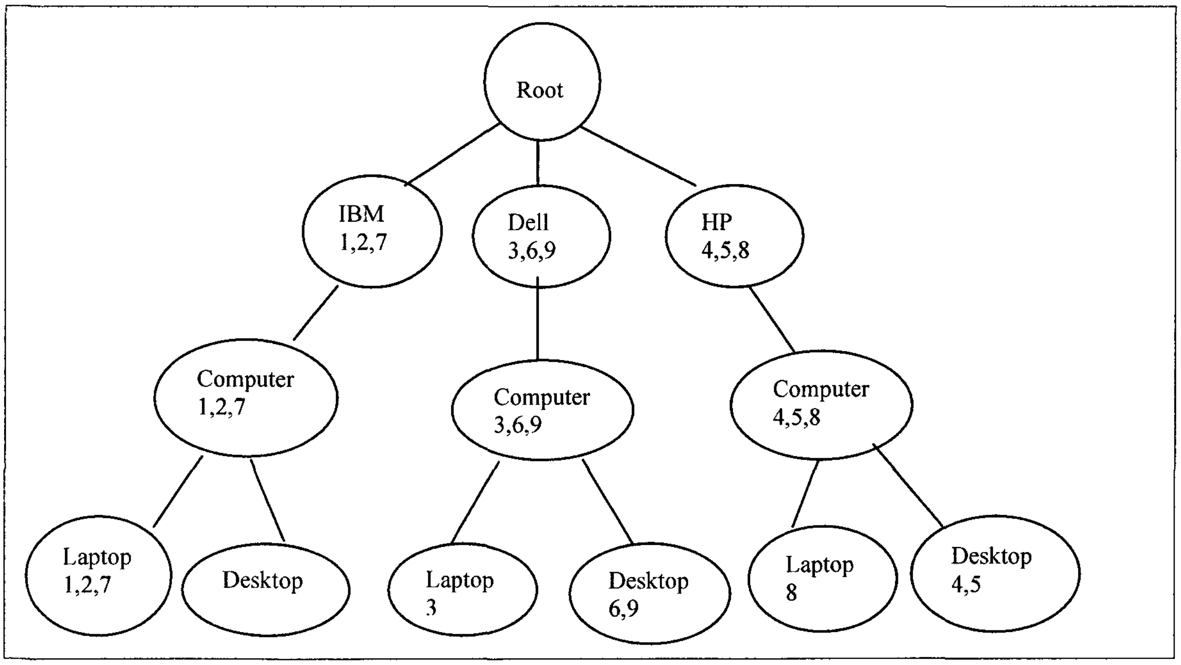

Figure 3.22 TMHtree stores all transaction IDs

3.3.3.3 Generate 1-itemset candidate pattern with transaction IDs

Step 2.2 of the method *MineRootFPs* is processed concurrently with Step 2.1 of the *MineRootFPs method.* While scanning the database in order to populate the MHTree nodes with the appropriate transaction ids, the other attributes of each record in the transaction table is also extracted and 1-itemset candidate patterns are generated. The generation of the 1-itemset candidate patterns is similar to the way that the *TidFP* algorithm generates 1-itemset candidate patterns. The algorithm for generating the 1-item candidate patterns with transaction ids is shown in Figure 3.23

```
Algorithm GenOneCand(Root)
Input: Root table.
Output: a set 1-itemset candidate patterns in the format of <TidList, itemset>.
Other variables: candSet, initialized as empty //a set stores 1-itemset candidate
patterns,
setPointer //a pointer points to the element of set of 1-itemset candidate patterns

Begin

 1.0 For Each tuple $t_i$ in the integrated Root table
     1.1For each attribute $a_{ij}$ in $t_i$
         1.1.1 setPointer points to the begin of candSet;
         1.1.2 while(setPointer != end of the canSet)
               1.1.2.1 if(setPointer->itemset = $a_{ij}$)
                       insert $t_i$.Tid into setPointer->TidList;
               end if
               1.1.2.2 setPointer points to next element of canSet;
         End while
         1.1.3 if($a_{ij}$ is not found in candSet)
                   1.1.3.1Create a new 1-itemset candidate pattern $c$;
                   1.1.3.2 insert $t_i$.Tid into $c$.TidList;
                   1.1.3.3 $c$.itemset = $a_{ij}$;
        End if
     End for
 End for

End
```

Figure 3.23 Algorithm for generating 1-item candidate patterns with transaction IDs

Table 3.4 (Root table) shows the sample input for the *genCandidate* algorithm. The variable *candSet* is initialized as empty. The first attribute of the first record in Table 3.4 is “2GHz”. *candSet* is empty and therefore does not contain the pattern “2GHz”. A 1-itemset candidate pattern <1, 2GHz> is generated. Insert transaction ids “1” in to TidList and itemset is “2GHz”. The candidate pattern <1, 2GHz> is placed into the *candSet* variable. Then the second attribute of first record is checked and a candidate pattern <1, 2G> is generated and placed into the *candSet* variable. The rest of attributes in the record will create 1-itemset candidate pattern <1, 250G>, <1, 13">, and <1, 3hrs>. After the first record has been processed, the *candSet* variable will contain the following set of candidate patterns, <1, 2GHz>, <1, 250G>, <1, 13">, and <1, 3hrs>.

The second record is processed in a similar way to how the first record was processed. The difference is that a candidate pattern having the itemset “3hrs” already exists and therefore there is no need to create a new candidate pattern having the itemset “3hrs”. Only the transaction id (“2”) of the second record needs to be added to the transaction id list of the candidate pattern <1, 3hrs> which then becomes <1,2, 3hrs>.

By the same process the 1-itemset candidate patterns are: <1, 3, 7, 2GHz>, <1, 3, 7, 2G>, <2, 4, 5, 6, 8,9, 3GHz>, <2, 4, 5, 6, 8, 9, 4G>, <1, 250G>, <1, 13">, <2, 14"> <2, 320G>, <3, 350G>, <7,9, 300G>,<8,160G>, <4, 5, 6, 500G>, <7, 15">,<3, 8, 17">, <1, 2, 7, 3hrs>, <3, 8, 3.5hrs >, <4, 128M>, <5, 9, 256M>, and <6, 512M>.

3.3.3.4 Build Linkage for Multiple Inheritance Hierarchy Tree

As discussed in section 3.3.1.6, the linkage can be used to access the nodes of the tree. Step 3.0 of the method *MineRootFPs* belonging to the *Root* class is using the

BuildLinkage algorithm to build the linkage to the nodes of the *MHTree* having the appropriate transaction ids. The *BuildLinkage* algorithm is given in Figure 3.24

```
Algorithm BuildLinkage(TMHTree)
Input: TMHTree //transaction ids stored MHTree.
Output: LTMHTree //linkage built TMHTree.
Other variables: linkage queues $q_t$ // to store nodes in an appropriate queue
                 Link header $t_t$
                 Link header table $T_t$
                 Root //the root node of TMHTree
                 $C_t$ //class tables

Begin

  1.0 create link header table $T_t$ according to class tables $C_t$, and each link header $t_t$
      is named as the name of each class table $C_t$;
  2.0 create linkage queue $q_t$ for each link header $t_t$;
  3.0 From Root, pre-order traversal of LTMHTree, visit Root, then left and then
      right subtrees and add all nodes to the appropriate linkage queue $q_t$, for every
      node, retrieve the hierarchy and store it in the node;

End
```

Figure 3.24 Algorithm for BuildLinkage

To serve as an example, the sample input of the BuildLinkage algorithm is the *TMHTree* (Figure 3.21). The step 1.0 of the algorithm is to build the link header table. In the computer object database, there are three object tables, "Computer", "Laptop", and "Desktop". The link header tables is shown bellow.

Computer
Laptop
Desktop

Step 2.0 is to build a *linkage queue* for each link header. Step 3.0 is to pre-order traverse the tree and to add all the nodes of the tree to the appropriate *linkage queue*. Firsth the

root is visited, then the “IBM” node is visited. There is no appropriate queue for node “IBM”. The left child of “IBM” is visited, which happens to be the “Computer” node. The “Computer” node is added to the “Computer” queue. Then the left child node of “Computer” is visited, this happens to be the “Laptop” node and the “Laptop” node is added to the “Laptop” queue. The “Laptop” node does not have a left child. The algorithm traverses backward. The “Laptop” node has the “Desktop” node as a sibling and the “Desktop” node is added to the “Desktop” queue. The “Desktop” node has no left child. The algorithm traverses backward. In this manner, the algorithm will build the *node linkage*. This step also retrieves the hierarchy of every node in the linkage queues. Hierarchy is retrieved by traversing backward along the parent node until the node is *Root*. The linkage built LTMHTree is shown in Figure 3.25.

