

# Dealing with Semantic Heterogeneity by Generalization-Based Data Mining Techniques\*

                    Jiawei Han  Raymond T. Ng  
(Simon Fraser University)                    (University of British Columbia)  
                    Yongjian Fu  Son K. Dao  
(University of Missouri - Rolla)                    (Hughes Research Laboratories)

## Abstract

Data mining, or knowledge discovery from databases, may play an important role at the construction of cooperative information systems. A major challenge for building cooperative information systems is the semantic heterogeneity problem. Methods for schema analysis, transformation, and integration have been investigated for providing a good tool to handle this problem. However, schema level analysis may sometimes be too general to solve the problem. Data level analysis, i.e., the analysis of database contents, should be taken into serious consideration. Data mining provides a powerful tool to view database contents at a high abstraction level and transform low-level heterogeneous data into high-level homogeneous information. We discuss the necessity of data mining in cooperative information systems and study the methods for construction and maintenance of multiple-layer databases and intelligent query answering using generalization-based data mining techniques.

## 1 Introduction

The study of heterogeneous databases, or cooperative database systems, has been an active research area for the past decade. Most studies focus on issues such as architectural design, schema translation, schema integration, and transaction control [6, 21]. As the name implies, a key challenge of heterogeneous databases is how to deal with the semantic heterogeneity presented by the multiple autonomous databases. Techniques for schema analysis, translation, and integration attempt to solve the semantic heterogeneity problem at the schema level. That is to say, the solution to the problem is based on finding schemas (e.g., federated schemas in [22]) that all databases involved can agree upon and can relate (i.e., translate) to.

However, based on our observation, schema-level analysis may only touch one aspect of the semantic heterogeneity problem. Many heterogeneous databases, even being analyzed and transformed into federated schemas, may still have problems communicating with each other effectively.

---

\*Research of the first author is partially supported by the grant NSERC-A3723 from the Natural Sciences and Engineering Research Council of Canada, the grant NCE:IRIS/PRECARN-HMI-5 from the Networks of Centres of Excellence of Canada, and grants from MPR Teltech Ltd. and Hughes Research Laboratories. Research of the second author is partially supported by the Natural Sciences and Engineering Research Council of Canada under grants OGP0138055.

A deeper level analysis of semantic heterogeneity, called *data-level analysis*, i.e., the analysis of database contents, should be taken into consideration.

Consider a simple example where the heterogeneous databases are university databases. A graduate admission office receives thousands of applications regularly from all over the world. The schema level transformation and analysis may help transform different schemas of student transcripts into one federated schema, such as,

*grading(name, student\_id, semester, year, course\_num, department, university, . . . , grade),*

which paves the way for information exchange among different databases. Nevertheless, one may still have a difficult time comparing the applicants (i.e., comparing the “so-transformed” data) based on the federated schema. This is because different universities may have quite different standards in course offering and course grading. In many cases, it is necessary to perform a deeper level analysis, i.e., the analysis of database contents, to solve the semantic heterogeneity problem.

The diversity of data and the difficulty of understanding the meaning of the data poses great challenges to cooperating multiple databases based on their information contents [4]. It is difficult to communicate among databases based on the low-level heterogeneous database contents, such as concrete course or grade information in a university database. However, their corresponding high level concepts, such as general course information or grade distribution, could be less heterogeneous and easier to communicate. Thus generalization of low level data to relatively high level concepts may help communicate among different databases despite the existence of semantic heterogeneity of concrete database contents.

In the university example, although different universities may offer courses with very different names and contents, those courses can be classified according to their contents, subjects, and levels. For example, a course “CMPT354 database systems I” could be viewed, based on different concept “granularities”, as “database systems course, introductory\_level”, “DB course”, “CS course”, “applied\_science\_course”, etc. Similarly, *grade* assignment, though diverse at different universities, such as letter grades versus percentage grades, etc., can be transformed into relatively homogeneous high level concepts. Notice that one may specify some data transformation functions between the local schemas of the databases and the federated schema by domain-specific experts. For example, percentage grades can be converted into letter grades based on some conversion formula specified by experts (e.g., B corresponds to the range of [68%, 79%]) and vice versa. However, conversion formulas may not always be possible. It is often desirable to express grades in the federated schema in terms of top-2%, next-8%, top-one-third, etc., which will create a basis for people from one university to understand the transcripts from another. In this case, there is no simple conversion formula that can be set up without taking the distribution of grades into consideration, whereas the distribution of grades can be obtained by performing data level analysis against the database using some data analysis techniques. Especially, a generalization technique which transforms low level diverse data into relatively high level, commonly sharable information will be valuable for

such analysis.

The above analysis demonstrates why, apart from schema level analysis, data level analysis, i.e., the analysis of database contents, is valuable for solving the semantic heterogeneity problem. Many techniques in data mining [8] provide powerful means to view database contents at a *high concept level* (i.e., *a high level of abstraction*) and to transform low-level heterogeneous data into high-level homogeneous information. In this chapter, we study the relationships between data mining and cooperating heterogeneous databases. The data mining methods discussed in this chapter are confined to those for data generalization, summarization, and characterization [12, 4, 15], and methods for the construction of *multiple-layer databases*. We also explore its potential for supporting heterogeneous databases.

The chapter is organized as follows. In Section 2, the concept of a multiple-layer database (MLDB) is introduced. The techniques for generalizing different kinds of data are presented in Section 3. The construction and maintenance of an MLDB is studied in Section 4. Query answering in MLDBs is presented in Section 5. Finally, the potential of MLDBs for supporting heterogeneous databases is discussed in Section 6.

