Metadata of the chapter that will be visualized in SpringerLink

Book Title	Transactions on Large-Scale Data- and Knowledge-Centered Systems XIII	
Series Title	558	
Chapter Title	Mining Multiple Related Data Sources Using Object-Oriented Model	
Copyright Year	2014	
Copyright HolderName	Springer-Verlag Berlin Heidelberg	

Corresponding Author	Family Name	**Ezeife**
	Particle	
	Given Name	**C. I.**
	Prefix	
	Suffix	
	Division	School of Computer Science
	Organization	University of Windsor
	Address	Windsor, ON, N9B 3P4, Canada
	Email	cezeife@uwindsor.ca
Author	Family Name	**Zhang**
	Particle	
	Given Name	**Dan**
	Prefix	
	Suffix	
	Division	School of Computer Science
	Organization	University of Windsor
	Address	Windsor, ON, N9B 3P4, Canada
	Email	woddlab@uwindsor.ca

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 comprehensive and detailed complexity of real world data such as capturing multiple related tables representing data schemas of a retail store web site, or capturing multiple databases such as several retail store web sites. Modeling web and other data as a number of object database schemas would enable derived, historical, and comparative mining of multiple databases and tables.

This paper proposes an object-oriented class model and database schema, and a series of class methods including that for object-oriented join (OOJoin) for mining multiple data sources through object oriented model. The OOJoin procedure joins superclass and subclass tables by matching their type and super type relationships. Mining Hierarchical Frequent Patterns (MineHFPs) from multiple integrated databases is done by applying an extended TidFP technique which specifies the object class hierarchy by traversing the multiple database inheritance hierarchy. This paper also extends map-gen join method used in TidFP algorithm to oomap-gen join for generating k-itemset object candidate patterns. The oomap-gen join reduces the number of candidate itemsets generated through indexing of the (k-1)-itemset candidate pattern with start and end position codes for the inheritance hierarchy level. Experimental results show that the proposed MineHFPs algorithm for mining hierarchical frequent patterns is effective and efficient for complex queries.

Keywords (separated by '-') Object-oriented database - Mining frequent patterns - Inheritance hierarchy - Multiple data sources - Hierarchical frequent patterns

Mining Multiple Related Data Sources Using Object-Oriented Model

C.I. Ezeife(✉) and Dan Zhang

School of Computer Science, University of Windsor, Windsor, ON N9B 3P4, Canada
{cezeife,woddlab}@uwindsor.ca
http://cezeife.myweb.cs.uwindsor.ca/

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 comprehensive and detailed complexity of real world data such as capturing multiple related tables representing data schemas of a retail store web site, or capturing multiple databases such as several retail store web sites. Modeling web and other data as a number of object database schemas would enable derived, historical, and comparative mining of multiple databases and tables.

This paper proposes an object-oriented class model and database schema, and a series of class methods including that for object-oriented join (OOJoin) for mining multiple data sources through object oriented model. The OOJoin procedure joins superclass and subclass tables by matching their type and super type relationships. Mining Hierarchical Frequent Patterns (MineHFPs) from multiple integrated databases is done by applying an extended TidFP technique which specifies the object class hierarchy by traversing the multiple database inheritance hierarchy. This paper also extends map-gen join method used in TidFP algorithm to oomap-gen join for generating k-itemset object candidate patterns. The oomap-gen join reduces the number of candidate itemsets generated through indexing of the (k-1)-itemset candidate pattern with start and end position codes for the inheritance hierarchy level. Experimental results show that the proposed MineHFPs algorithm for mining hierarchical frequent patterns is effective and efficient for complex queries.

Keywords: Object-oriented database · Mining frequent patterns · Inheritance hierarchy · Multiple data sources · Hierarchical frequent patterns

1 Introduction

Real world data are complex and good to be presented or modeled as objects. An object-oriented database is suitable for capturing more comprehensive and

This research was supported by the Natural Science and Engineering Research Council (NSERC) of Canada under an operating grant (OGP-0194134) and a University of Windsor grant.

A. Hameurlain et al. (Eds.): TLDKS XIII, LNCS 8420, pp. 1–29, 2014.
DOI: 10.1007/978-3-642-54426-2_6,

detailed complexity of real world data, such as different products on a Business to Customer (B2C) website, their histories, versions, price, images. Changes in contents or structure of a website may cause changes in the schema of the database that stores the web content. For example, a new product demonstrated on a B2C website which has its own specifications will need a different class object schema to store it [2,4,5,11]. Since the object-oriented database allows values of its attributes to be of complex types such as another database object, tables having attributes with type of new product do not need to change in structure, but only the new product object schema is also created. The attribute "new product" can be a set of new product classes whereby members of this set can take on any newly created product class schema as their type. An example schema representation for such a B2C web site is B2C(Webid:string, Products:set of products, NewProducts: set of products). With relational database system, values of all attributes of a table are single-valued such that the same B2C database schema above can be represented as B2C(Webid:string, product1:string, product2:string, newproduct1:string). If the web site gets new products, the B2C schema together with other schemas in the database need to be updated. The object oriented model presents a data structure for more clearly establishing complex relationships (e.g., superclass, subclass, part-of) between different data entities (e.g., classes and tables) so that mining of multiple tables, classes and databases on historical, derived and other data can be accomplished. The object schemas of the complex data types can be used to define version, histories, derived and other features of the products and new products so that when there are changes, only the relevant class structure needs to change in the object oriented database. Therefore, there is a great advantage in using an object-oriented database model to represent contents captured from web sites for comparative analysis as it presents a clear conceptual model that enables diverse, scalable mining of multiple databases which can still be implemented with the relational database management system (DBMS). Some recent work that also used a more realistic conceptual model such as the object oriented model being proposed in this paper to implement analysis of XML data (not multiple databases) include [21]. In an object-oriented database model, the same type of product (e.g., laptop) will be classified in the same class which inherits the properties (attributes) from its superclass (e.g., computer) and also has its own attributes. When a new product joins, a new class (e.g., pad) will be created for this type of product. For example, in a B2C website that sells computers and laptops, an object-oriented database to store the contents of this website has two classes, Computer and Laptop. The class "Computer" has the attributes "CPU", "RAM", "Hard_drive" while the class "Laptop" inherits the above three attributes of its superclass, "Computer" and also has its own attributes "Screen_size" and "Battery_life". If a new product, Pad which is a subtype of laptop comes to the website, a new class "Pad" will be created and it inherits the attributes of "Laptop" class and would also have its own attributes "3G_device" or "Touch_screen". An object-oriented database is a database management system in which information is represented in the form of encapsulated objects (possibly active) rather than static data

values [8,18,22]. Due to the first normal form (1NF) requiring only single valued attributes, relational databases do not allow complex values, such as sets, lists, or other data structures. On the other hand, the attributes of an object-oriented database model can be a complex collection of types, such as, sets, lists, or some other data structure such as another class object. When implemented with an object-oriented database management system, 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 (that is, transactions for manipulating the static data) must be maintained outside of the relational data model itself through mechanisms for querying and manipulating the data. 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. AQ1

2 Object-Oriented Database Schema

An object-oriented database model is represented by a set of classes connected by their class inheritance hierarchy through superclass and subclass relationships [9,18]. An object-oriented database consists of a set of classes, C_i, with a class inheritance hierarchy H which is used to depict superclass and subclass relationships between classes in the object-oriented database and can be represented as a set of pairs of class and superclass in the form of (class, superclass). A superclass (e.g., Computer) of a class (e.g., Laptop), is a generalization of the class such that a class inherits all the attributes and methods of its superclass. Each class is defined as an ordered relation $C_i = (K, T, S, A, M, O)$, where K is the class identifier (e.g., computer id), T is the class type (e.g., Computer), S is the super type (superclass) of the class (e.g., Root), A is a set of attributes of the class (e.g., CPU, RAM, Hard_drive) [9]. M is a set of methods of the class (e.g., get_Number_of_Computer, get_Number_ofSalesof_Computer). O is a set of encapsulated instance objects (equivalent to tuples) of the class which have instances of the class attributes and methods (e.g., computers with specific CPU, RAM, Hard_drive and have instances of class methods). For example, a computer retail store object-oriented database consists of four classes (Root, computer, laptop, desktop), which are related through class inheritance hierarchy, *H*. H is specified in the format of all pairs of (class, superclass) relationships where H={(Root, Computer), (Laptop, Computer), (Desktop, Computer)}. Root class is the special class which exists in every database and is the only class with no superclass. Database schema for this database is provided as:

C_1 = (K_1, Type, Super, {oid, CPU, RAM, Hard_drive, computer_name}, o_1, o_2, …, o_n);
C_2 = (K_2, Type, Super, {oid, Screen_size, Battery_life}, o_1, o_2, …, o_n);
C_3 = (K_3, Type, Super, {oid, Graphic}, o_1, o_2, …, o_n);

Multiple object-oriented databases and classes can also be connected by their class inheritance hierarchy through superclass and subclass relationships with a multiple object-oriented database inheritance hierarchy MH which is used to

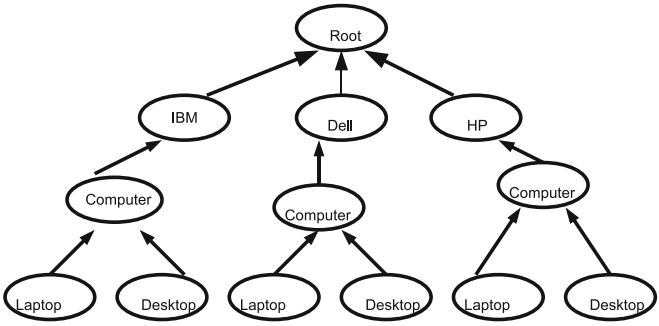

Fig. 1. The Multiple databases inheritance hierarchy Tree (MHTree) for 3 databases

depict superclass and subclass relationship between classes and object-oriented databases, and can be represented as a set of pairs of class and superclass in the form of (class, superclass). For example, the inheritance hierarchy in the multiple databases, MH for three computer object databases for IBM, Dell, and HP computers, has every database consisting of four classes (Root, computer, laptop, desktop). Note that when more than one database is integrated in the object schema, the Root class of each database becomes the name of the database. For example, the Root for the IBM database is now IBM since the integrated schema has only one global Root class. This MH is represented with the set of (subclass, superclass) relationships as follows. MH={(IBM, Root), (Dell, Root), (HP, Root), (Computer, IBM), (Computer, Dell), (Computer, HP), (Laptop, Computer), (Desktop, Computer)}.

Definition 2.01. *Tree structure of a multiple database class inheritance hierarchy (MHTree): is the tree structure representation of multiple databases inheritance hierarchy. For example, the MHTree for three object-oriented computer databases for IBM, Dell, and HP is shown in Fig. 1.* ■

In a multiple database, inheritance hierarchy as shown in Fig. 1, there is a Root class. The database schema of the Root class is defined as Root(K, T, S, A, M, O). The Root class table is a transaction table which records the transactions on the classes in the multiple database. For example, sales transactions from different object databases can be recorded in the Root table. K in this Root table is 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 consisting of the set of super_type attributes of C_i class in T called $super_i$ (the number of $super_i$ depends on the levels of the hierarchy of C_i) and all attributes A of C_i. M is the set of class methods which are behaviors of the Root class, such as those for updating the Root table and mining patterns in the Root table. O is the set of instance objects of the transactions (one object stands for one transaction). For example, a sales transaction of a purchased laptop from IBM database recorded in the Root table (sales transaction table) has object id as an instance of K for transaction id (an integer number), class type T

which indicates the database or the most senior ancestor class of the path where the transaction comes from (database is "IBM" in this case), super_type S of the T class of the transaction, which is "Root", the attributes A of the transaction class path (IBM/Computer/Laptop) includes two super types of the transaction class (IBM/Computer/Laptop). The super_type S of a transaction class consists of class hierarchy from the Root to the class the transaction is on. Thus, $super_1$ is Computer and $super_2$ is Laptop. In the computer object database, there are two levels of the hierarchy and thus, the number of possible $super_i$'s in the Root table corresponds to the length of the inheritance hierarchy. If a transaction record of the Root table concerns a desktop computer, then, the class path of this transaction is IBM/Computer/Desktop and in that case, the $super_1$ (is Computer) and $super_2$ (is Desktop). The attributes of all classes in the hierarchy make up the attributes of the Root class, A and in this case, they are: C_i, CPU, RAM, Hard_drive, Screen_size, battery_life and Graphic. In an object-oriented database model, the instantiated objects (instances) are referenced (retrieved) by following their object pointers. The relational model has the advantage of availability for its database management system for implementation purposes. Thus, we have chosen to have the current implementation of our object oriented mining (OOMining) model with the readily available relational DBMS. The relational database model also provides a clear visual conceptual representation of table schemas showing all attributes of a table (a relational table can be used to represent an object oriented class), including the information on the class inheritance hierarchy. However, converting the object-oriented database model into a relational model poses some challenges with regards to extensions needed for such operations as join operation between objects of different classes. 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 object join operation in Sect. 4.1. We can also define the object-oriented database as a relational database represented with a set of tables (relations) as classes connected through foreign key relationships as inheritance hierarchy. The foreign keys in our object-oriented database model of object database ODB = a set of classes C_i including the Root class, where each C_i = (K, T, S, A, M, O) and Root(K, T, S, A, M, O) are realized through the inheritance hierarchy using the subclass and superclass relationships as defined in the class type T, supertype S attributes of each object class. For example, a relational database schema that represents the computer world object database and its Root class is shown below:

Computer (comp_id: string, type: string, super_type: string, cpu: string, ram: string, hard_drive: string);
Laptop (laptid: string, type: string, super_type: string, screen_size: string, battery_life: string);
Desktop (deskid: string, type: string, super_type: string, graphic: string);
Root (transactionid: integer, type: string, super_type: string, super1: string, super2: string, cpu: string, ram: string, hard_drive: string, screen_size: string, battery_life: string, graphic: string);

Table 1. Object table of computer class

Comp_id	Type	Super_type	CPU	RAM	Hard drive
Comp1	Laptop	Computer	2 GHz	2 GB	250 GB
Comp2	Laptop	Computer	2 GHz	2 GB	320 GB
Comp3	Laptop	Computer	3 GHz	4 GB	350 GB
Comp4	Desktop	Computer	3 GHz	4 GB	500 GB
Comp5	Desktop	Computer	3 GHz	4 GB	500 GB
Comp6	Desktop	Computer	3 GHz	4 GB	500 GB

Table 2. Object table of laptop class

Lap_id	Type	Super_type	Screen_size	Battery_life
Lapt1	Ideapad laptop	Laptop	15”	3 h
Lapt2	Ideapad laptop	Laptop	15”	3 h
Lapt3	Thinkpad laptop	Laptop	17”	3.5 h

In the above relational database schema, comp*id* is the primary key of the computer table, lapt*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 computer object database for the respective classes of Computer, Laptop and Desktop is shown in Tables 1, 2, and 3.

Table 1 is the Computer class table that stores the specifications of computers and contains all instances of all computer types. Tables 2 and 3 store the specifications of laptops and desktops which inherit from the computer class. An example of the Root class table that records all computers purchased from different databases, such as IBM, Dell, or HP is shown in Table 4. Table 4 is a sales transactions table containing eight Root class instance objects where every object indicates one transaction of computer purchased from database specified in ‘Type’. In the schema of the Root(K, T, S, A, M, O),“Oid” is the object id for each transaction. Of course, “oid” is an instance of K (transaction id), which is represented by an integer number. “Type” is the class type T of the transaction which indicates the database or the full inheritance path for the class involved in the transaction. For example, in transaction 1 of the Root table, it can be seen that the full class path for this transaction is “IBM/computer/laptop”. Although in Type, the most senior ancestor class (IBM) in the path is recorded, the attributes of $super_1$ and $super_2$ will record the other classes of “computer” and “laptop” along this inheritance hierarchy of the transaction. Thus, “Types” are recorded as “IBM”, “Dell”, or “HP”. The “Super_type” is the Root class in this case. The set of attributes (A) of the Root class, includes: (1) $super_1$ (computer) and $super_2$ (laptop or desktop) for the class of the transaction. In this example, computer class has subclasses laptop and desktop. There are two levels of the hierarchy and so there are 2 “super” attributes and for each $super_i$ attribute (e.g., at Computer class level), the domain (number of possible values)