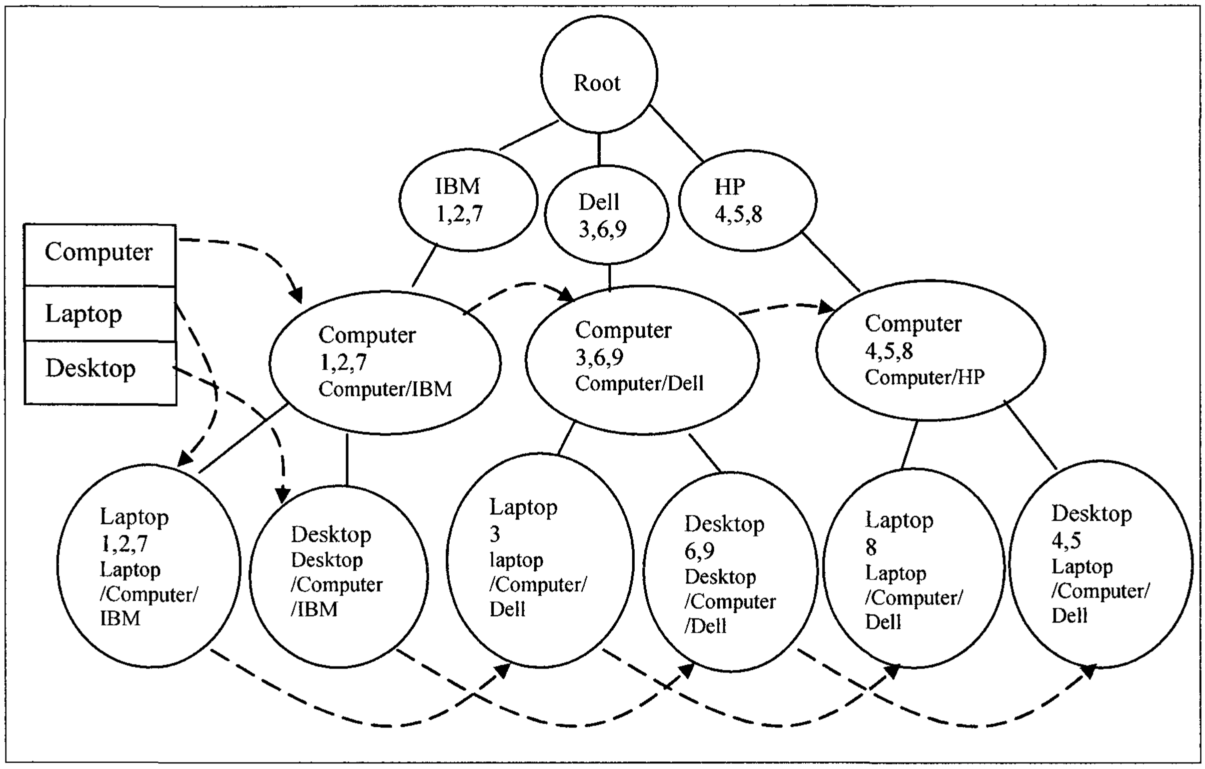
Root
IBM 1,2,7
Dell 3,6,9
HP 4,5,8
Computer
Laptop
Desktop
Computer 1,2,7 Computer/IBM
Computer 3,6,9 Computer/Dell
Computer 4,5,8 Computer/HP
Laptop 1,2,7 Laptop /Computer/ IBM
Desktop Desktop /Computer /IBM
Laptop 3 laptop /Computer/ Dell
Desktop 6,9 Desktop /Computer /Dell
Laptop 8 Laptop /Computer/ Dell
Desktop 4,5 Laptop /Computer/ Dell

Figure 3.25 Linkage built LTMHTree

3.3.3.5 Mining Hierarchical Frequent Patterns

The Step 4.0 of the method *MineRootFPs*, the *MineHFPs* algorithm finds the hierarchical frequent patterns in the table Root. The algorithm is shown in Figure 3.26.

```
Algorithm MineHFPs($LTMHTree$, $C_1$, $s\%$)

Input: linkage built and transaction ids stored multiple database inheritance
        hierarchy $LTMHTree$, minimum support $s\%$,
        a set of 1-itemset candidate pattern $C_1$, in the format of <Tidlist, itemset>

Output: a set of hierarchical frequent patterns $F_k$ in the format of
        <Tidlist, itemsets, $class_i$>

Other variable: a set of candidate pattern $C_k$

Begin

        1.0 $C_k = C_1$
        2.0 $F_k$ = CheckSupp($LTMHTree$, $C_k$, $s\%$);
        3.0 if ($F_k$ is not empty)
            3.1 $C_{k+1}$ = oomap-gen-join($F_k$);
            3.2 k = k+1
            3.3 go to step 2.0

End
```

Figure 3.26 Algorithm for MineHFPs

The *MineHFPs* algorithm takes as an input the *LTMHTree* (with transaction IDs stored and linkage built), a set of 1-itemset candidate patterns with transaction IDs, and a minimum support value *s%*. The algorithm *MineHFPs* will call the algorithm *CheckSupp* which uses every 1-itemset candidate pattern to traverse *LTMHTree* to check the support of each 1-itemset candidate pattern. If the support is greater than or equal to the minimum support of *s%* at any level in the hierarchy, then the 1-itemset candidate pattern counts as a 1-itemset frequent pattern. If 1-itemset frequent pattern(s) already exists, the *oomap-gen-join* algorithm is used to generate 2-itemset candidate patterns.

The *CheckSupp* algorithm is utilized to check the support level of the newly generated 2-itemset candidate patterns and to generate 2-itemset frequent pattern(s) if the support level is sufficient. If 2-itemset frequent patterns exist, use algorithm *oomap-gen-join* to generate 3-itemset candidate patterns. By the same process, k-itemset frequent patterns can be generated. The algorithm of CheckSupp is given in Figure 3.27.

```
Algorithm CheckSupp($LTMHTree$, $C_k$, $s\%$);
Input: MHTree, k-itemset candidate pattern with transaction IDs $C_k$, in the format of
        <Tidlist, itemsets, class$_l$>, k = 1 initially, minimum support $s\%$.
Output: Frequent k-itemsets $F_k$, in the format of <Tidlist, itemsets, $class_l$>.
Other variables: intersected transaction id list intersectTidlist, unioned Transaction
                 id list UTidlist, Pointer nodePtr, Frequent pattern $f$,
                 Boolean Flag=false, linkage queue of $LTMHTree$ $q_l$,
                 Hierarchy of every node $class_l$
Begin

1.0 For each element $c_x$ in $C_k$ do
        1.1 Flag = false;
        1.2 For each queue $q_l$ do
            1.2.1 For each element $e_{ly}$ in the queue $q_l$ do
                1.2.1.1 intersectTidlist = $c_x$.Tidlist ∩ $e_{ly}$.Tidlist;
                1.2.1.2 if((number of IDs in intersectTidlist)/(number of
                            IDs in $e_{ly}$.Tidlist) >= $s\%$)
                            1.2.1.2.1 $f = c_x$;
                            1.2.1.2.2 insert $f$ into $F_k$;
                            1.2.1.2.3 $f = f$ append $e_{ly}$. $class_l$;
                            1.2.1.2.4 Flag = true;
                1.2.1.3 UTidlist = UTidlist ∪ $e_{ly}$.Tidlist;
            1.2.2 intersectTidlist = $c_x$.Tidlist ∩ UTidlist;
            1.2.3 if((number of IDs in intersectTidlist)/(number of
                            IDs in UTidlist) >= $s\%$)
                            1.2.3.1 insert $f$ into $F_k$;
                            1.2.3.2 $f = c_x$ concatenate $e_{ly}$. $class_l$;
                            1.2.3.3 Flag = true;
                        End if
                    End for
                End for
        1.3 if(Flag = true)
                4.3.1 Insert $c_x$ into $F_k$;
            End if
    End for

End
```

Figure 3.27 Algorithm for CheckSupp

To serve as an example, the *MineHFPs* algorithm uses the input *LTMHTree* (Figure 3.24), 1-itemset candidate patterns: <1, 3, 7, 2GHz>, <1, 3, 7, 2G>, <2, 4, 5, 6, 8,9, 3GHz>, <2, 4, 5, 6, 8, 9, 4G>, <1, 250G>, <1, 13">, <2, 14"> <2, 320G>, <3, 350G>,

<7,9, 300G>,<8,160G>, <4, 5, 6, 500G>, <7, 15">,<3, 8, 17">, <1, 2, 7, 3hrs>, <3, 8, 3.5hrs >, <4, 128M>, <5, 9, 256M>, and <6, 512M>, and a minimum support value of 50%. The *MineHFPs* algorithm will output a set of hierarchical frequent patterns in the format of <Tidlist, itemsets, $class_i$>.