## 2 A Multiple Layered Database

To facilitate our discussion, we assume that the database to be studied is constructed based on an extended-relational data model with the capabilities to store and handle different kinds of complex data, such as structured or unstructured data, hypertext, spatial or multimedia data, etc. It is straightforward to extend our study to other data models, such as object-oriented, deductive, etc. Intuitively, an MLDB is a database composed of several layers of information, with the lowest layer corresponding to the primitive information stored in a conventional database, and with higher layers storing more general information extracted from lower layers. More formally, an MLDB can be defined as follows:

**Definition 1** *A multiple-layer database consists of 4 major components:  $\langle S, H, C, D \rangle$*

1. *S: a database schema, which contains the meta-information about the multiple-layer database structures;*
2. *H: a set of concept hierarchies;*
3. *C: a set of integrity constraints; and*
4. *D: a set of database relations, which consists of all the relations (primitive or generalized) in the multiple-layer database.* □

The first component, a database schema, outlines the overall database structure of an MLDB. It stores general information such as types, ranges, and data statistics about the relations at different

layers, their relationships, and their associated attributes. More specifically, it describes which higher layer relation is generalized from which lower layer relation(s) and how the generalization is performed. Therefore, it presents a route map for schema browsing and database content browsing and for assistance of cooperative query answering and query optimization. More details will be given later.

The second component, a set of concept hierarchies, provides a set of predefined concept hierarchies to assist the system to generalize lower layer relations to high layer ones and map queries to appropriate layers for processing.

The third component, a set of integrity constraints, consists of a set of integrity constraints to ensure the consistency of an MLDB.

The fourth component, a set of database relations, stores data relations, in which some of them are primitive (i.e., layer-0) relations, whereas others are higher layer ones, obtained by generalization.

**Example 1** A real-estate database is taken as a running example to see how multiple-layer database can be constructed to facilitate the analysis and understanding of database contents, and hence the information exchange among heterogeneous databases. Suppose the database contains the following four data relations.

1. *house*(*house\_id*, *address*, *construction\_date*, *constructor*(...), *owner* (*name*, ...), *living\_room* (*length*, *width*), *bed\_room\_1* (...), ..., *surrounding\_map*, *layout*, *picture*, *video*, *listing\_price*).
2. *buyer* (*name*, *id\_#*, *birth\_date*, *education*, *income*, *work\_address*, *home\_address*, *spouse*, *children* (...), *phone*, ...).
3. *sales* (*house*, *buyer*, *agent*, *contract\_date*, *sell\_price*, *mortgage* (...), ..., *notes*).
4. *agent* (...).

These relations are layer-0 relations in the MLDB. Suppose the database contain the concept hierarchies for *geographic locations*, *occupations*, *income ranges*, etc. An MLDB can be constructed as follows.

First, the relation *house* can be generalized to a higher layer relation *house'*. The generalization can be performed, for example, as follows: (1) transform the *house construction date* to *years\_old*, e.g., from “*Sept. 10, 1980*” to 16; (2) preserve the *owner's name* but remove other information associated with the *owner*; (3) compute the *total floor area* of all the rooms and the *number of rooms* but remove the detailed specification for each room; and (4) remove some attributes: *surrounding\_map*, *layout*, *video*, etc. The generalized relation *house'* can be considered as the layer-1 information of the house, whose schema is presented as follows.

*house'*(*house\_id*, *address*, *years\_old*, *owner\_name*, *floor\_area*, *#\_of\_rooms*, ..., *picture*, *listing\_price*).

Secondly, further generalization on *house'* can be performed to produce an even higher layer relation *house''*. For example, generalization may be performed as follows: (1) remove the attributes

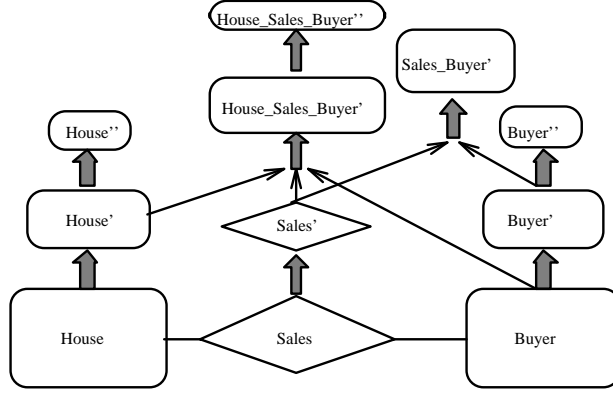


Figure 1: The route map of a real-estate DB.

*house\_id*, *owner*, *house\_picture*, etc.; (2) generalize the *address* to *areas*, such as *north\_burnaby*, *east\_vancouver*, etc.; (3) generalize *years\_old* to *year\_range*, etc.; (4) transform *#\_of\_rooms* and other associate information into *category*, such as *5-bedroom house*, *3-bedroom town-house*, etc.; and (5) merge identical tuples in the relation and store the total *count* of such merged tuples. The generalized relation *house''* could be as follows.

*house''(area, year\_range, floor\_area\_range, category, . . . , price\_range, count)*.

Similarly, *buyer* can be generalized to *buyer'*, *buyer''*, etc., which forms multiple layers of a *buyer* relation. Multiple layers can also be formed in a similar way for the relations, *sales* and *agent*.

A high layer relation can also be formed by joining two or more primitive or generalized relations. For example, *buyer\_sales'* can be produced by generalization on the join of *buyer'* and *sales'* as long as it follows the regulation(s) for the construction of MLDBs (to be presented in the next section). Similarly, one may join several relations at different layers to form new higher-layer relations. For instance, the relation *house\_sales\_buyer'* in Fig. 1 is defined by a 3-way join among *house'*, *sales'*, and *buyer'*. This shows that multiple relations at different generalization levels can participate in a join.