Table 3. Object table of desktop class

Desk_id	Type	Super_type	Graphic
Desk1	Work station	Desktop	256M
Desk2	Work station	Desktop	256M
Desk3	Desktop	Desktop	512M

Table 4. An instance of the root class table

Oid	Type	Super type	Super1	Super2	CPU	RAM	Hard drive	Screen size	Battery life	Graphic
1	IBM	Root	Computer	Laptop	2 GHz	2 GB	250 GB	15”	3 h	
2	IBM	Root	Computer	Laptop	2 GHz	4 GB	320 GB	15”	3 h	
3	Dell	Root	Computer	Laptop	2 GHz	2 GB	350 GB	17”	3.5 h	
4	HP	Root	Computer	Desktop	3 GHz	4 GB	500 GB			256M
5	HP	Root	Computer	Desktop	3 GHz	4 GB	500 GB			256M
6	Dell	Root	Computer	Desktop	3 GHz	4 GB	500 GB			512M
7	IBM	Root	Computer	Laptop	2 GHz	2 GB	320 GB	15”	3 h	
8	HP	Root	Computer	Laptop	3 GHz	4 GB	350 GB	17”	3.5 h	

consists of its number of breadth-wise sibling classes itself included (e.g., it is one for class computer), while for $super_i$ class at the laptop level, the number of possible values is two consisting of the two sibling classes, laptop and desktop. Thus, with the example database, the Root class has two $super_i$ classes as $super_1$ (computer) and $super_2$ (laptop or desktop). If there are n levels of hierarchy, there will be $super_1, \ldots, super_n$. (2) CPU, RAM, Hard_driver, Screen_size, Battery_life, Graphic are all attributes of all the classes in the hierarchy consisting of computer, laptop, and desktop classes.

2.1 Frequent Pattern Mining in Object-Oriented Model

Frequent patterns are itemsets that appear in a data set with frequency (also called support) not less than a user-specified threshold (also called minimum support). Frequent pattern mining is the task of discovering frequent patterns from 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. Frequent pattern mining in a single relational database table is used to find the itemsets whose frequencies over all transactions in the database table are no less than a user-specified threshold (also called minimum support). Therefore, frequent patterns in traditional database system consist of items or combination of items (itemsets). In an object database table, every object can be considered as one row (tuple) of a relational database table. The attributes of the object can be considered as object itemsets (patterns). Mining frequent patterns in object table is used to discover object attributes or combinations of object attributes that appear frequently in all objects of

the class (or table) [15,20]. In Table 1, a computer class table has attributes "CPU", "RAM", "Hard_drive". The objects in Table 1 have attributes, such as < 2 GHz >, < 3 GHz >, < 2 GB >, < 4 GB >, or < 500 GB >. These attributes can be considered as itemsets. Based on Table 1 (sample of computer objects table), some frequent pattern mining queries that can be answered are:

Query 1: What are the most frequently used hardware components (CPU, RAM, hard drive) in IBM computer model products with a minimum support of 50 %? Query 1 can be answered by applying one of the frequent pattern mining algorithms, such as TidFP [12] on Table 1.
Query 2: What are the most frequently used hardware components (CPU, RAM, Screen size) in IBM laptop model products with a minimum support of 50 %?. Query 2 cannot be answered by applying TidFP algorithm on only computer class Table 1 or only on laptop class Table 1, because laptop IS-A-TYPE of computer and the computer class does not contain the specialization features of a laptop. Similarly, the laptop class alone does not contain the generalization features of a computer. Thus, to answer Query 2, there is need to involve the two tables, Tables 1 and 2. Tables 1 and 2 for classes computer and laptop need to be joined first, then we need to apply frequent pattern mining algorithms on the joined table. If we want to mine frequent patterns of the hardware specifications of computers that have been sold, we need to mine sales transaction table (shown in the Root Table 4). Assume 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 TidFP algorithm on Table 4, we can only obtain the patterns in a format of transaction id list and itemset, <Tidlist, itemset >, <1,2,3,7, 2 GHz>, <4,5,6,8, 3 GHz>, <1,3,7, 2 GB>, <2,4,5,6,8, 4 GB>, <1,3,7, 2 GHz,2 GB>, and <4,5,6,8, 3 GHz,4 GB>. However, query 3 is not good enough to discover patterns in different hierarchies in an integrated multiple database table such as the Root Table 4. This table integrates information of hierarchy from multiple class tables in different databases using the object oriented data model. Therefore, we need the queries that can not only mine the frequent patterns, but also specify at which hierarchy level the pattern is frequent. For example, the queries such as:
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 that are sold by a particular company like Dell with a minimum support of 50 %? To answer queries labeled as query 4 and query 5 (queries mining frequent patterns in transactional table), the algorithm is required to mine the attributes of computer, laptop, or desktop classes (computer, laptop, or desktop specifications) and also specify if the pattern is frequent at which hierarchy level.

Hierarchical Frequent Pattern. The TidFP algorithm [12] proposes a method that mines frequent patterns with transaction IDs to enable mining frequent patterns from more than one database table. With its technique, the resultant frequent patterns from more than one table are found by performing appropriate set operations of intersection, union and others on frequent patterns from different tables aided by common transaction ids from those tables as the tables were not pre-joined before mining. Thus, in TidFP, the frequent patterns are combinations of itemsets and their transaction id sets in the format of <Tidlist; itemsets>. Example queries such as Query 4 and Query 5 are looking for patterns that are frequent in different class hierarchy levels, and need to specify which hierarchy levels the pattern belongs to. Therefore, a new term, called hierarchical frequent pattern is defined.

Definition 2.11. *Frequent patterns 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 in which class hierarchy that a frequent pattern (itemset) appears. For example, a pattern <1,3,4; 2*GHz*,2 G; laptop/computer/IBM> where 1,3,4 are transaction IDs(Tidlist), 2 GHz, 2 G are itemsets, and laptop/computer/IBM is the class hierarchy of the class starting from the class to the Root.* ■

Contributions and Outlines. The contributions of this paper are as follows.
1. In order to enable mining diverse data from more than one database and table (e.g., representing different B2C product web sites like CompUSA and BestBuy), we define an object-oriented class model where each database is represented by a set of object classes, their class inheritance hierarchy and the Root transaction class (Sect. 2). The inheritance hierarchy is specified as a set of type, supertype pairs. The database schema is defined as a set of object classes C_i, where C_i = (K, T, S, A, M, O) for K its class id, T its class type, S its set of superclass types, A its set of attributes, M its set of methods and O its set of instance objects.
2. In Sect. 4, we define proposed techniques including: the method called Object-Oriented Join (OOJoin) which joins superclass table C_{super} and sub class table C_{sub} by selecting the tuples which have distinct object id, $C_{super}.K$ and $C_{sub}.K$ from the result of $C_{super} \bowtie C_{sub}$, that is, selected tuples with distinct object ids occur where $C_{super}.T = C_{sub}.T or\ C_{super}.T = C_{sub}.S$.
3. We define the new term, hierarchical frequent pattern, HFP, formed as <Tidlist; Itemset; Hierarchy>, where Tidlist is a set of object ids drawn from the set of instances of K. Itemset is a set of class attributes drawn from the set A, and Hierarchy is a sequence of classes from Root to class, C_i (called in the pattern as $class_i$). Hierarchical frequent pattern specifies at which hierarchy level the pattern is frequent and is an extension of the TidFP's pattern <Tidlist; itemset>.
4. We 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 the multiple database hierarchy tree (MHTree) with the 1-itemset candidate patterns and transaction IDs.
5. We extend the map-gen join used in TidFP algorithm to oomap-gen join for generating k-itemsets candidate patterns during the process of MineHFPs

algorithm to reduce the number of k-itemsets candidate patterns and avoid unnecessary intersecting of transaction ids by indexing the patterns using two position codes according to inheritance hierarchy, start position and end position and checking the position code before generating k-itemsets candidate patterns.

Section 3 has other related work, Sect. 5 has comparative analysis while conclusions and future work are presented in Sect. 6.