Step 1.0 and 2.0 of the *MineHFPs* algorithm use the transaction id list (Tidlist) of every 1-itemset candidate pattern to intersect the Tidlist of every node in every linkage queue of the LTMHTree in order to discover the 1-itemset frequent patterns. The *MineHFPs* algorithm starts from the first 1-itemset candidate pattern <1, 3, 7, 2GHz>. The Tidlist of the candidate pattern <1, 3, 7, 2GHz> is <1, 3, 7>. The first node of linkage queue of "Computer <1, 2, 7>" is <1, 2, 7> (according to Figure 3.13). Intersecting <1, 3, 7, 2> and <1, 2, 7> obtains <1, 7>. There are two transaction ids in <1, 7>. The number of ids in <1, 2, 7> is 3. The frequency is 2/3 which is greater than 50%. Hierarchy of node "Computer <1, 2, 7>" is node "computer/IBM". Therefore, we obtain the hierarchical frequent pattern <1, 7, 2GHz, computer/IBM>. We also insert the candidate pattern <1, 3, 7, 2GHz> into frequent pattern set F_1. In the same way of processing the candidate pattern <1, 3, 7, 2GHz> and node "Computer <3,6,9>" is intersected, and pattern <1, 3, 7, 2GHz> and node "Computer <4,5,8>"is intersected. We find out that pattern <1, 3, 7, 2GHz> is not frequent at node "Computer <3,6,9>" nor at node "Computer <4,5,8>". We also need to union the Tidlists of all three nodes in the "Computer" linkage queue. Union of Tidlists <1,2,7>, <3,6,9>, and <4,5,8> is <1, 2, 7, 3, 6, 9, 4, 5, 8>. Intersecting Tidlist of pattern <1, 3, 7, 2, 2GHz> and <1, 2, 7, 3, 6, 9, 4, 5, 8> is <1, 3, 7, 2>. The frequency is 4/9 which is less than the minimum support of 50%. Therefore the pattern <1, 3, 7, 2,

2GHz> is not frequent at the hierarchy “Computer”. The Tidlist of candidate pattern <1, 3, 7, 2, 2GHz> will intersect Tidlist of nodes in “Laptop” linkage queue and “Desktop” linkage queue. The rest of 1-itemset candidate patterns in the 1-itemset candidate pattern set C_1, will be processed by the same procedure as above to discover the frequent 1-itemset patterns. The complete set of 1-itemset frequent patterns is: <1, 7, 2GHz, computer/IBM>,

<1, 2, 7, 2GHz, laptop/computer/IBM>, <1, 7, 2G, computer/IBM>,

<1, 2, 2G, laptop/computer/IBM>, <6, 9, 3GHz, computer/Dell>,

<6, 9, 3GHz, desktop/computer/Dell>, <8, 3GHz, laptop/computer/HP>,

<6, 3GHz, desktop/computer/HP>, <2, 4, 5, 6, 8, 3GHz, computer>,

<4, 5, 6, 8, 9, 3GHz, desktop>, <6, 9, 4G, computer/Dell>,

<6, 9, 4G, desktop/computer/Dell>, <8, 4G, laptop/computer/HP>,

<6, 4G, desktop/computer/HP>, <2, 4, 5, 6, 8, 4G, computer>,

<4, 5, 6, 8, 9, 4G, desktop>,<1, 2, 7, 3hrs, computer/IBM>,

<1, 2, 7, 3hrs, laptop/computer/IBM>, <3, 350G, laptop/computer/Dell>,

<3, 17", laptop/computer/Dell>, <8, 17", laptop/computer/HP>,

<3, 3.5hrs, laptop/computer/Dell>, <8, 17", laptop/computer/HP>,

<6, 500G, desktop/computer/Dell>, <4, 5, 500G, desktop/computer/HP>,

<9, 300G, desktop/computer/dell>, <8, 160G, laptop/computer/dell>,

<4, 128M, desktop/computer/HP>, <9, 256M, desktop/computer/Dell>,

<5, 256M, desktop/computer/HP>, and <6, 512M, desktop/computer/Dell>.

The frequent 1-itemset patterns set F_1 contains these patterns: <1, 3, 7, 2GHz>, <1, 3, 7, 2G>, <2, 4, 5, 6,8, 9, 3GHz>, <2, 4, 5, 6,8, 9, 4G>, <1,2, 7, 3hrs>, <2, 350G>, <3, 8, 17">, <3, 8, 3.5hrs>, <4, 5, 6, 500G>, <7, 9, 300G>, <8, 160G>, <4, 128M>, <5, 9, 256M>, and <6, 512M>. *oomap-gen join* will be applied on these frequent 1-itemset patterns to generate candidate 2-itemset patterns. We have indexed patterns by assigned position codes. Therefore, the frequent 1-itemset patterns with position codes in the set F_1 are: <1, 3, 7, 2GHz>(0, 5), <1, 3, 7, 2G>(0, 5), <2, 4, 5, 6, 8, 9, 3GHz>(0, 5), <2, 4, 5, 6, 8, 9, 4G>(0, 5), <1, 2, 7, 3hrs>(1, 2), <2, 350G>(0, 5), <3, 8, 17">(1, 2), <3, 8, 3.5hrs>(1, 2), <4, 5, 6, 500G>(0, 5), <7, 9, 300G>(0, 5), <8, 160G>(0, 5), <4, 128M>(3, 4), <5,9, 256M>(3, 4), <6, 512M>(3, 4). There exist 1-itemset frequent patterns, so the algorithm will go to step 3.1 and will apply the *oomap-gen join* algorithm on these 1-itemset patterns to generate 2-itemset candidate patterns. The result of the *oomap-gen join* algorithm is: <1, 3, 7, 2GHz, 2G>(0, 5), <None, 2GHz, 3GHz>(0, 5), <None, 2GHz, 4G>(0, 5), <1, 7, 2GHz, 3hrs>(1, 2), <None, 2GHz, 350G>(0, 5), <3, 2GHz, 17">(1, 2), <3, 2GHz, 3.5hrs>(1, 2), <None, 2Ghz, 500G>(0, 5), <7, 2GHz, 300G>(0, 5), <None, 2GHz, 160G>(0, 5), <None, 2GHz, 128M>(3, 4), <None, 2GHz, 256M>, <None, 2G, 3GHz>(0, 5), <None, 2G, 4G>(0, 5), <1,7, 2G, 3hrs>(1, 2), <None, 2G, 350G>(0, 5), <3, 2G, 17">(1, 2), <3, 2G, 3.5hrs>(1, 2), <None, 2G, 500G>(0, 5), <7, 2G, 300G>(0, 5), <None, 2G, 160G>(0, 5), <None, 2G, 128M>(3, 4), <None, 2G, 256M>(3, 4), <None, 2G, 512M>(3, 4), <2, 4, 5, 6, 8, 9, 3GHz,4G >(0, 5), <2, 3GHz, 3hrs>(1, 2), <2, 3GHz, 350G>, <8, 3GHz, 17">(1, 2), <8, 3GHz, 3.5hrs>(1, 2),