A possible overall MLDB structure, i.e., the schema of an MLDB, is presented in Fig. 1.

Queries can be answered efficiently and intelligently using the MLDB. For example, a user may ask for information about the houses with the price range between \$250k and \$300k. The query can be answered indirectly by first using *house''*, which may return “*none in West Vancouver, 10% in East Vancouver, 15% in South Burnaby, etc.*” Such an answer may help the user form more accurate queries to search for houses in specific regions. □

### 3 Generalization of Different Kinds of Data

An MLDB is constructed by generalization of the layer-0 (original) database. Since a database may contain different kinds of data, it is important to examine the method for generalization of each

kind of data, including unstructured and structured values, spatial and multimedia data, etc.

### 3.1 Generalization of Simple Nonnumerical Values

Simple (containing no internal structures) numerical and nonnumerical data are the most commonly used attribute values in databases. Generalization of nonnumerical values may rely on the available concept hierarchies specified by domain experts or users or implicitly stored in the database. Concept hierarchies represent necessary background knowledge which directs the generalization process. Different levels of concepts are often organized into a taxonomy of concepts. The concept taxonomy can be partially ordered according to a general-to-specific ordering. The most general concept is the null description (described by “any”), and the most specific concepts correspond to the specific values of attributes in the database. Using a concept hierarchy, the primitive data can be expressed in terms of generalized concepts in a higher layer.

A conceptual hierarchy could be given by users or experts, stored or partially stored as data in a database, specified by some generalization meta-rules, such as deleting the street number from a street address, etc., being derived from the knowledge stored elsewhere, or being computed by applying some rules or algorithms, such as deriving “*British Columbia*  $\Rightarrow$  *Western Canada*” from a geographic map stored in a spatial database, etc. Moreover, it can be defined on a single attribute or on a set of related attributes, and it can be in the shape of a balanced tree, a lattice or a general DAG. Furthermore, a given concept hierarchy can be adjusted dynamically based on the analysis of the statistical distribution of the relevant data sets [13].

### 3.2 Generalization of Simple Numerical Values

Generalization of numerical attributes can be performed similarly but in a more automatic way by the examination of data distribution characteristics [1, 9, 5]. In many cases, it may not require any predefined concept hierarchies. For example, the household income of buyers can be clustered into several groups, such as  $\{below\ 30K, 30K-50K, 50K-70K, over\ 70K\}$ , according to a relatively uniform data distribution criteria or using some statistical cluster analysis tools. Appropriate names can be assigned to the generalized numerical ranges, such as  $\{low-income, mid-range, mid-high, high\}$  by users or experts to convey more semantic meaning.

In the past 30 years, cluster analysis has been widely applied to many areas such as medicine (classification of diseases), chemistry (grouping of compounds), social studies (classification of statistical findings), and so on. The main goal is to identify structures or *clusters* present in the data. Existing clustering algorithms can be classified into two main categories: *hierarchical* methods and *partitioning* methods. Hierarchical methods are either agglomerative or divisive. Given  $n$  objects to be clustered, agglomerative methods begin with  $n$  clusters (i.e., all objects are apart). In each step, two clusters are chosen and merged. This process continues until all objects are clustered into one group. On the other hand, divisive methods begin by putting all objects in one cluster. In each

step, a cluster is chosen and split up into two. This process continues until  $n$  clusters are produced. While hierarchical methods have been successfully applied to many biological applications (e.g., for producing taxonomies of animals and plants [18]), hierarchical methods are not useful in grouping simple numerical values into ranges.

In contrast, given the number  $k$  of partitions to be found, a partitioning method tries to find the best  $k$  partitions<sup>1</sup> of the  $n$  objects. It is very often the case that the  $k$  clusters found by a partitioning method are of higher quality (i.e., more similar) than the  $k$  clusters produced by a hierarchical method. Because of this property, developing partitioning methods has been one of the main focuses of cluster analysis research. Indeed, many partitioning methods have been developed, some based on  $k$ -means, some on  $k$ -medoid, some on fuzzy analysis, etc. See [18] for a more detailed comparison of these methods. Also see [20] for the CLARANS clustering algorithm designed specifically for data mining. CLARANS and other recently developed database-oriented partitioning methods, such as [7, 28], are directly applicable to clustering numerical values into groups.

### 3.3 Generalization of Structured Data

A set-valued attribute may be of homogeneous or heterogeneous types. Typically, set-valued data can be generalized in two ways: (1) generalization of each value in a set into its corresponding higher level concepts, or (2) derivation of the general behavior of a set, such as the number of elements in the set, the types or value ranges in the set, the weighted average for numerical data, etc. Moreover, the generalization can be performed by applying different generalization operators to explore alternative generalization paths. In this case, the result of generalization is a heterogeneous set.

For example, the *hobby* of a person is a set-valued attribute which contains a set of values, such as  $\{tennis, hockey, chess, violin, nintendo\}$ , which can be generalized into a set of high level concepts, such as  $\{sports, music, video\_games\}$ , or into 5 (the number of hobbies in the set), or both, etc.

Set-valued attributes are simple structure-valued attributes. In general, a structure-valued attribute may contain sets, tuples, lists, trees, records, etc. and their combinations. Furthermore, one structure can be nested in another structure at any level. Similar to the generalization of set-valued attributes, a general structure-valued attribute can be generalized in several ways, such as (1) generalize each attribute in the structure while maintaining the shape of the structure, (2) flatten the structure and generalize the flattened structure, (3) remove the low-level structures or summarize the low-level structures by high-level concepts or aggregation, and (4) return the type or an overview of the structure.