3 Other Related Work

Frequent pattern mining algorithms, such as Apriori [1,3,23] and FP-tree [16], can only mine frequent patterns from one single database table. They cannot discover frequent patterns from multiple tables and multiple data sources. And also they cannot discover patterns in different class hierarchies, as the inputs of these algorithms are simple transactional database tables with no class inheritance hierarchies. These frequent pattern mining algorithms such as Apriori and FR-tree, and TidFP algorithm take one database table as input. The database table contains a number of transactions (or tuples) to be mined. Each transaction contains one transaction id and the patterns (or attributes involved in the transaction such as tv, laptop, desktop). For example, in one transaction <1,a, b, c, d>, "1" represents transaction id while the items purchased by the transaction id are represented as "a", "b", "c", "d" represent patterns. The TidFP algorithm [12] 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 represents each frequent i-itemset as an m-attribute tuple of the form $< F_{i1} ; T_{1i1} , T_{2i1}, \ldots , T_{mi1} >$, where F_{i1} is the first frequent i-itemset, and T_{mi1} is the mth transaction id of the first frequent i-itemset. For example, given the minimum support of 50 % and a table with only 4 transactions with transaction ids $D_1 \ldots D_4$ where each transaction has a list of itemsets drawn from the list 1, 2, 3, 4, 5. The TidFp algorithm would find the list of frequent 1-itemsets as $F_1 = \{< 1, D_1, D_3 >, < 2, D_2, D_3, D_4 >, < 3, D_1, D_2, D_3 >, < 5, D_2, D_3, D_4 >\}$. This means that the 1-itemset 1 is frequent because it can be found in 2 database transactions D_1, and D_3. To find the 2-candidate itemsets, the algorithm would obtain the 2-itemset list by joining the same way the Apriori-gen would obtain those, but would now obtain the resulting transaction id list as the intersection of the transaction id list of the two joined patterns. Thus, a mapgen-join of the two 1-itemset patterns $< 1, D_1, D_3 >$ and $< 2, D_2, D_3, D_4 >$ will yield the resulting 2-itemset pattern $< 1, 2, D_3 >$. The TidFP algorithm does not mine frequent patterns in integrated object-oriented multiple databases with inheritance hierarchies, nor specify the hierarchy levels that patterns belong to and carries the extra overhead of using set operations to integrate discovered patterns from individual related tables. Existing work, such as in Mining Multi-level Association Rule [14], mining in distributed databases [6,7,13,17] replace the patterns by another pattern in higher or lower hierarchy

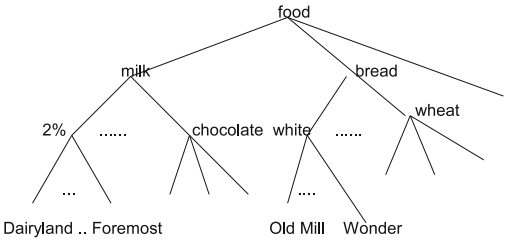

Fig. 2. The itemset concept hierarchy Tree

level and discover frequent patterns in different concept hierarchy level. However, these algorithms do not take object databases as inputs and do not consider the objects or object attributes as patterns.

For example, "2 % Foremost milk" is encoded as "112". Following 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. For example, the transaction 1 in the transaction table is encoded as <1,111, 112, 211, 212>. In this transaction, the pattern "111" represents 2% Dairyland milk, the pattern "112" represents 2 % Foremost milk, the pattern "211" represents white OldMills bread, the pattern "212" represents white Wonder bread. Although the concept hierarchy provides some encoding of hierarchical semantic information about individual items (attributes), it is not the same as an object schema for representing the entire set of tables (classes) and the relationships between them. The OR-FP algorithm [19] takes object-oriented database as input and mines 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. The data in their object-oriented database is represented as: oi: class = $\{attribute_1, attribute_2, \ldots, attribute_n\}$. For example, parts of the sample data used by this system are: o1: Person = {Smith, Canada, 16000}; o2: Actor = {John, Canada, 12000} (Fig. 2). AQ2

4 Mining Multiple Object-Oriented Databases

In this section, we define the object-oriented class model and a set of class methods in different classes. These class methods are able to integrate multiple data sources (by updating the Root class table), join object tables, and answer frequent pattern mining queries. The object-oriented database model consists of (1) a Root transactional class, Root (2) a set of object classes, $C_i \ldots C_n$, and (3) the inheritance hierarchy that defines the superclass-subclass relationships between the object classes, HTree. The structures of the three components are given as Algorithm 4.01 for the object database model.

Algorithm 4.01 *(The Object Database Model)*

OOModel()
begin
Root{
a set of transaction attributes A_i
//including super_type and all physical attributes of C_i
private void InsertTransactions;
private set MineRootFPs;
public set OOJoin;
}
Set of Classes $C_i \ldots C_n$ where for each C_i{
a set of physical attributes A_i
private set MineClassFPs;
}
Class Inheritance Hierarchy HTree in the form (subclass,superclass){
a set of (subclass,superclass) relationships
}
end

Class C_i has a set of physical attributes which are the properties of the class C_i. In the example of computer object database, physical attributes are "CPU", "RAM", and "Hard_drive" of a "Computer" class, or "Screen_size" and "Battery_life" of a "Laptop" class. Class Root has a set of transaction attributes. The transaction's attributes include a set of $super_i$ which consists of all the hierarchical super_types of the leaf class C_i and all physical attributes of classes C_i of the database. Private method InsertTransactions of Root class is used to insert transactions into the Root table and is only called in the class Root.

4.1 Object-Oriented Join (OOJoin)

To answer query 2 in Sect. 2.1, the computer class Table 1 and laptop class Table 2 need to be joined first. In the object database schema we defined in Algorithm 4.01, classes are connected by superclass and subclass relationships in the object-oriented database. Object-Oriented Join (OOJoin) is defined as a method which joins superclass and subclass tables on their type and super_type foreign keys. The main algorithm of OOJoin is shown as Algorithm 4.11.

Algorithm 4.11 *(OOJoin Algorithm)*

Algorithm OOJoin()
Input: Super class table C_{super}, sub class table C_{sub},
superclass primary key K_1, the superclass foreign keys T_1 and S_1,
subclass primary key K_2, the subclass foreign keys T_2 and S_2.
Output: A set of tuples of objects on Table T_d
Other Variables: Table T_c to hold result of cross product of
two class tables, initialized as empty
Table T_t for tuples of $C_{super}.T_1 = C_{sub}.T_2$ or

$C_{super}.T_1 = C_{sub}.S_2$, initialized as empty
List1: set of IDs of super class table, initialized as empty.
List2: set of IDs of sub class table, initialized as empty.

Begin
1.0 $T_c = C_{super} \times C_{sub}$. // cross product of tables
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 $List_1$ and 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

Description of the OOJoin Algorithm

The objective of the OOJoin algorithm is to join a class (e.g., laptop) with its superclass (e.g., computer) so that all inherited attributes of the class stored with the superclass can be obtained for queries of the class involving the inherited attributes as well. The OOJoin algorithm cascades from the class to the most senior ancestor class. Step 1.0 of the OOJoin algorithm finds the cross product of the super class and the sub class tables and stores the result in a temporary table T_c. The resulting tuples from the cross product operation contain all the attributes of the superclass (e.g., computer) and the subclass (e.g., laptop) including their primary and foreign keys. The subclass (laptop) keys consist of its primary key (K_2 such as lapt*id*), its first foreign key which is the type for the subclass (T_2 such as laptop), and the second foreign key which is the super_type for the subclass (S_2 such as computer). Similarly, the joining superclass (computer) keys consist of computer class primary key (K_1 such as computer_id), first foreign key type (T_1 such as computer), and second foreign key for super_type (S_1 such as Root) are also in the attributes of this table, T_c. Step 2.0 of the OOJoin operation discards certain tuples from the result of the cross product operation according to the following conditions. 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. Step 3.0 in the algorithm further prunes the list of tuples to keep only distinct tuples. The first tuple is always kept. Two lists are created, each list is a list of primary keys. The first list will be referred to as $List_1$ and it is used to store the K_1 primary keys and the other list is referred 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.