<4, 5, 6, 3GHz, 500G>(0, 5), <9, 3GHz, 300G>(0, 5), <8, 3GHz, 160G>(0, 5),

<4, 3GHz, 128M>(3, 4), <5, 9, 3GHz, 256M>(3, 4), <6, 3GHz, 512M>(3, 4),

<2, 4G, 3.5hrs>(1, 2), <4, 5, 6, 4G, 500G>(0, 5), <None, 4G, 300G>(0, 5),

<8, 4G, 160G>(0, 5), <4, 4G, 128M >(3, 4), <5, 9, 4G, 256M>(3, 4),

<6, 4G, 512M>(3, 4), <2, 3hrs, 350G>(0, 5), <None, 3hrs, 17">(1, 2),

<None, 3hrs, 3.5hrs>(1, 2), <None, 3hrs, 500G>(3, 4), <7, 3hrs, 300G>(0, 5),

<None, 3hrs, 160G>(0, 5), <None, 350G, 17">(1, 2), <None, ,350G, 3.5hrs>(1, 2),

<None, 350G, 500G>(0, 5), <None, 350G, 300G>(0, 5), <None, 350G, 160G>(0, 5),

<None, 350G, 128M>(3, 4), <None, 350G, 256M>(3, 4), <None, 350G, 512M>(3, 4),

<3, 8, 17", 3.5hrs>(1, 2), <None, 17", 500G>(0, 5), <None, 17", 300G>(0, 5),

<8, 17", 160G>(0, 5), <None, 3.5hrs 500G>(0, 5), <None, 3.5hrs, 300G>(0, 5),

<8, 3.5hrs, 160G>(0, 5), <None, 500G, 300G>(0, 5), <None, 500G, 160G>(0, 5),

<4, 500G, 128M>(3, 4), <5, 500G, 256M>(3, 4), <6, 500G, 512M>(3, 4)),

<None, 300G, 160G>(0, 5), <None, 300G, 128M>(3, 4), <9, 300G, 256M>(3, 4),

<None, 300G, 512M>(3, 4), <None, 160G, 128M>(3, 4), < None, 160G, 256M>(3, 4),

< None, 160G, 512M>(3, 4), <None, 128M, 256M>(3, 4), <None, 128M, 512M>(3, 4),

<None, 256M, 512M>(3, 4). These 2-itemset candidate patterns will serve as inputs to the *CheckSupp* algorithm and 2-itemset frequent patterns are generated. Then the 2-itemset frequent patterns will be used to generate 3-itemset candidate patterns by *oomap-gen join*. By the same process we obtain all k-itemsets hierarchical frequent patterns, until there are no frequent patterns generated.

4. IMPLEMENTATION AND EXPERIMENTS

To test the performance of our proposed method of mining hierarchical frequent patterns in table Root (transaction table) , we use the *IBM quest synthetic data generator* to generate datasets. The *IBM quest synthetic data generator* is publicly available at http://www.almaden.ibm.com/cs/quest/and is used by other pattern mining researchers. The proposed algorithm *MineHFPs* in section 3.5.3.4 will be compared with the *TidFP* algorithm with respect to CPU execution time and memory usage.

4.1 Generate and Process Testing Dataset

The characteristics of the data generated by *IBM quest synthetic data generator* are described as follows: $|D|$: Number of transactions in the database, $|C|$: Average length of the transactions, $|S|$: Average length of maximal potentially frequent itemset, $|N|$: number of items (attributes). The *IBM quest synthetic data generator* generates a set of transactions to serve as the dataset. Every transaction has a set of patterns presented by integers. The integers that represent patterns are from 0 to $|N|$.

4.1.1 Generate the Class Tables C_i

In Example 1.3 in section 1.2 (computer objects database), there are three databases, IBM, Dell, and HP. Every database has three class tables, "Computer", "Laptop", and "Desktop". The *IBM quest synthetic data generator* is used to generate three datasets for the three databases. There are three datasets (class object table C_i) in every database, the first one represents the "Computer" objects table, the second for the "Laptop" objects table, and the third for the "Desktop" objects table. The dataset generated by the data generator are set of transactions in the format of <transaction id, number of items, a set of items> .

The *IBM quest synthetic data generator* generates integer numbers to represent patterns (attributes of objects in the case of object-oriented databases). If we specify the number of items, $|N|$, as "15", it means that the patterns will be represented by the integer numbers from "1" to "15". When we use the *IBM quest synthetic data generator* to generate the dataset which represents the *Computer* class table, we specify $|N|$ as "15". This means that the integer numbers from "1" to "15" will represent the patterns of the *Computer* class table. When we generate the dataset for the *Laptop* class table, we specify $|N|$ as "60". However, the integer numbers from "1" to "15" have already been used to represent the patterns of the *Computer* class table. We need to eliminate the numbers "1" to "15" so that the dataset generated will only contain the integer numbers from "16" to "60". Therefore, the integer number from "16" to "60" will be used to represent patterns of the *Laptop* class table. When we generate the dataset for the *Desktop* class table, we specify $|N|$ as "120". Since the numbers from "1" to "15" has already been used to represent the patterns of the *Computer* class table and the integer numbers from "15" to "60" has already been used to represent the patterns of the *Laptop* class table, we need to eliminate the numbers from "1" to "60" so that the dataset generated will only contain the integer numbers from "60" to "120". Therefore, the integer number from "60" to "120" will be used to represent patterns of the *Desktop* class table.

Each transaction of the dataset represents one instantiated object. The transaction id of a transaction record will represent the object id of one instantiated object and a set of items in a transaction record will represent a set of object attributes in one instantiated object. We generate three datasets (Computer, Laptop, Desktop) for each of the three databases (IBM, Dell, and HP). We use an integer number to represent a particular database (the database name) and an integer number to represent a particular class object table. For example, "1" represents the "IBM" database, "2" represents the "Dell" database, "3" represents the "HP" database, "4" represents the "Computer" class, "5" represents the "Laptop" class, and "6" represents the "Desktop" class.

The "Computer" class is inherited by the "Laptop" and the "Desktop" class. As discussed in section 1.2, the database schema of C_t is $C_t(K, T, S, A, M, O)$. T is the type and S is the super type. The dataset that stands for the "Computer" class will be assigned a number "4" as S *(super_type),* and randomly assigned "5" or "6" as T (*type).* With regards to the "Laptop" class, *S(super_type)* will be assigned as "5", and T (*type)* will be randomly assigned as "5","7" or "8" (which represent different subclasses of the "Laptop" class). With regards to the dataset that stands for the "Desktop" class, *S(super_type)* will be assigned as the number "6", and T (*type)* will be randomly assigned as "6","9" or "10" (which represent different subclasses of the "Desktop" class)..

4.1.2 Generate the *Root* Tables

In Example 1.3 in section 1.2, the ***Root*** table has the database schema *Root* (K, T, S, A, M, O). As we discussed in section 1.2, the ***Root*** table is a transaction table that has transaction id K as a primary key, T and S as foreign keys (which represent type and super type of the transactions in *Root* table). K is the transaction id which is an integer number from 1-|*D*| sequentially. As discussed in the beginning of this section, |*D*| is number of transactions in the database. We will generate a number of transaction ids (depending on the size of dataset we want to test). Type, *T*, is used to represent the name of the database where the transactions come from. We randomly generate an integer number among "1", "2", "3" for type, *T*, for every transaction to represent the name of a databases (such as IBM, Dell, and HP). Then we apply *OOJoin* algorithm to join all class tables C_i in every database to obtain an object joined table. Finally, randomly select the objects from object joined table in each database to fill in the attributes *A* in the Root Table.

4.2 Algorithm Implementation

The algorithm of *MineHFPs* and *TidFP* are both implemented in C++ with the same data structures and can run on both windows and UNIX platforms. In a UNIX environment, the programs are compiled with "g++ filename" and executed with "a.out". The class object table $\boldsymbol{C_i}$, inheritance hierarchy *H*, and multiple database inheritance hierarchy *MH* are all stored in text file.

4.3 Performance Comparison

Our proposed algorithm *MineHFPs* can be applied on the integrated ***Root*** table to answer all the Root table mining queries from section 1.4. If we separate the integrated ***Root*** table by class hierarchy, the TidFP algorithm can also be applied to each separated part to answer those queries. For example, using the TidFP algorithm to answer "*Query 4: What are the most popular hardware component specifications (CPU, RAM, Hard_drive,* *screen size, and battery life) among a computer system* ***subgroup*** *such as laptops and sold by a* ***particular company*** *like Dell (with a minimum support of 50%)?*", we will select transactions having type as "3" (transaction comes from Dell database), and also has super$_1$ "4" and super$_2$ "5" to represent "Computer" and "Laptop", respectively.

In this section, we will compare the performance of our proposed algorithm *MineHFPs* and the TidFP algorithm. Both the CPU execution time and the memeory usage is measured for each algorithm. The *MineHFPs* algorithm performance measures include the tasks of creating the *MHTree*, storing transaction ids in the MHTree, generating 1-itemset candidate patterns, building linkage, and executing the *MineHFPs* algorithm. We generate the ***Root*** tables of size 125K, 250K, 500K, and 1M. The characteristics of the generated dataset are described in Table 4.1

Root table	Computer class	Laptop class	Desktop class
100K	C7.S4.N20.D100K	C15.S4.N60.D50K	C25.S4.N120.D50K
250K	C7.S54.N20.D250K	C15.S4.N60.D125K	C25.S4.N20.D125K
500K	C7.S4.N20.D500K	C15.S4.N60.D250K	C25.S4.N20.D250K
1000K	C7.S4.N20.D1000K	C15.S4.N60.D500K	C25.S4.N20.D500K

Table 4.1 characteristics of the generated dataset

Table 4.2 and Figure 4.1 describes the execution time of the MineHFPs and the TidFP algorithm on **125K** dataset with low minimum support (20%, 10%, 9%, 8%, and 7%).