---

<sup>1</sup>Partitions here are defined in the usual way: each object is assigned to exactly one group.

### 3.4 Aggregation and Approximation as a Means of Generalization

Besides concept tree ascension (i.e., replacing concepts by their corresponding higher level concepts in a concept hierarchy) and structured data summarization, aggregation and approximation [23] should be considered as an important means of generalization, which is especially useful for generalization of attributes with large sets of values, complex structures, spatial or multimedia data, etc.

Take spatial data as an example. It is desirable to generalize detailed geographic points into clustered regions, such as business, residential, industry, or agricultural areas, according to the land usage. Such generalization often requires the merge of a set of geographic areas by spatial operations, such as spatial union, or spatial clustering algorithms. Approximation is an important technique in such generalization. In spatial merge, it is necessary not only to merge the regions of similar types within the same general class but also to ignore some scattered regions with different types if they are unimportant to the study. For example, different pieces of land for different purposes of agricultural usage, such as vegetables, grain, fruits, etc. can be merged into one large piece of land by spatial merge. However, such an agricultural land may contain highways, houses, small stores, etc. If a majority of land is used for agriculture, the scattered spots for other purposes can be ignored, and the whole region can be claimed as an agricultural area by approximation. The spatial operators, such as *spatial\_union*, *spatial\_Overlapping*, *spatial\_Intersection*, etc., which merge scattered small regions into large, clustered regions can be considered as generalization operators in spatial aggregation and approximation.

### 3.5 Generalization of Multimedia Data

A multimedia database may contain complex text, graphics, images, maps, voice, music, and other forms of audio/video information. Such multimedia data are typically stored as sequences of bytes with variable lengths, and segments of data are linked together for easy reference. Generalization of multimedia data can be performed by recognition and extraction of the essential features and/or general patterns of such data.

For an image, the size, shape, and color of the contained objects and/or their proportional distributions in the image can be extracted by aggregation and/or approximation. For a segment of music, its melody can be summarized based on the approximate patterns that repeatedly occur in the segment and its style can be summarized based on its tone, tempo, major musical instruments played, etc. For an article, its abstract or general organization such as the table of contents, the subject and index terms frequently occurring in the article, etc. may serve as generalization results. In general, it is a challenging task to generalize multimedia data to extract the interesting knowledge implicitly stored in the data. Further research should be devoted to this issue.

## 4 Building and Maintaining MLDBs

### 4.1 Key-Preserving vs. Key-Altering Generalizations

With attribute generalization techniques available, the next important question is how to selectively perform appropriate generalizations to form useful layers of databases. In principle, there could be a large number of combinations of possible generalizations by selecting different sets of attributes to generalize and selecting the levels for the attributes to reach in the generalization. However, in practice, a few layers containing most frequently referenced attributes and patterns will be sufficient to balance the implementation efficiency and practical usage.

Frequently used attributes and patterns should be determined before generation of new layers of an MLDB by the analysis of the statistics of query history or by receiving instructions from users or experts. It is wise to remove rarely used attributes but retain frequently referenced ones in a higher layer. Similar guidelines apply when generalizing attributes to a more general concept level. For example, users may like the oldness of a house to be expressed by *ranges* (of the years since construction) such as  $\{below\_5, 6-15, 16-30, over\_30\}$  instead of the exact *construction date*, etc.

Note that finding frequently used attributes amounts to detailed bookkeeping and thresholding. For instance, the MLDB administrator may decide that there would be three layers for a particular relation, and all attributes are divided into three classes. The first class consists of attributes that are in the top-10% in terms of frequencies of accesses. The second class consists of attributes that are in the top-30%, and the third class consists of all attributes. Then the highest layer only includes attributes in the first class, the intermediate layer only attributes in the second class, and the bottom layer all attributes.

A new layer could be formed by performing generalization on one relation or on one or more joins of several relations based on the selected or frequently used attributes and patterns. Generalization is performed by removing a set of less-interested attributes, substituting the concepts in one or a set of attributes by their corresponding higher level concepts [12, 14], performing aggregation or approximation on certain attributes, etc.

Since most joins of several relations are performed on their key and/or foreign key attributes, whereas generalization may remove or generalize the key or foreign key attributes of a data relation, it is important to distinguish the following two classes of generalizations.

1. **key-preserving generalization**, in which all the key or foreign key values are preserved.
2. **key-altering generalization**, in which some key or foreign key values are generalized, and thus altered. The generalized keys should be marked explicitly since they usually cannot be used as join attributes at generating subsequent layers.

It is crucial to identify altered keys since if the altered keys were used as join attributes for joining different relations, it may generate incorrect information. This can be observed in the following example.

**Example 2** Suppose one would like to find the relationships between the ages of the houses sold and the household income level of the house buyers. Let the relations  $house'$  and  $buyer'$  contain the following tuples.

$house'(945\_Austin, \dots, 35(years\_old), \dots).$   
 $house'(58\_Austin, \dots, 4(years\_old), \dots).$   
 $buyer'(945\_Austin, mark\_Lee, 30\_40k(income), \dots).$   
 $buyer'(58\_Austin, tim\_akl, 60\_70k(income), \dots).$

Their further generalization may result in the relations  $house''$  and  $buyer''$  containing the following tuples.