Table 5. Result of OOJoin of Computer (C) class with Laptop(L) class

ID	Type	Super	CPU	RAM	Hard drive	Comp name	ID	Type	Super	Screen size	Battery life
Comp1	L	C	2 GHz	2G	250 G	I. laptop	Lapt1	Ideapad	L	15"	3 h
Comp2	L	C	2 GHz	2 G	320G	I. laptop	Lapt2	Ideapad	L	15"	3 h
Comp3	L	C	3 GHz	4 G	350 G	T. laptop	Lapt3	Thinkpad	L	17"	3.5 h

For example, OOJoin operation of Tables 1 and 2 will result in Table 5. The three tuples (comp1, comp2, comp3) of Table 5 join the computer class with laptop class to select all laptop instances with their inherited attributes specified in the join operation to select joined tuple from the cross product if $class_1$.type = $class_2$.supertype or $class_1$.supertype = $class_2$.supertype. With this join, if $class_1$.type = $class_2$.supertype or Table 2. supertype, then the two tuples of Tables 1 and 2 are joined. For example, for tuple comp1, comp1.Type = Laptop in Table 1 and lapt1.supertype = Laptop in Table 2 and these two tuples are joined to obtain tuple comp1 of Table 5. Other results are obtained in similar fashion.

4.2 Mining Frequent Patterns in One Class

The MineClassFPs algorithm is used to mine frequent patterns of any class. This it does by using the OOJoin algorithm to obtain all inherited attributes and methods of the class from its superclasses before it applies either the TidFp algorithm for mining the frequent patterns at different hierarchy levels of the inheritance hierarchy. As shown in the class model, every class C_i has a private method MineClassFPs which mines frequent patterns in the class and outputs a set of class attributes as frequent patterns. The algorithm for MineClassFPs is provided as Algorithm 4.21.

Algorithm 4.21 *(MineClassFPs Algorithm)*

Algorithm MineClassFPs()
Input: class table C to be mined, super class tables (CS_i) of class C where CS_k is superclass of CS_{k-1}, minimum support s%
Other Variables: Joined class table T
Output: A set of frequent patterns FPs.
Begin
1.0 // Call JoinClasses (C, CS_i) to join classes as in step 1.x below
    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
2.0 TidFP(T, s%); // Call TidFP on Joined table T
end

Description of the MineClassFPs Method
Step 1.0 of the algorithm joins the class and all super classes using the OOJoin algorithm. Step 2.0 applies TidFP algorithm which takes resulting table from Step 1.0 and the minimum support to mine the frequent patterns. This private method MineClassFPs can answer queries such as query 2 in Sect. 2.1.

4.3 Mining Hierarchical Frequent Patterns in the Root Class

As shown in the class model (Algorithm 4.01), the Root class has a private method MineRootFPs (as given in the formal algorithm 4.51). This method mines frequent patterns in the class and outputs a set of frequent patterns specifying the levels of the inheritance hierarchy. The hierarchical frequent patterns are mined from a Root table of transactions on classes (tables) in the inheritance hierarchy such as Table 4. The inheritance hierarchies exist in the transaction in the Root table. The algorithm for mining hierarchical frequent patterns first creates multiple database inheritance hierarchy tree (MHTree), such as Fig. 1. Then, the transaction ids of the Root table are stored in the MHTree node according to the inheritance hierarchy of each transaction in the Root table. Then, the algorithm traverses the MHTree through the linkage table to access every node and intersect the transaction ids in every node with transaction ids of 1-itemset candidate patterns to obtain 1-itemset frequent patterns with the hierarchy information. A modified version of map-gen join algorithm in the TidFP algorithm is used to generate 2-itemset candidate pattern, and it then traverses the MHTree to obtain 2-itemset frequent patterns. Finally, it obtains the n-itemset frequent patterns. The process of mining the hierarchical FP from the Root table is given in algorithm 4.51 which starts by obtaining the multiple inheritance tree (MHTree) and this calls the OOJoin algorithm to join each such subclass (e.g., laptop) with its superclass (e.g., computer) so that all inherited attributes of the class stored with the superclass can be obtained. The MineClassFP algorithm 4.21 for mining frequent patterns of any class also uses the OOjoin algorithm to obtain inherited attributes of all superclasses of this class. To mine only class FP, the algorithm would use OOJoin to obtain the full class information before applying the TidFP algorithm to obtain the class FPs with Tids. To mine the Root FPs, the OOJoin is used to create the MHTree before calling the MineHFPs with inheritance hierarchy information about each transaction to generate hierarchical FPs where each iteration involves oomap gen join of frequent F_k with itself.

4.4 Position Coding Method

In the PLWAPLong algorithm [10], two position codes, start position and end position (two 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 traversal of the tree (in the order visit node, left subtree and right subtree) and starting with the root node of the tree having the start position code of 0. The idea of position coding method can be used to represent the

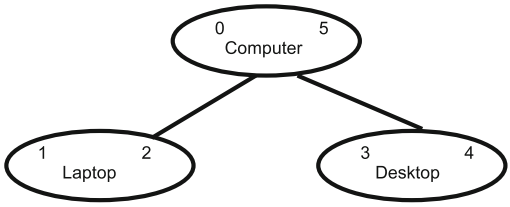

Fig. 3. The position code assigned HTree

level of inheritance hierarchy. As shown in Fig. 1, multiple database inheritance hierarchy can be represented in a tree structure called MHTree, inheritance hierarchy in one database can also be represented in a tree structure called HTree. Position coding method can be used to assign two position codes, start position and end position to each node of the HTree/MHTree by pre-order traversal in order to represent the levels of inheritance hierarchy. The position code assigned HTree of the example computer database is shown in Fig. 3.

In Fig. 3, two position codes are assigned to every node of the HTree. The start position of the Computer class (root parent class) is "0" which is less than the start position of the laptop class (child class) and the desktop class (child class). The end position of the computer class is "5" which is greater than the start positions of the child classes. The laptop class and the desktop class are the sibling classes. The start position and end position of one sibling class are both smaller than those of the other sibling class or the start position and end position of one sibling class are both greater than those of the other sibling class.

The oomap-gen Join Method

Like in the Apriori-gen join, the purpose of the oomap-gen is to obtain the extended (i + 1)-itemsets from the frequent i-itemsets (F_i) by joining F_i with itself oomap-gen fashion. The map-gen join method used in the TidFP algorithm avoids multiple database scanning by intersecting transaction id lists of two patterns being joined to get the resulting transaction id list. The resulting itemset is obtained as the union of the two joined itemsets. However, map-gen join still suffers from large candidate generation and intersecting transaction id lists of every candidate patterns and unable to apply to object hierarchy. When the number of transactions is large, intersecting transaction id lists is an expensive process. Figure 4 provides an example application of the map-gen join of patterns from the example sales transaction table (Root table) shown in Table 4. The patterns in map-gen join are in the slightly reordered format (where Tidlist comes before the itemset list) of <Tidlist; itemset>. In that Fig. 4, it can be seen that the computer feature attribute 1-itemset of <15">, <3 h>, and <256M> are all 1-frequent items where <15">, and <3h>, are both frequent in the Root table transactions with ids 1, 2 and 7. However, the 1-itemset <256M> is frequent in Root table transaction ids 4 and 5. The goal of the map-gen join of these three frequent 1-itemsets $< (1,2,7);(15'') >$, $< (1,2,7);(3\,\mathrm{h}) >$, $< (4,5);(256M) >$

with themselves, is to obtain the frequent 2-itemsets of $< (1,2,7); (15'', 3h) >$, $< (None); (15'', 256M) >$, $< (None); (3h, 256M) >$. The 1-itemsets in the map-gen join above are 15″, 3 h, 256M while <1,2,7> and <4,5> are the transaction id lists. This step is similar to the ap-gen join used in Apriori algorithm. The transaction id lists will be intersected to get the resulting transaction id list.