Algorithms	Runtime (in Seconds) at different supports(%)				
	20%	10%	9%	8%	7%
MineHFPs	290	4186	6356	9606	17785
TidFP	279	12327	23083	40097	74046

Table 4.2 CPU executing time on 125K dataset with minimum support 20%, 10%, 9%, 8%, and 7%

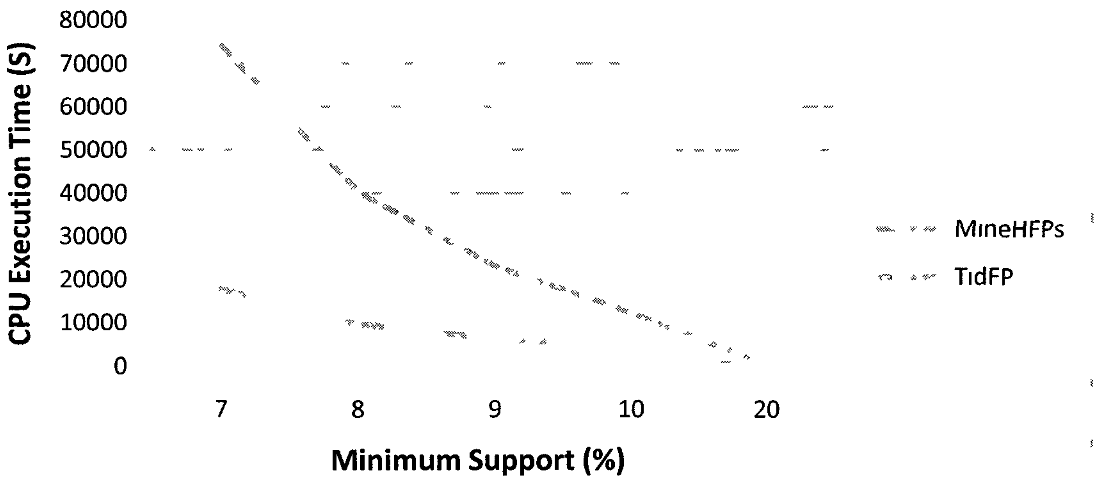

Figure 4.1 CPU executing time on 125K dataset with minimum support 20%, 10%, 9%, 8%, and 7%

Table 4.3 and Figure 4.2 describes the memory usage of the MineHFPs and the TidFP algorithm on **125K** dataset with low minimum support (20%, 10%, 9%, 8%, and 7%).

Algorithms	Memory usage (in M) at different supports(%)				
	20%	10%	9%	8%	7%
MineHFPs	62	430	590	774	1070
TidFP	26	158	214	266	350

Table 4.3 Memory usage on 100K dataset with minimum support 20%, 10%, 9%, 8%, and 7%

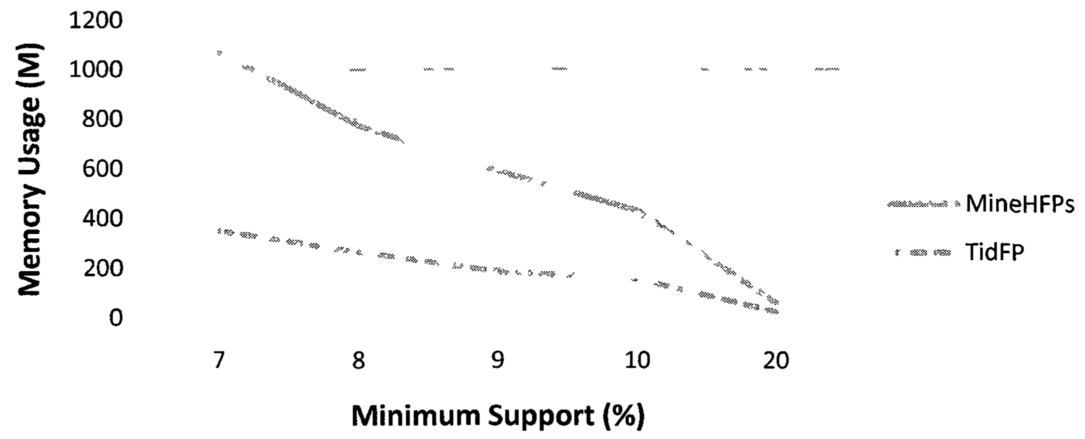

Figure 4.2 Memory usage on 125K dataset with minimum support 20%, 10%, 9%, 8%, and 7%

Table 4.4 is the execution time of the *MineHFPs* and the *TidFP* algorithm on **250K** dataset with low minimum support (20%, 10%, 9%, 8%, and 7%).

Algorithms	Runtime (in Seconds) at different supports(%)				
	20%	10%	9%	8%	7%
MineHFPs	584	8321	12382	19241	35281
TidFP	577	24008	43584	74432	crashed

Table 4.4 CPU executing time on 250K dataset with minimum support 20%, 10%, 9%, 8%, and 7%

Table 4.5 is the memory usage of the *MineHFPs* and the *TidFP* algorithm on **250K** dataset with low minimum support 20%, 10%, 9%, 8%, and 7%.

Algorithms	Memory usage (in M) at different supports(%)				
	20%	10%	9%	8%	7%
MineHFPs	114	814	1098	1145	2001
TidFP	46	282	422	490	crashed

Table 4.5 Memory usage on 250K dataset with minimum support 20%, 10%, 9%, 8%, and 7%

Table 4.6 is the execution time of the *MineHFPs* and the *TidFP* algorithm on **500K** dataset with low minimum support (20%, 10%, 9%, 8%, and 7%).

Algorithms	Runtime (in Seconds) at different supports(%)				
	20%	10%	9%	8%	7%
MineHFPs	1180	16233	24679	37514	68143
TidFP	1150	48077	85027	crashed	crashed

Table 4.6 CPU executing time on 500K dataset with minimum support 20%, 10%, 9%, 8%, and 7%

Table 4.7 is the memory usage of the *MineHFPs* and the *TidFP* algorithm on **500K** dataset with the low minimum support (20%, 10%, 9%, 8%, and 7%).

Algorithms	Memory usage (in M) at different supports(%)				
	20%	10%	9%	8%	7%
MineHFPs	222	1150	2130	2770	3839
TidFP	78	530	722	crashed	crashed

Table 4.7 Memory usage on 500K dataset with minimum support 20%, 10%, 9%, 8%, and 7%

From Table 4.2, 4.4, and 4.6, we can see that the *MineHFPs* algorithm outperforms the *TidFP* at the low minimum support thresholds. The *MineHFPs* algorithm is approximately 3.5 times faster than the *TidFP* algorithm for a 125K dataset, 3.9 times faster for a 250K dataset, and 4.4 times faster for a 500K dataset when the minimum support is lower than 20%. As the size of the dataset is increased, the performance margin between the *MineHFPs* and the *TidFP* algorithm increases in the favour of the *MineHFPs* algorithm. From Table 4.3, Table 4.5, and Table 4.7, we can see that the *MineHFPs* algorithm has greater memory usage compared to the TidFP algorithm. The memory usage of the *MineHFPs* algorithm is approximately 2.8 times, 2.5 times, and 2.6 times greater than the TidFp algorithm for respective dataset sizes of 125K, the 250K, and 500K (at the minimum supports of 20%, 10%, 9%, 8%, and7%).

Table 4.8 and Figure 4.3 describe the execution time of the *MineHFPs* and the *TidFP* algorithm at the minimum support of 10% on dataset sizes of 125K, 250K, 500K, and 1M.