$house''(Austin, \dots, over\_30(years\_old), \dots).$   
 $house''(Austin, \dots, below\_5(years\_old), \dots).$   
 $buyer''(Austin, 30\_40k(income), \dots).$   
 $buyer''(Austin, 60\_70k(income), \dots).$

If the join is performed between  $house'$  and  $buyer'$ , it still produces the correct generalized information  $house\_buyer'$ , which contains two tuples,

$house\_buyer'(945\_Austin, 35, mark\_Lee, 30\_40k, \dots).$   
 $house\_buyer'(58\_Austin, 4, tim\_akl, 60\_70k, \dots).$

and further generalization can still be performed on such a joined relation.

However, if join is performed on the altered keys between  $house''$  and  $buyer''$ , four tuples will be generated, which is obviously incorrect.

$house\_buyer''(Austin, over\_30, 30\_40k, \dots).$   
 $house\_buyer''(Austin, over\_30, 60\_70k, \dots).$   
 $house\_buyer''(Austin, below\_5, 30\_40k, \dots).$   
 $house\_buyer''(Austin, below\_5, 60\_70k, \dots).$

Clearly, joins on generalized attributes may produce more tuples than on original ones since different values in the attribute may have been generalized to identical values at a high layer.  $\square$

Notice that join on generalized attributes, though undesirable in most cases, could be useful if a join is to link the tuples with *approximately* the same attribute values together. For example, for bus transfer, the bus stops within two street blocks may be considered “approximately the same” location. Such kind of join is called an **approximate join** to be distinguished from the **precise join**. In this paper, the term *join* refers to *precise join* only. This restriction leads to the following regulation.

**Regulation 1 (Join in MLDB)** Join in an MLDB cannot be performed on the generalized attributes.

Based on this regulation, if the join in an MLDB is performed on the generalized attributes, it is called **information-loss join** (since the information could be lost by such a join). Otherwise, it is called **information-preserving join**.

## 4.2 An MLDB Construction Algorithm

Based on the previous discussion, the construction of an MLDB can be summarized into the following algorithm, which is similar to attribute-oriented generalization in knowledge discovery in databases [12, 14].

**Algorithm 1** Construction of an MLDB.

**Input:** A relational database, a set of concept hierarchies, and a set of frequently referenced attributes and frequently used query patterns.

**Output:** A multiple-layer database.

**Method.** An MLDB is constructed in the following steps.

1. Determine the multiple layers of the database based on the frequently referenced attributes and frequently used query patterns.
2. Starting with the most specific layer, generalize the relation step-by-step (using the given concept hierarchies) to form multiple-layer relations (according to the layers determined in Step 1).
3. Merge identical tuples in each generalized relation and update the *count* of the generalized tuple.
4. Construct a new schema by recording all the primitive and generalized relations, their relationships and the generalization paths.  $\square$

**Rationale of Algorithm 1.**

Step 1 indicates that the layers of an MLDB should be determined based on the frequently referenced attributes and frequently used query patterns. This is reasonable since to ensure the elegance and efficiency of an MLDB, only a small number of layers should be constructed, which should provide maximum benefits to the frequently accessed query patterns. Obviously, the frequently referenced attributes should be preserved in higher layers, and the frequently referenced concept levels of these attributes should be considered as the candidate concept levels in the construction of higher layers. Steps 2 and 3 are performed in a way similar to attribute-oriented induction, studied previously [12, 14]. Step 4 constructs a new schema which records a route map and the generalization paths for database browsing and cooperative query answering, which will be discussed in detail below.  $\square$

## 4.3 Schema: a Route Map and a Set of Generalization Paths

Since an MLDB schema provides a route map, i.e., a general structure of the MLDB for query answering and database browsing, it is important to construct a concise and information-rich schema.

Besides the schema information stored in a conventional relational database system, an MLDB schema should store two more important pieces of information.

1. A **route map**, which outlines the relationships among the relations at different layers of the database. For example, it shows which higher layer relation is generalized from one or a set of lower layer relations.
2. A set of **generalization paths**, each of which shows *how* a higher layer relation is generalized from one or a set of lower layer relations.

Similar to many extended relational databases, a *route map* can be represented by an extended E-R (entity-relationship) diagram [24], in which the entities and relationships at layer-0 (the original database) can be represented in a conventional E-R diagram [19]; whereas generalization is represented by a double-line arrow pointed from the generalizing entity (or relationship) to the generalized entity (or relationship). For example, *house'* is a higher layer entity generalized from a lower layer entity *house*, as shown in Fig. 1. Similarly, *sales\_buyer'* is a higher layer relationship, obtained by generalizing the join of *sales'* and *buyer'*. It is represented as a generalization from a relationship obtained by joining one entity and one relationship in the route map (Fig. 1). Since an extended E-R database can be easily mapped into an extended relational one [19], our discussion assumes such mappings and still adopts the terminologies from an extended relational model.

A *generalization path* is created for each high layer relation to represent how the relation is obtained in the generalization. Such a high layer relation is possibly obtained by removing a set of infrequently used attributes, preserving some attributes and/or generalizing the remaining set of attributes. Since attribute removing and preserving can be obviously observed from a relational schema, the generalization path needs only to register how a set of attributes are generalized. A generalization path consists of a set of entries, each of which contains three components:  $\langle old\_attr(s), new\_attr(s), rules \rangle$ , which tells how one or a set of old attributes is generalized into a set of new (generalized) attributes by applying some generalization rule(s), such as generalizing to which concept levels of a concept hierarchy, applying which aggregation operations, etc. If an existing hierarchy is adjusted or a new hierarchy is created in the formation of a new layer, such a hierarchy should also be registered in the *hierarchy* component of an MLDB.