The oomap-gen join method applies a modification of the map-gen join function. The oomap-gen method can join a set of frequent i-itemsets F_i with itself, where itemsets are from an object oriented class inheritance hierarchy, to obtain the candidate (i+1)-itemsets. Thus, the candidate (i+1)-itemsets C_{i+1} is obtained from the frequent i-itemsets for $i \geq 1$, by joining frequent i-itemsets F_i with itself oomapgen way such that $C_{i+1} = F_i \bowtie F_i$. To join oomapgen way, for each pair of itemsets M and $P \in F_i$ where each F_i itemset is in the format "< transaction id list, itemset, (class start position code, class end position code) >", 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 F_i,
(b) the first i-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 $Tid_{M \cup P}$ is obtained as the intersection of the Tid lists of the two joined i-itemsets M and P and thus, $Tid_{M \cup P} = Tid_M \cap Tid_P$.
(d) To speed up processing, ignore non-joinable patterns by applying the oomap pattern joinable rule which states that only patterns belonging to the same class or classes with ancestor-descendant relationships determined using the start and end position codes of the patterns are joinable.

Definition 4.41. *Ancestor-Descendant Nodes (a,b): Node a of a tree is an ancestor of node b of the tree iff the start position code of node a is less than the start position code of node b, but the end position code of node a is greater than the end position code of node b. For example, in the Htree of Fig. 3, the node Computer with (start,end)codes of (0, 5) is an ancestor of the node Laptop with codes (1, 2).* ■

Definition 4.42. *Sibling Nodes (a,b): Node a of a tree is a sibling of node b of the tree iff both the start and end position codes of node a are either less than or greater than the start and end codes of node b. For example, in the Htree of Fig. 3, the node Laptop node with (start,end)codes of (1, 2) is a sibling of the node Desktop with codes (3, 4).* ■

Definition 4.43. *The oomap pattern Join Rule: If two patterns belong to the same class or belong to two different classes but have an ascendant-descendant relationship as can be determined with Ancestor-Descendant Node definition, they can be joined. If two patterns belong to different classes which have a non ascendant-descendant relationship, they cannot be joined. For example, using this rule for* $< (1,2,7)(15'') > (1,2)$ *oomap-gen join* $< (4,5)(15'') > (3,4)$ *patterns will yield no join since from the ancestor-descendant rule the start and end position codes of the first pattern are all less than those of the second pattern showing that they are not from joinable classes.* ■

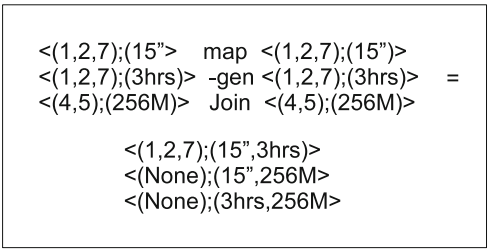

Fig. 4. The map-gen join

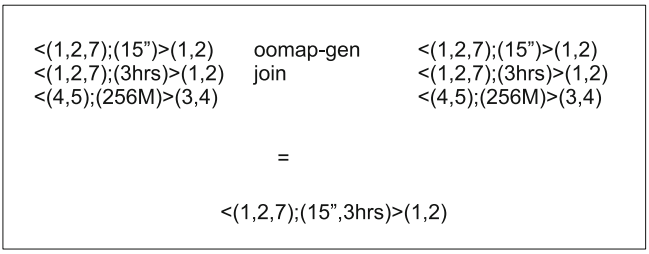

Fig. 5. The OOmap-gen join

From Fig. 4, it can be seen that applying map-gen join on three 1-itemset patterns will result in three 2-itemset candidate patterns. The transaction id lists of newly generated 2-itemset patterns <15”, 256M> and <3h, 256M> are None, because the patterns <15”> and <3h> belong to the laptop class, but the pattern <256M> belongs to the desktop class, they cannot appear in the same transaction in the sales transaction table. Therefore, the position coding method introduced in the previous section will be used to reduce the candidate pattern generation. With the position codes involved, patterns will be checked for their inheritance hierarchy relationships before generating the new candidate patterns. The start position and end position can be used to distinguish the ascendant-descendant or sibling relationships of classes. As given in the oomap-gen pattern join rule, if two patterns belong to the same class or belong to two different classes but have an ascendant-descendant relationship as can be determined with Ancestor-Descendant Node definition, they can be joined. If two patterns belong to different classes which have a non ascendant-descendant relationship, they cannot be joined. Figure 5 shows the result of the oomap-gen join where the start and end position codes may be used to more quickly identify the non-joinable patterns such as $< (None); (15'', 256M >$ that appeared in the result of the map-gen join of Fig. 4 and to exclude them in the result of the oomap-gen join as shown in Fig. 5.

From Fig. 5, it can be seen that patterns are in the format of <Tidlist; itemset>(start, end).

4.5 The MineRootFPs Method

The main algorithm of MineRootFPs method is used for answering comparative queries involving transactions of many classes in the object database which can also include an integration of multiple databases such as computers from several vendors like IBM, Dell, HP as shown in the multiple inheritance hierarchy of Fig. 6. The formal algorithm for MineRootFPs(MH, s%, Root) is given as Algorithm 4.51.

Algorithm 4.51 *(MineRootFPs Algorithm)*

Algorithm MineRootFPs()
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
2.0 StoreTidMHTree(MHTree, Root);
//store transaction ids into MHTree and Obtain TMHtree
3.0 GenOneCand(Root); //generate 1-itemset candidate patterns
4.0 BuildLinkage(TMHTree); //build linkage of TMHTree and obtain TMHTree
5.0 MineHFPs(LTMHTree, C_k, s%)
5.1 C_k = 1-itemset candidate patterns
5.2 F_k = CheckMinS(MHTree, C_k, s%);
5.3 if (F_k is not empty)
5.3.1 C_{k+1} = oomap-gen-join(F_k);
5.3.2 $k = k + 1$
5.3.2 go to step 5.2
End

Description of MineRootFPs Algorithm of the Root Class
Step 1.0 is creates a multiple database inheritance hierarchy tree (MHTree) as shown in Fig. 6. Step 2.0 scans the entire transaction Table 4 and stores the transaction ids into the nodes (class) of the MHTree to create a MHTree with transaction ids stored which is called TMHTree. For example, from Fig. 6, it can be seen that transactions 1, 2, 7 of the Root transaction Table 4 are on IBM laptop computers. Concurrently with steps 2.0, step 3.0, it generates the 1-itemset candidate patterns in the format of <Tidlist; itemset>(start, end). This step is similar to the step of generating 1-itemset candidate patterns in the TidFP algorithm. Step 4.0 is building the header linkage to TMHTree to create a LTMHTree, so that nodes of the tree can be easily accessed. The header linkage

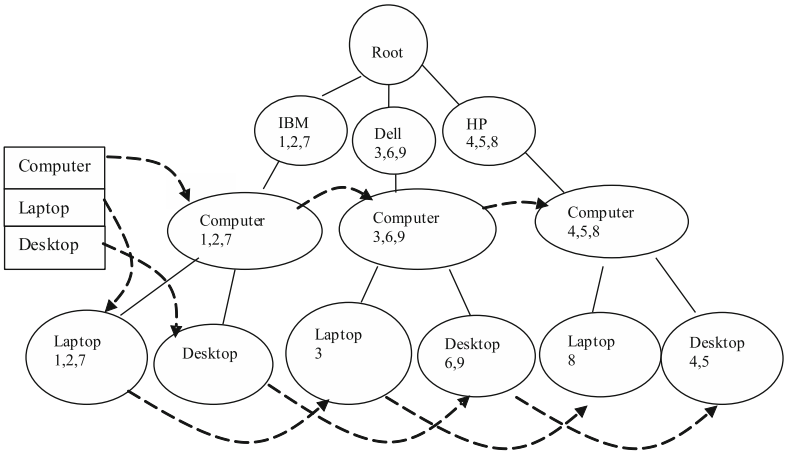

Fig. 6. The LTMHTree:linkage tree multiple inheritance tree

is built such that every unique class in a database (e.g., Computer, Laptop, Desktop) has an entry and using the preorder traversal, all similar classes across multiple databases are linked in a queue. In the computer object database, there are three object tables, "Computer", "Laptop", and "Desktop". Therefore, there will be three entries in the link header table. It builds linkage queue for each entry. Finally, it uses pre-order traversal (visit node, visit left subtree, visit right subtree) to access every node of the tree and store the node into the appropriate queue. An LTMHTree with transaction ids stored and linkage built is shown in Fig. 6. Step 4.0 is for mining the hierarchical frequent patterns in the Root table by calling the MineHFPs algorithm.