Algorithms	Runtime (in Seconds) on different size of dataset at the minimum support of 10%			
	125K	250K	500K	1M
MineHFPs	3311	8321	16233	34089
TidFP	10264	24008	48077	98858

Table 4.8 CPU executing time at the minimum support 10% on the size of dataset of 125K, 250k, 500k, and 1M

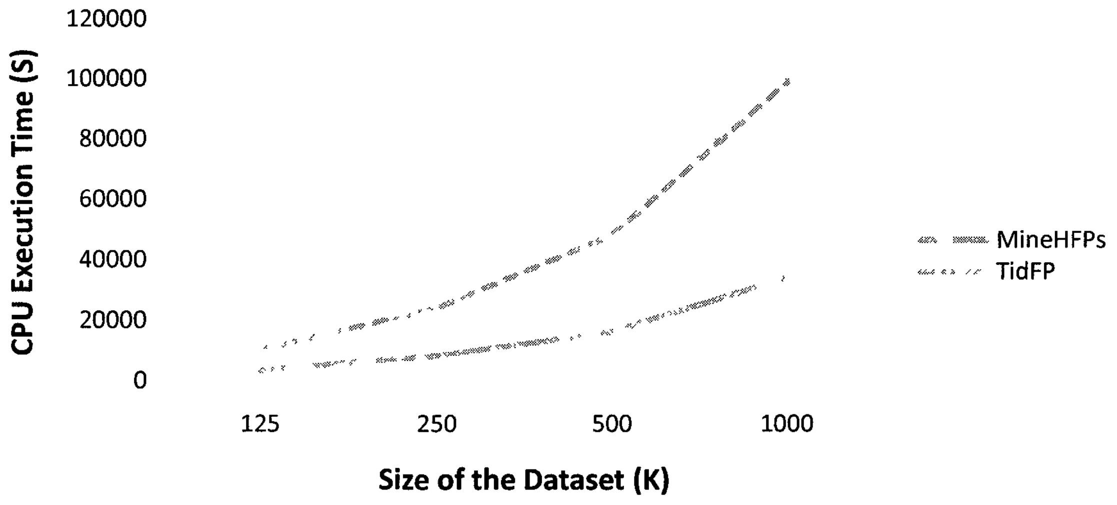

Figure 4.3 CPU executing time at the minimum support 10% on the size of dataset of 125K, 250k, 500k, and 1M

Table 4.9 and Figure 4.4 describes the memory usage of the *MineHFPs* and the *TidFP* algorithm at the minimum support level of 10% on dataset sizes of 125K, 250K, 500K, and 1M.

Algorithms	Memory usage (in M) on different size of dataset at the minimum support of 10%			
	125K	250K	500K	1M
MineHFPs	430	814	1150	1840
TidFP	158	282	530	836

Table 4.9 Memory usage at the minimum support 10% on the size of dataset of 125K, 250k, 500k, and 1M

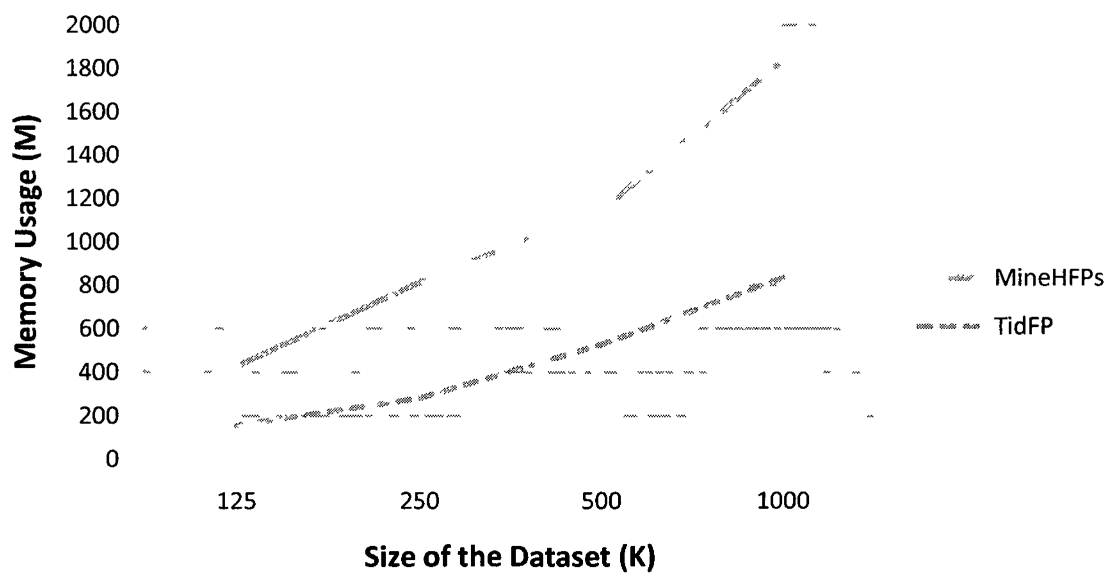

Figure 4.4 Memory usage at the minimum support 10% on the size of dataset of 125K, 250k, 500k, and 1M

From Table 4.8 and Figure 4.4, we can see that when the size of the dataset is doubled, The CPU execution times of both the *MineHFPs* and *TidFP* algorithm approximately increase by double, and so the CPU execution time seems to grow almost linearly. From Table 4.9, we can also see that when the size of the dataset is doubled, the memory usages of both algorithms increased in the same way.

When the minimum support is 20%, the *TidFP* algorithm actually performs better than the *MineHFP* algorithm. This is because when the minimum support is not very low, there are not many frequent patterns and as such not many candidate patterns are generated. Also, it is time consuming to traverse every node of the *MHTree* to count the support of every candidate pattern. Therefore, the advantage of *MineHFPs* algorithm cannot be seen obviously. At high support levels the *MineHFPs* algorithm even performs worse than TidFP algorithm. When the minimum support is low enough, there will be a large number of frequent patterns and a large number of candidate patterns generated. The *MineHFPs* algorithm only needs to scan the database once and generate candidate a pattern once. However, TidFP algorithm needs to re-generate same candidate patterns for every query. Most importantly, *oomap-gen join* can reduce the k-itemset candidate pattern generation and avoid unnecessary support counting. When the number of candidate patterns is large, the advantage of the *oomap-gen join* is more obvious. However, because the *MHTree* stores the transaction ids, the *MineHFPs* algorithm utilizes more memory. It is a reasonable trade off.

4.4 Time and Space Complexity Analysis

This section provides a comparsion of time and space complexity of the Apriori algorithm, the TidFP algorithm, and the MineHFPs algorrithm.

4.4.1 Time and space Complexity of the Apriori algorithm

The Apriori algorithm has two main steps: 1) Candidate patterns generation, and 2) support counting of candidate patterns. Candidate pattern generation performs a breadthfirst traversal over lattice to find the k-itemsets candidate patterns, as shown in Figure 4.5.

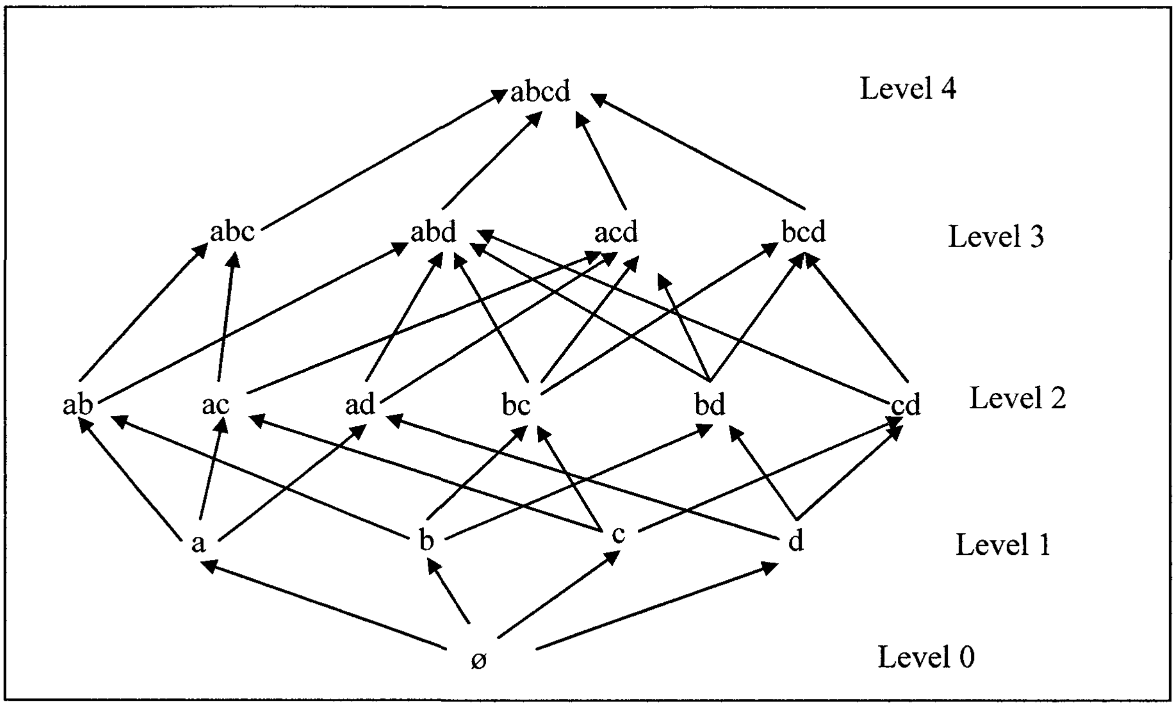

Figure 4.5 lattice of candidate pattern generation

Let |S| be the length of a maximal frequent pattern. It is well known that the worst case time complexity of generating candidate pattern of the Apriori algorithm is:

$$O(2^{|S|})$$

Let |D| be the number of transactions in the database and let |C| be the length of the longest transaction. Then, the worst case time complexity of support counting is:

$$O(|D|\times|C|\times 2^{|S|})$$

Therefore , the worst case time complexity of Apriori algorithm is:

$$O(2^{|S|} + |D|\times|C|\times 2^{|S|}) = O(|D|\times|C|\times 2^{|S|})$$

The space complexity of Apriori algorithm is $O(2^{|S|})$.