#### 4.4 Maintenance of MLDBs

Since an MLDB is constructed from extracting extra-layers from an existing database by generalization, an MLDB will take more disk space than its corresponding single layer database. However, since a higher layer database is usually much smaller than the original database, query processing is expected to be more efficient if done in a higher database layer. The rapid progress of computer hardware technology has reduced the cost of disk space dramatically in the last decade. Therefore, it could be beneficial to trade disk space for intelligent and fast query answering.

In response to the updates to the original relations, the corresponding higher layers should be updated accordingly to keep the MLDB consistent. Incremental update algorithms can be used to minimize the cost of update propagation. Here we examine how to propagate incremental database updates at insertion, deletion and update of tuples in an original relation.

When a new tuple  $t$  is inserted into a relation  $R$ ,  $t$  should be generalized to  $t'$  according to the route map and be inserted into its corresponding higher layer. Such an insertion will be propagated to higher layers accordingly. However, if the generalized tuple  $t'$  is equivalent to an existing tuple in this layer, it needs only to increment the count of the existing tuple, and further propagations to higher layers will be confined to count increment as well. The deletion of a tuple from a data relation can be performed similarly.

When a tuple in a relation is updated, one can check whether the change may affect any of its high layers. If not, do nothing. Otherwise, the algorithm will be similar to the deletion of an old tuple followed by the insertion of a new one.

Although an MLDB consists of multiple layers, database updates should always be performed at the primitive database (i.e., layer-0) and the updates are then propagated to their corresponding higher layers. This is because a higher layer represents more general information, and it is impossible to transform a more general value to a more specific one, such as from *age* to *birth-date* (but it is possible in the reverse direction by applying appropriate generalization rules).

## 5 Query Answering in An MLDB

A query may involve concepts matching different layers. Moreover, one may expect that the query be answered *directly* by strictly following the request, or *intelligently* by providing some generalized, neighborhood, or associated answers.

In this section, we first examine the mechanisms for *direct* answering of queries in an MLDB and then extend the results to *cooperative* query answering.

### 5.1 Direct Query Answering

Direct query answering refers to answering queries by strictly following query specifications without providing (extra) associative information in the answers. Rigorously speaking, if all the provided and inquired information of a query are at the primitive concept level, a query can be answered directly by searching the primitive layer without exploring higher layers. However, a cooperative system should provide users with flexibility of expressing query constants and inquiries at a relatively high concept level. Such kind of “high-level” queries can be answered directly in an MLDB.

At the first glance, it seems to be easy to process such high-level queries by simply matching the constants and inquiries in the query to a corresponding layer and then directly processing the query in this layer. However, there could be dozens of attributes in a relation and each attribute may have several concept levels. It is impossible and often undesirable to construct all the possible

generalized relations whose different attributes are at different concept levels. In practice, only a small number of all the possible layers will be stored in an MLDB based on the analysis of the frequently referenced query patterns. This implies that transformations often need to be performed on some query constants to map them to a concept level corresponding to that of an existing layer database.

In principle, a high-level query constant is defined in a concept hierarchy, based on which the high-level constant can be mapped to primitive level concepts. For example, “*great vancouver area*” can be mapped to all of its composite regions, and “*big house*” can be mapped to “*total\_floor\_area*  $\geq$  3,000(*sq. ft.*)”, etc. Thus, a query can always be transformed into a primitive level query and be processed in a layer-0 database. However, to increase processing efficiency and present high-level (and more meaningful) answers, our goal is to process a query in the highest possible layer, *consistent* with all of the query constants and inquiries.

**Definition 2** A database layer  $L$  is *consistent* on an attribute  $A_i$  with a query  $q$  if the constants of attribute  $A_i$  in query  $q$  can absorb (i.e., level-wise no lower than) the concept(s) (level) of the attribute in the layer.

For example, if the query constant in query  $q$  for the attribute “*floor\_area*” is “*big*”, whereas the concept level for “*floor\_area*” in layer  $L$  is the same as “*big*”, or lower, such as “3,000-4,999”, “over\_5,000”, etc., then layer  $L$  is consistent with query  $q$  on the attribute “*floor\_area*”.

**Definition 3** The *watermark* of a (nonjoin) attribute  $A_i$  for query  $q$  is the topmost database layer which is consistent with the concept level of query constants/inquiries of attribute  $A_i$  in query  $q$ .

**Lemma 1** *All the layers lower than the watermark of an attribute  $A_i$  for query  $q$  must be consistent with the values of attribute  $A_i$  in query  $q$ .*

This lemma is a property immediately following from definitions 2 and 3.

We first examine the case that a query references only one generalized relation and all the high level query constants are nonnumerical values.

**Proposition 1** *If a query  $q$  references only one generalized relation and no other relations, and all the high level query constants are nominal (nonnumerical) values, the highest possible layer consistent with the query should be the lowest watermark of all the participant attributes of  $q$  in the route map of the MLDB.*

**Rationale.** Suppose layer  $L$  is the lowest watermark of all the participant attributes of  $q$  in the route map of the MLDB. Since a layer no higher than the watermark of attribute  $A_i$  must be consistent with the corresponding query constant/inquiry on attribute  $A_i$ ,  $L$  must be consistent with all the constants and inquiries of all the participant attributes of query  $q$ . Furthermore, since a watermark for an attribute is the highest possible database layer for such an attribute, the layer so derived

must be the highest possible layer which is consistent with all the participating attributes in the query.  $\square$

Next, we examine the case of queries involving join(s) of two or more relations. If such a join or its lower layer is already stored in the MLDB by an information-preserving join, the judgement should be the same as the case for single relations. However, if no such a join has been performed and stored as a new layer in the MLDB, the watermark of such a join attribute must be the highest database layer in which generalization has not been performed on this attribute (i.e., on which the information-preserving join can be performed). This is because join cannot be performed on the generalized attributes according to Regulation 1.