Algorithm 4.52 *(MineHFPs Algorithm)*

Algorithm MineHFPs()
Input: linkage built, transaction ids 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 patterns 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

Description of the MineHFPs Algorithm

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 calls the algorithm CheckSupp which uses every 1-itemset candidate pattern to traverse LTMHTree in order 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, it uses oomap-gen-join algorithm 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, it uses algorithm oomap-gen-join to generate 3-itemset candidate patterns. By the same process, k-itemset frequent patterns can be generated. The CheckSupp algorithm is given as Algorithm 4.53.

Algorithm 4.53 *(CheckSupp Algorithm)*

Algorithm CheckSupp()
Algorithm CheckSupp(LTMHTree, Ck, s%);
Input: MHTree, k-itemset candidate pattern with transaction IDs C_k,
in the format of <Tidlist, itemsets, $class_i$ >, k = 1; initially, minimum support s%.
Output: Frequent k-itemsets F_k, in the format of <Tidlist, itemsets, $class_i$ >.
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_i, Hierarchy of every node $class_i$
Begin // Check supports of generated patterns
1.0 For each element c_x in C_k do
1.1 Flag = false;
1.2 For each queue qi do
1.2.1 For each element e_{ij} in the queue qi do
1.2.1.1 intersectTidlist = $c_x.Tidlist \cap e_{ij}.Tidlist$;
1.2.1.2 if((number of IDs in intersectTidlist)/(number of
IDs in $e_{ij}.Tidlist$) >= 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 $e_{ij}.class_i$; 1.2.1.2.4 Flag = true;
1.2.1.3 UTidlist = UTidlist $\cup e_{ij}.Tidlist$;
1.2.2 intersectTidlist = $c_x.Tidlist \cap$ 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* $e_{ij}.class_i$;
1.2.3.3 Flag = true;
1.3 if(Flag = true) Insert c_x *into* F_k;
end

Application of the CheckSupp Algorithm

To serve as an example, the MineHFPs algorithm uses the inputs LTMHTree (Fig. 6), two 1-itemset candidate patterns: <1, 3, 7, 2 GHz> (0,5) and <1, 3, 7, 2 G> (0,5), and a minimum support value of 50 %. 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, 2 GHz>. The Tidlist of the candidate pattern <1, 3, 7, 2 GHz> is <1, 3, 7>. The first node of linkage queue of "Computer <1, 2, 7>" is <1, 2, 7> (according to Fig. 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, 2 GHz> into frequent pattern set F_1. In the same way of processing the candidate pattern <1, 3, 7, 2 GHz> and node "Computer <3,6,9>" is intersected, and pattern <1, 3, 7, 2 GHz> and node "Computer <4,5,8>" is intersected. We find out that pattern <1, 3, 7, 2 GHz> 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, 2 GHz> 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, 2 GHz> is not frequent at the hierarchy "Computer". The Tidlist of candidate pattern <1, 3, 7, 2, 2 GHz> will intersect Tidlist of nodes in "Laptop" linkage queue and "Desktop" linkage queue. The 1-itemset candidate pattern <1, 3, 7, 2 G> will be processed by the same procedure as above and will obtain patterns as: <1, 7, 2 G, computer/IBM>, <1, 2, 2 G, laptop/computer/IBM>. Patterns <1, 3, 7, 2 GHz>(0,5) and <1, 3, 7, 2 G> (0,5) will use oomap-gen join to generate 2-itemset candidate pattern <1,3,7, 2 GHz, 2 G>(0,5). This 2-itemset pattern 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.

5 Implementation and Experimentation

One of the most important contributions of the paper is proposing an object oriented model for representing and mining data from multiple databases while maintaining the class inheritance hierarchy for purposes of answering more complex, historical, derived queries across such integrated multiple database data. The experiments below serve to show both the effectiveness of the proposed algorithms in performing such object oriented mining while remaining reasonably efficient in comparison with existing system such as the TidFP that cannot handle all tasks that can be handled by the proposed approach. To test the performance of our proposed method for mining hierarchical frequent patterns in table Root (transaction table), we use the IBM quest synthetic data generator 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.

5.1 Generating the Class Table C_i

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 for 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 for 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" have already been used to represent the patterns for the Computer class table and the integer numbers from "15" to "60" have already been used to represent the patterns for 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 numbers from "60" to "120" will be used to represent patterns for 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 Sect. 1.2, the database schema of C_i is C_i (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).

5.2 Generate the Root Table

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 to $\|D\|$ sequentially. $\|D\|$ is the number of transactions in the Root table. 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 database (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.

5.3 Performance Comparison

The proposed algorithm MineHFPs is compared with the TidFP algorithm with respect to CPU execution time and memory usage because it is the algorithm that is closest to being able to answer the types of mining queries involving multiple tables and databases which the proposed algorithm and model is designed for. The most important contribution of work is providing a model that can mine multiple database tables and answer such complex queries involving history and derived data. It should also be mentioned that while the proposed approach mines frequent patterns in integrated or joined tables (classes), the TidFP mines FPs from individual tables and integrates the FPs to answer the query through relevant set operations. Thus, this could also be a reason for slower execution time for the TidFP in comparison with the MineHFP in some of the reported experiments.

MineHFP 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 C_i, inheritance hierarchy H, and multiple database inheritance hierarchy MH are all stored in text files. 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 have super1 "4" and super2 "5" to represent "Computer" and "Laptop", respectively. In this section, we compare the performance of our proposed algorithm MineHFPs and the TidFP algorithm. Both the CPU execution times and the memory usages are 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, 250 K, 500K, and 1M. The characteristics of the generated datasets are described in Table 6. Table 7 describes the execution times for the MineHFPs and the TidFP algorithm on 125K dataset with low minimum support (20 %,

Table 6. The characteristics of the generated dataset

Root	Computer	Laptop	Desktop
Table	Class	Class	Class
125K	C7.S4.N20.D125K	C15.S4.N60.D63K	C25.S4.N120.D62K
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
1M	C7.S4.N20.D1000K	C15.S4.N60.D500K	C25.S4.N20.D500K

Table 7. CPU execution time on 125K dataset with varying minimum support

Algorithms (minimum Support)	Execution times (secs) at varying minimum supports				
	20 %	10 %	9 %	8 %	7 %
MineHFPs	290	4186	6356	9606	17785
TidFP	279	12327	23083	40097	74046

Table 8. Memory usage on 100K dataset with varying minimum support

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

10 %, 9 %, 8 %, and 7 %). Table 8 describes the memory usage of the MineHFPs and the TidFP algorithm on 125K dataset with low minimum support (20 %, 10 %, 9 %, 8 %, and 7 %). Table 9 gives the execution time of the MineHFPs and the TidFP algorithm on 250K dataset with low minimum support (20 %, 10 %, 9 %, 8 %, and 7 %). Table 10 gives the memory usage of the MineHFPs and the TidFP algorithm on 250K dataset with low minimum support (20 %, 10 %, 9 %, 8 %, and 7 %). Table 11 is the execution time of the MineHFPs and the TidFP algorithm on 500K dataset with low minimum support (20 %, 10 %, 9 %, 8 %, and 7 %). Table 12 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 %).

From Tables 9, 10, 11, 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 favor of the MineHFPs algorithm. From these tables, we can see that the MineHFPs algorithm has greater memory usage compared with 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, 250K, and 500K (at the minimum supports

Table 9. CPU execution time on 250K dataset with varying minimum support

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

Table 10. Memory usage on 250K dataset with varying minimum support

Algorithms (minimum support)	Memory usage (in MB) at varying minimum supports				
	20 %	10 %	9 %	8 %	7 %
MineHFPs	114	814	1098	1145	2001
TidFP	46	282	422	490	Crashed

Table 11. CPU execution time on 500K dataset with varying minimum support

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

Table 12. Memory usage on 500K dataset with varying minimum support

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

Table 13. CPU execution time at minimum support of 10 % on varying sizes of dataset

Algorithms (dataset size)	Runtime (in Seconds) at different dataset sizes			
	125K	250K	500K	1M
MineHFPs	3311	8321	16233	34089
TidFP	10264	24008	48077	98858

of 20 %, 10 %, 9 %, 8 %, and 7 %). Table 13 describes 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.