4.4.2 Time and space Complexity of the TidFP algorithm

The TidFP algorithm is an Apriori-based algorithm. Let |S| be the length of a maximal frequent pattern. The time complexity of generating candidate patterns is same as Apriori algorithm. Therefore, the worst case time complexity of candidate patterns generation of algorithm TidFP is $O(2^{|S|})$.

Support counting of the TidFP algorithm is performed by bitmap operation. Let |D| be the number of transactions in the database. The worst case time complexity of support counting of the TidFP algorithm is:

$$O(|D|\times 2^{|S|})$$

Therefore, the worst case time complexity of TidFP is:

$$O(|D|\times 2^{|S|} + 2^{|S|}) = O(|D|\times 2^{|S|})$$

Besides memory space for holding the candidate pattern, the TidFP algorithm needs additional memory space to hold the transaction id list of every candidate pattern. Therefore, the worst case space complexity of TidFP algorithm is:

$$O(|D|\times 2^{|S|})$$

4.4.3 Time and space Complexity of the MineHFPs algorithm

The algorithm, MineHFPs, proposed in this thesis is an extension of the TidFP algorithm. Let |S| be the length of a maximal frequent pattern. The time complexity of candidate patterns generation is same as those of the Apriori algorithm and the TidFP algorithm. Therefore, the worst case time complexity of candidate patterns generation of MineHFPs algorithm is $O(2^{|S|})$.

The MineHFPs algorithm needs to count the support of each candidate pattern in every class of every database. Let |D| be the number of transactions in the database table (*Root* class table). Let |L| be the number of classes in every database, and |B| be the number of databases. The worst case time complexity of algorithm MineHFPs is:

$$O(|L|\times|B|\times|D|\times 2^{|S|})$$

The MineHFPs algorithm needs memory space for holding the multiple databases inheritance hierarchy tree and every node of the tree holds the transaction ids besides the memory space for holding the transaction id list and candidate pattern. Therefore, the worst case space complexity of MineHFPs is:

$$O(|L|\times|B|\times|D| + |D|\times 2^{|S|}) = O(|D|\times 2^{|S|})$$

In conclusion, the worst case time complexity of the Apriori algorithm is greater than that of the TidFP algorithm. The worst case space complexity of the Apriori algorithm is smaller than that of the TidFP algorithm. We can have the same conclusion from the experimental results of the TidFP algorithm compared with the Apriori algorithm (Ezeife and Zhang (2009)). The time complexity of the MineHFPs algorithm is greater than that of the TidFP algorithm by a factor of $|L|\times|B|$. This is because that the MineHFPs algorithm mines for the frequent patterns at every level of hierarchy. If we apply the TidFP algorithm at every level of hierarchy, we need to run the TidFP algorithm $|L|\times|B|$ times. From the experimental results of section 4.3 of this thesis, the MineHFPs algorithm is a constant factor faster than running the TidFP algorithm $|L|\times|B|$ times on the different size of datasets and at the different minimum support threshold. The memory usage of MineHFPs algorithm is a constant factor smaller than running the TidFP algorithm $|L|\times|B|$ times.

5. CONCLUSION AND FUTURE WORK

The early research on mining multiple databases focuses on applying parallel and distributed computing techniques and developed a series of Apriori-based algorithm to mine frequent pattern, such as Count Distribute (CD) and Distributed Mining Association Rules (DMA) proposed by Agrawal and Shafer (1996). Machine learning techniques, such as clustering, are also used to mine patterns in multiple databases to discover both global and local patterns, such as mining sequential patterns by multiple alignment (approxMAP) proposed by Kum, Chang and Wang (2006). Ezeife and Zhang (2009) proposed mining frequent patterns by transaction ids (TidFP) algorithm based on Apriori technique that generates frequent patterns and counts the support of each candidate patterns by intersecting the transaction ids to avoid multiple scanning of the dataset. Also, TidFP algorithm can mine frequent patterns from multiple database tables by applying set operations on transaction ids. Han and Fu (1995) proposed Mining Multiple-Level Association Rules which discovers the frequent patterns at different concept hierarchy levels. A pattern can be replaced by another pattern in its higher or lower hierarchy level and be mined by Apriori algorithm. The recent work, object-oriented frequent pattern mining (OR-FP) algorithm proposed by Kuba and Popelinsky (2003) is also Apriori based which takes input as object-oriented database and mines class objects and class attributes as the frequent pattern. There is not much research on mining multiple object-oriented databases. However, more comprehensive and detailed real world data, such as different products on a Business to Customer (B2C) Website, their histories, versions, price, images, or specifications are more suitable to be stored in object-oriented databases.

This thesis proposes an object-oriented class model and database schema, and a series of class methods. The methods can mine frequent patterns on each local object database and also mines the Hierarchical Frequent Pattern (MineHFPs) which specify at which hierarchy level the pattern is frequent in a global integrated table by extending Apriori-based TidFP algorithm. The thesis also proposed object-oriented join (*OOJoin*) which joins superclass and subclass tables by matching their type and super type relationships. To improve the performance of the MineHFPs algorithm, This thesis also extends *map-gen join* method used in TidFP algorithm to *oomap-gen join* for generating k-itemset candidate pattern to reduce the candidate generation and avoid unnecessary support counting by indexing the (k-1)-itemset candidate pattern using two position codes, start position and end position tied to inheritance hierarchy. The experimental results show that the proposed *MineHFPs* algorithm for mining hierarchical frequent patterns is approximately 3 to 4 times faster than TidFP algorithm to mine the same patterns but have trade off on costing 2 to 3 times more memory usage. However, the MineHFPs algorithm can discover the frequent pattern at different hierarchy levels in the format of <Tidlist, itemsets, $class_i$>. The TidFP algorithm can only discover the patterns in the format of <Tidlist, itemsets>.

Our proposed object oriented class model and database schema can be applied to other application domains, such as a Student Information System. Every department or faculty has its own database tables C_l. The Root table can be the class enrolment table and it may store the class and students enrolment information.

In this thesis, we are using *IBM quest synthetic data generator* to generate the datasets. However, the data stored in database tables C_i can be collected from webpage contents. The existing system such as WebOMiner (Mutsuddy(2010)) extracts webpage contents and mines them in a object-oriented model. WebOMiner can be improved to extract not only the contents but also the inheritance hierarchy from the Web pages.

The database tables C_i and *Root* do not include any historical attribute such as a time stamp (which may include date, month and year). The historical attribute can display the history of the products and the history of sales transactions. The mining of historical information is also an important problem in the field of data mining.

The Object-oriented class model defined in this thesis gives the same names to classes in different databases. For example, the name "Computer", "Laptop", and "Desktop" are used as names for classes in different databases such as "IBM", "Dell", and "HP". This approach is convenient for integrating multiple data sources to the *Root* table and mining frequent patterns at different hierarchy levels. Future work may wish to modify the algorithm proposed so that classes can be named differently across databases.

The newly defined pattern, HFP (Hierarchical Frequent Pattern), in this paper, is in the format of <Tidlist, Itemsets, class_i>. The *TidFP* algorithm can use the Tidlist of frequent patterns as foreign keys to answer more complex queries in more than one database table. Future work may also consider methods that use the Tidlist or hierarchy class_i of HFP to answer more complex queries in different database tables across databases.