**Definition 4** The **watermark** of a join attribute  $A_i$  for query  $q$  is the topmost database layer which is consistent with the concept level of query constants/inquiries of attribute  $A_i$  in query  $q$  and in which the information-preserving join can be performed on  $A_i$ .

Thus, we have the following proposition.

**Proposition 2** *If a query  $q$  involves join(s) of two or more relations, and all the high level query constants are nominal (nonnumerical) constants, the highest possible layer consistent with the query should be the lowest watermark of all the participant attributes (including the join attributes) of  $q$  in the route map of the MLDB.*

**Rationale.** Based on the definition of the watermark of a join attribute in a query, and the similar reasoning in the rationale for Proposition 1, it naturally leads to the assertion for a query involving one join. By induction, one can easily show that the proposition holds for queries involving more than one join.  $\square$

**Example 3** Suppose the query on the real-estate MLDB is to describe the relationship between *house* and *sales* with the following given information: *located in north-vancouver, 3-bedroom house, and sold in the summer of 1993*. Moreover, suppose the route map of an MLDB corresponding to this query is shown in Fig. 2.

The query involves a join of *sales* and *house* and the provided query constants are all at the levels high enough to match those in *house''* and *sales''*. However, a join cannot be performed at these two high layer relations since the join attributes of *house''* and *sales''* have been generalized (with their join keys altered). The watermarks of the join attributes, *house.location* and *sales.house\_loc*, are one layer lower than their topmost layers.

If there exists a relation such as *house\_sales* in the MLDB, which represents the join between the two relations and/or their further generalizations, the query can be processed within such a layer. Otherwise (as shown in Fig. 2), join must be performed on the highest joinable layers (which should be *house'* and *sales'*, as shown in Fig. 2). Then further generalization can be performed on this joined relation to form appropriate answers.  $\square$

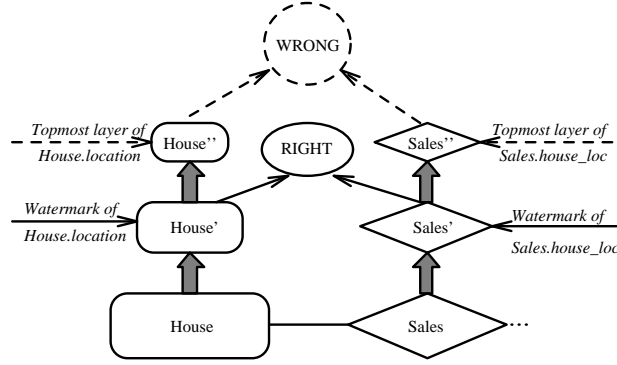


Figure 2: Perform joins at the appropriate layers.

Finally, we examine the determination of the highest possible database layers if the query contains numeric attributes. If the value in a numeric attribute in the query is expressed as a generalized constant, such as “*expensive*”, or the specified range in the query has an exact match with some (generalized) range in a concept hierarchy, such as “*\$300-400k*”, the numeric value can be treated the same as a nonnumeric concept. Otherwise, we have two choices: (1) set the watermark of the attribute to the highest layer in which such numeric attributes has not been generalized, or (2) relax the requirement of the preciseness of the query answering. In later case, the appropriate layer is first determined by nonnumeric attributes. A coverage test is then performed to see whether the generalized range is entirely covered by the range provided in the query. For the entirely covered (generalized) ranges, the precision of the answer remains the same. However, for that which only partially covers a range, the answer should be associated with certain probability (e.g., by assuming that the data are relatively uniformly distributed within the generalized range), or be associated with a necessary explanation to clarify that the answer may not match the exact query condition but cover the entire generalized range.

**Example 4** Suppose the query on the real-estate database is to describe the big houses in north-vancouver ranging in price from \$280k to \$350k. Since the query is to *describe* houses (not to find exact houses), the inquired portion can be considered at a high layer, matching any layers located by its query constants. To find the layer of its query constants, we have “*floor\_area = big*”, “*address = north-vancouver*”, and “*price\_range = \$280k-\$350k*”. The watermarks of the first two are at the layer *house''*, whereas the third one is a range value. Suppose in the layer *house''*, the generalized tuples may have the ranges like \$250k-\$300k, \$300k-\$350k, etc., which do not have the exact match of the range \$280k-\$350k. Still, the query can be processed at this layer, with the information within the range \$300k-\$350k returned without additional explanation, but with the information within the range \$250k-\$300k returned, associated with an explanation that the returned information is for the range of \$250k-\$300k instead of \$280k-\$300k to avoid misunderstanding. □

## 5.2 Cooperative Query Answering

Since an MLDB stores general database information in higher layers, many techniques investigated in previous researches into cooperative query answering in (single-layer) databases [17, 3, 2, 10, 16] can be extended effectively to cooperative query answering in MLDBs.

The following reasoning may convince us that an MLDB may greatly facilitate cooperative query answering.

Many cooperative query answering techniques need certain kinds of generalization [2, 11]; whereas different kinds of frequently used generalizations are performed and stored in the high layers of an MLDB. Also, they often need to compare the “neighborhood” information [3, 2]. The generalized neighborhood tuples are usually stored in the same high layer relations, ready for comparison and investigation. Moreover, they often need to summarize answer-related information, associated with data statistics or certain aggregations [2, 26]. Interestingly, a higher layer relation not only presents the generalized tuples but also the *counts* of the identical tuples or other computed aggregation values (such as sum, average, etc.). Such high-level information with counts conveys important information for data summarization and statistical data investigation.