6 Conclusions and Future Work

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 represented in an object-oriented database model. This paper proposes an object-oriented class model and database schema, and a series of class methods for mining multiple data sources. This paper also provides mechanisms that allow the flexibility of implementing this model with the popularly used relational DBMS. The methods can mine frequent patterns on each local object database and also mine 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. This paper also proposes object-oriented join (OOJoin) which joins superclass and subclass tables by matching their type and super type relationships. Thus, to implement the OO database model proposed using a relational DBMS, each relational database table corresponds to an OO class, each relation DB tuple corresponds to an OO class instance object. Each relational foreign key attribute is implemented with both the class type and supertype value of the class with the defined OOJoin condition. To improve the performance of the MineHFPs algorithm, this paper 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 the TidFP algorithm to mine the same patterns but have the trade off of 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_i. The Root table can be the class enrolment table and it may store the class and students enrolment information. 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). Future work may include extending this model for representing and comparative analysis of non-structured multiple data sources such as documents, their derived forms (e.g., summaries), historical data sources (data warehouses), derived data (e.g., data warehouse materialized views). The historical attribute can display the history of the products and the history of sales transactions. Although the proposed object oriented data model representation of a database as presented in Sect. 2 currently focuses on a set of classes C_i connected by their class inheritance hierarchy H that is used to depict the superclass and subclass relationships between classes, this model is easily extendible to accommodate as well complex object type or attribute hierarchy where attributes are of type of another existing class. Current implementation and discussions assume all class attributes to be of simple type (e.g., string) and if attributes are of complex types (e.g., CPU is of type computer), they can be accommodated by having those complex attributes

(e.g. CPU) as nested list of attributes (having all attributes of its complex type computer) and applying the process on all attributes including those inherited from the complex type (e.g., computer). Future work should extend the Mine-HFP algorithm to handle nested objects in the model definition such that as the model definition of inheritance hierarchy is provided, that of complex attribute hierarchy is also provided and an equivalent of OOJoin function for obtaining all inherited attributes of a complex attribute defined and used during both MineClassFP and MineRootFP methods.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: 20th International Conference on Very Large Databases, Santiago, pp. 487–499. Morgan Kaufmann (1994)
2. Annoni, E., Ezeife, C.I.: Modeling web documents as objects for automatic web content extraction. In: ACM Sponsored 11th International Conference on Enterprise Information Systems (ICEIS), Milan, Italy, p. 91100. LNCS, Springer (2009)
3. Ayres, J., Flannick, J., Gehrke, J., Yiu, T.: Sequential patterns mining using a bitmap representation. In: ACM SIGKDD Conference, Edmonton, Canada, pp. 429–435. ACM (2002)
4. Ceci, M., Malerba, D.: Classifying web documents in a hierarchy of categories: a comprehensive study. J. Intell. Inf. Syst. **28**(1), 37–78 (2007)
5. Cheng, H., Zhou, Y., Yu, J.X.: Clustering large attributed graphs: a balance between structural and attribute similarities. ACM Trans. Knowl. Disc. Data **5**(2), 1–3 (2011)
6. Cheung, D., Ng, V., Fu, A., Fu, Y.: Efficient mining of association rules in distributed databases. IEEE Trans. Knowl. Data Eng. **8**(6), 911–922 (1996)
7. Dai, H.: An Object-oriented Approach to Schema Integration and Data Mining in Multiple Databases, pp. 294–303. IEEE Computer Society (1998)
8. Ezeife, C.I., Barker, K.: A comprehensive approach to horizontal class fragmentation in a distributed object based system. Int. J. Distrib. Parallel Databases (DPDS) **3**(3), 247–273 (1995)
9. Ezeife, C.I., Barker, K.: Distributed object based design: vertical fragmentation of classes. Int. J. Distrib. Parallel Databases (DPDS) **6**(4), 327–360 (1998)
10. Ezeife, C.I., Saeed, K., Zhang, D.: Mining very long sequences in large databases with PLWAPLong. In: 13th ACM Sponsored International Database Engineering and Applications Symposium, pp. 234–241. ACM (2009)
11. Ezeife, C.I., Mutsuddy, T.: Towards comparative mining of web document objects with NFA: WebOMiner system. J. Data Warehouse. Mining (IJDWM) **8**(4), 121 (2012)
12. Ezeife, C.I., Zhang, D.: TidFP: mining frequent patterns in different databases with transaction ID. In: Pedersen, T.B., Mohania, M.K., Tjoa, A.M. (eds.) DaWaK 2009. LNCS, vol. 5691, pp. 125–137. Springer, Heidelberg (2009)
13. Fortin, S., Liu, L.: An object-oriented approach to multi-level association rule mining. In: International Conference on Information and Knowledge Management, pp. 12–16. ACM (1996)
14. Han, J., Fu, Y.: Discovery of multiple-level association rules from large databases. In: 21st International Conference on very Large Databases, Zurich, Switzerland, pp. 420–431. Morgan Kaufmann (1995)

15. Han, J., Nishio, S., Kawano, H., Wang, W.: Generalization-based data mining in object-oriented databases using an object cube model. Int. J. Data Knowl. Eng. **25**(1), 55–97 (1998)
16. Han, J., Pei, J., Yin, Y., Mao, R.: Mining frequent patterns without candidate generation: a frequent-pattern tree approach. Int. J. Data Mining Knowl. Discov. **8**(1), 53–87 (2004)
17. Jin, Y., Murali, T.M., Ramakrishnan, N.: Compositional mining of multirelational biological datasets. ACM Trans. Knowl. Discov. Data **2**(1), 1–35 (2008)
18. Kemper, A., Moerkotte, G.: Object-oriented Database Management. Prentice-Hall Inc., Upper Saddle River (1994). ISBN: 0-13-629239-9
19. Kuba, P., Popelinsky, L.: Mining frequent patterns in object-oriented data. In: Proceedings of the 2nd International Workship on Mining Graphs, Trees and Sequences, ECML/PKDD, Pisa, pp. 15–25. University of Pisa (2004)
20. Satheesh, A., Patel, R.: Use of object-oriented concept in database for effective mining. Int. J. Comput. Sci. Eng. **1**(3), 206–216 (2009)
21. Sengupta, A.: On the feasibility of using conceptual modeling constructs for the design and analysis of XML Data. ACM Trans. Knowl. Discov. Data **72**, 219–238 (2012)
22. Wikepedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Object_database
23. Zaki, M.: SPADE: an efficient algorithm for mining frequent sequences. Mach. Learn. J., Special Issue on Unsupervised Learning **42**(1), 31–60 (2001)

Author Queries

Chapter 6

Query Refs.	Details Required	Author's response
AQ1	Due to technical reasons, affiliation URL has been changed to "http://cezeife.myweb.cs.uwindsor.ca/". Please check and confirm.	
AQ2	Please check and confirm the inserted citation of Fig. 2. If not, please suggest an alternate citation.	

MARKED PROOF

Please correct and return this set

Please use the proof correction marks shown below for all alterations and corrections. If you wish to return your proof by fax you should ensure that all amendments are written clearly in dark ink and are made well within the page margins.

Instruction to printer	*Textual mark*	*Marginal mark*
Leave unchanged	··· under matter to remain	
Insert in text the matter indicated in the margin		New matter followed by or
Delete	through single character, rule or underline or through all characters to be deleted	or
Substitute character or substitute part of one or more word(s)	through letter or through characters	new character or new characters
Change to italics	under matter to be changed	
Change to capitals	under matter to be changed	
Change to small capitals	under matter to be changed	
Change to bold type	under matter to be changed	
Change to bold italic	under matter to be changed	
Change to lower case	Encircle matter to be changed	
Change italic to upright type	(As above)	
Change bold to non-bold type	(As above)	
Insert 'superior' character	through character or where required	or under character e.g. or
Insert 'inferior' character	(As above)	over character e.g.
Insert full stop	(As above)	
Insert comma	(As above)	
Insert single quotation marks	(As above)	or and/or or
Insert double quotation marks	(As above)	or and/or or
Insert hyphen	(As above)	
Start new paragraph		
No new paragraph		
Transpose		
Close up	linking characters	
Insert or substitute space between characters or words	through character or where required	
Reduce space between characters or words	between characters or words affected	