REFERENCES

Agrawal R., Srikant R.: Fast Algorithms for Mining Association Rules. In: Proceedings of the 20th Very Large Database conference, pages 487-499 (1994)

Agrawal R., R. Srikant: (1995) Mining sequential patterns. In: Proceedings of the 11th International Conference on Data Engineering, pages 3-14. (1995)

Agrawal R., Shafer J.C.: Parallel Mining of Association Rules: Design, Implementation and Experience. In IEEE Knowledge & Data Engg.,8(6): pages 962-969. (1996)

Annoni, E., & Ezeife, C. I. (2009). Modeling Web Documents as Objects for Automatic WebContent Extraction. In: proceeding of ACM / LNCS sponsored 11th international conference on Enterprise Information Systems (ICEIS 09) pages 91-100, May 6-10, 2009

Ayres J., Flannick J., Gehrke J., Yiu T.: Sequential Patterns Mining Using A Bitmap Representation, In: Proceedings of the ACM SIKDD conference, pages 429–435 (2002)

Buchner A., Mulvenna M.: Discovering Internet Marketing Intelligence through Online Analytical Web Usage Mining. SIGMOD Record, Vol.27, No.4, 54-61. (1998)

Ceci, M., Malerba, D.: Classifying web documents in a hierarchy of categories: a comprehensive study. Journal of Intelligence Information System, 28(1): pages 37–78. (2007)

Cheung D., Ng V., Fu A., Fu Y.: Efficient mining of Association Rules in Distributed Databases. IEEE Transactions on Knowledge and Data Engineering, Vol. 8, No. 6, pages 911-922(1996)

Cooley R., Mobasher B., Srivastava J.: Data preparation for mining World Wide Web browsing patterns, Knowledge and Information Systems, pages 1-26. (1999)

Dai H.: An Object-oriented Approach to Schema Integration and Data Mining in Multiple Databases. IEEE Computer Society, pages 294-303. (1998)

Dzeroski S. and Lavrac N.: Relational Data Mining. In: proceeding of SIGKDD Explorations, 5(1) pages 339-364(2001)

Ezeife, C.I. and Barker K.: Distributed Object Based Design: Vertical Fragmentation of Classes, *International Journal of Distributed and Parallel Databases (DPDS)*, Vol. 6, No. 4, pages 327-360, Kluwer Academic Publishers. (1998)

Ezeife C.I. and Liu Y.: Fast Incremental Mining of Web Sequential Patterns with PLWAP Tree, Data Mining and Knowledge Discovery Journal (DAMI), Vol. 19, pages 376 – 418, Springer Verlag publishers, DOI number10.1007/s10618-009-0133-6. (2009)

Ezeife C. I., Lu Y., Liu Y.: PLWAP Sequential Mining, Open Source Code. In: proceedings of the Open Source Data Mining Wrokshop on Frequent Pattern Mining Implementations, in conjunction with ACM SIGKDD, pages 26–35 (2005)

Ezeife C.I. and Lu Y.: Mining Web Log sequential Patterns with Position Coded Pre-Order Linked WAP-tree. The International Journal of Data Mining and Knowledge Discovery (DMKD), Vol. 10, pages 5-38, Kluwer Academic Publishers. (2005).

Ezeife C.I., Saeed K., and Zhang D.: Mining Very Long Sequences in Large Databases with PLWAPLong. In: proceedings of the 13th ACM sponsored International Database Engineering & Applications Symposium Cetraro, Calabria, Italy, 16-18 September 2009 (IDEAS 09), pages 234 – 241 (2009)

Ezeife C.I., Zhang D.: TidFP: Mining Frequent Patterns in Different Databases with Transaction ID, In: LNCS proceedings of the 11th international conference on Data Warehousing and Knowledge Discovery (DAWAK 09), Linz, Austria, Springer Verlag, pages 125 – 137 (2009)

Fortin S., Liu L. An Object-Oriented Approach to Multi-Level Association Rule Mining, In Proc. of Int'l Conf. on Information and Knowledge Management, pages 12-16 (1996)

Han J., Fu Y.: Discovery of Multiple-Level Association Rules from Large Databases. In: Proceeding of the 21st VLDB Conference, pages 420 - 431 (1995)

Han J., Pei J., Yin Y.: Mining frequent patterns without candidate generation, In: Proceedings of ACM SIGMOD international conference on Management of data and Symposium on Principles of Database Systems, pages 1-12. (2000)

Juan T., Manuel P., Gomez J., Song L.: Designing Data Warehouses with OO Conceptual Models: IEEE Computer, special issue on Data Warehouses, pages 66-75 (2001)

Kadri O., Ezeife C.I.: "Mining Uncertain Web Log Sequences with Access History Probabilities", In: proceedings of the 26th ACM Symposium on Applied Computing, ACM SAC, DTTA - Database Theory, Technology, and Application, Tunghai University, TaiChung, Taiwan, March 21 – 24, 2011, pages 1064-1065. (2011)

Kemper A., Moerkotte G.: "Object-Oriented Database Management", a book published by Prentice-Hall Inc, ISBN: 0-13-629239-9. (1994)

Konovalov A.: Object-oriented Data Model for Data Warehouse, In: ADBIS 2002, LNCS, pages 319-325. (2002)

Kuba P., Popelinsky L.,: Mining Frequent Patterns in Object-Oriented Data, In: Proceedings of the 2nd International Workshop on Mining Graphs, Trees and Sequences, pages 15–25 (2004)

Kum H., Chang J. Wang W.: Sequential Pattern Mining in Multi-Databases via Multiple Alignment. Data Mining and Knowledge Discovery, 12, pages 151-180. (2006)

Lu. Y., Ezeife C.I.: Position Code Pre-Ordered Linked WAP-Tree for Web log sequential pattern mining, In: Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining, pages 337-349 (2003)

Lu Y. and C.I. Ezeife C.I.: Mining Web Log sequential Patterns with Position Coded Pre-Order Linked WAP-tree. The International Journal of Data Mining and Knowledge Discovery (DMKD), Vol. 10, pages 337-349 (2005)

Mabroukeh, N. and Ezeife C.I.: A Taxonomy of Sequential Pattern Mining Algorithms, ACM Computing Surveys (CSUR), Vol. 43, No. 1, Article 3, pages 3:1 – 3:41. (2010)

Mobasher B., Cooley R., Srivastava J.: Automatic personalization based on Web usage mining. Communications of the ACM Volume 43, Issue 8, pages 142 – 151. (2000)

Muslea, I., Minton, S., Knoblock, C.: A hierarchical approach to wrapper induction. In AGENTS'99: Proceedings of the third annual conference on Autonomous Agents, New York, USA. ACM, pages 190-197. (1999)

Mutsuddy T.: Towards Comparative Web Content Mining using Object Oriented Model. Master's thesis, School of Computer Science, University of Windsor (2010)

Pei J., Han J., Mortazavi-asi B., Zhu H.: Mining Access Patterns Efficiently from web logs. In: Proceedings Pacific-Asia conference on Knowledge Discovery and data Mining, pages 396-407. (2000)

Pei J., Hart J., Mortazavi-Asl B., Pinto H., Chen Q., Dayal U., Hsu M.: PrefixSpan mining sequential patterns efficiently by prefix projected pattern growth. In Proc. 2001 Int. Conf. Data Engineering (ICDE'01), pages 215-224 (2001)

Satheesh A., Patel R.: Use of Obejct-oriented Concpet in Database for Effective Mining, In: International Journal on Computer Science and Engineering vol.1(3) pages 206-216. (2009)

Srivastava J., Cooley R., Deshpande M., Tan P.: Web usage mining: discovery and applications of usage patterns from Web data. SIGKDD Explorations, Volume 1, Issue, pages 12- 23. (2000)

Wikepedia., The Free Encyclopedia :Object Databases, http://en.wikipedia.org/wiki/ Object_database. (2011)

Wu X. Zhang Z.: Synthesizing High-Frequency Rules from Different Data Sources. Knowledge and Data Engineering, IEEE Transactions, pages 353-367. (2003)

Zaki M. J.: SPADE: An efficient algorithm for mining frequent sequences, Machine Learning Journal, Special Issue on Unsupervised Learning, Vol. 42, No. ½, pages 31-60. (2001)

Zhang C., Liu. M., Nie W., Zhang S.: Identifying Global exceptional patterns in Multi-database mining. IEEE Computational Intelligence Bulletin, Vol.3 No.1, pages 19-21 (2004)

Zhao, H., Meng, W., Wu, Z., Raghavan, V., Yu, C.: Fully automated wrapper generation for search engines. In: WWW'05: Proceeding of the 14th international conference on WWW, NY, USA, ACM, pages 66-75. (2005)

Vita Auctoris

NAME	Dan Zhang
PLACE OF BIRTH	Shanghai, China
YEAR OF BIRTH	1979
EDUCATION	Department of Computer Science University of Windsor, Windsor, ON, Canada Honors Bachelor of Computer Science (2006)
	Department of Computer Science University of Windsor, Windsor, Ontario, Canada M.Sc. (2008-2011)