Furthermore, since the layer selection in the construction of an MLDB is based on the study of the frequently referenced attributes and frequently used query patterns, the MLDB itself embodies rich information about the history of the most regular query patterns and also implies the potential intent of the database users. It forms a rich source for query intent analysis and plays the role of confining the cooperative answers to frequently referenced patterns automatically.

Finally, an MLDB constructs a set of layers step-by-step, from most specific data to more general information. It facilitates progressive query refinement, from general information browsing to specific data retrieval. Such a process represents a top-down information searching process, which matches human’s reasoning and learning process naturally, thus provides a cooperative process for step-by-step information exploration [25, 27].

**Example 5** A query like “*what kind of houses can be bought with \$300k in the Vancouver area?*” can be answered using an MLDB efficiently and effectively. Here we examine several ways to answer this query using the MLDB constructed in Example 1.

1. Relaxation of query conditions using concept hierarchies and/or high layer relations:

Instead of answering the query using “*house\_price = \$300k*”, the condition can be relaxed to *about \$300k*, that is, the price range covering \$300k in a high layer relation, such as *house*”, can be used for query answering. This kind of relaxation can be done by mapping query constants up or down using concept hierarchies, and once the query is mapped to a level which fits a corresponding database layer, it can be processed within the layer.

2. Generalized answers with summarized statistics:

Instead of printing thousands of houses within this price range, it searches through the top layer *house* relation, such as *house''*, and print the generalized answer, such as “20% 20-30 years-old, medium-sized, 3\_bedrooms house in East Vancouver, ...”. With the availability of MLDBs, such kind of generalized answers can be obtained directly from a high layer DB by summarization of the answers (such as presenting percentage, general view, etc.) at a high layer.

### 3. Comparison with the neighborhood answers:

Furthermore, the printed general answer can be compared with its neighborhood answers using the same top-level relation, such as “10% 3-bedroom 20-30 years-old houses in the Central Vancouver priced between \$250k to \$350K, while 30% such houses priced between \$350 to \$500k, ...”. Notice that such comparison information can be presented as concise tables using an existing high layer relation.

### 4. Query answering with associative information:

It is often desirable to provide some “extra” information associated with a set of answers in cooperative query answering. Query answering with associative information can be easily achieved using high layer data relations. For example, the query can be answered by printing houses with different price ranges (such as \$230-280k, \$330-380k, etc.) as *row extension*, or printing houses in neighboring cities, printing other interesting features as *column extension*, or printing sales information related to such houses as *table extension*. These can be performed using high layer relations.

### 5. Progressively query refinement or progressive information focusing:

The query can be answered by progressively stepping down the layers to find more detailed information. The top layer is often examined first, with general data and global views presented. Such a presentation often give a user better idea on what should be searched further with additional constraints. For example, a user may focus the search to East Vancouver area after s(he) finds a high percentage of the houses within this price range since it is likely to find a satisfiable house within this area. Such a further inquiry may lead the search to lower layer relations and may also promote users to pose more restricted constraints or refine the original ones. In this case, the route map associated with the MLDB will act as a tour guide to locate related lower layer relation(s). □

## 6 MLDBs for Heterogeneous Database Systems

A major challenge to the interoperability of a heterogeneous database system is the semantic heterogeneity of multiple, autonomous information systems. Since each information system has its own

regulations and its own ways to specify its data and rules, it may cause ambiguity and incompatibility problems when interoperating multiple database systems.

The interoperability of multiple information systems should be taken as a major concern in the design and construction of the MLDB for a heterogeneous database system. Although such a heterogeneous system may still allow each component system to have its own independent multiple-layer relations, higher layer relations with its sub-schemas and concepts shared among different components should be constructed systematically.

The construction of a shared, cooperative MLDB may involve the negotiation and standardization of higher layer concepts and schemas for multiple components. The result of such negotiation and standardization may lead to an agreement on a minimum information consistent layer, called the *minimum cooperative layer*, in which the schema is a commonly agreed one and each attribute contains (*generalized* or *transformed*) concepts (or values) agreed by every component system. Each component system should specify the rules of mapping from a certain layer of their MLDB to this minimum cooperative layer. Higher layers constructed on top of this minimum cooperative layer will be shared among the components.

According to this architecture, a heterogeneous database system consists of a shared, cooperative MLDB for the whole system; each component may have its own MLDB; and the minimum cooperative layer is the interface layer between the MLDB of the whole system and the component MLDB. Queries on the whole system can be answered by referencing first the MLDB of the system and, when more detailed information is needed, mapping the query from the minimum cooperative layer to the corresponding component MLDB.

Notice that the minimum cooperative layer and the layers above, even constructed for the whole system, may still belong to each component database and be stored in their corresponding sites. Alternatively, they can also be stored in multiple copies, e.g., one copy at each site, if the size of the high layer relations is not so large, to reduce the cost of network transmission. In either case, if the query involves only frequently asked items or high level information, the data transmission across the network can be reduced substantially using the architecture of MLDBs.

Moreover, although the MLDB architecture may facilitate query transformation among different sites, cooperative query answering techniques should often be used for answering queries against the shared heterogeneous MLDBs instead of returning the primitive level data or the data in the nonshared portion of the heterogeneous MLDBs. This is because primitive level data or nonshared generalized data may carry different data semantics from different component databases which may be misinterpreted or misunderstood by users.

**Example 6** Let us examine how to cooperate heterogeneous university databases for graduate admission, the example illustrated in the introduction section.

Let the federated schema be worked out as follows, based on schema transformation and analysis.

*grading*(*name*, *student\_id*, *semester*, *year*, *course\_num*, *department*, *university*, . . . , *grade*),

Taking the primitive level data in each database as layer-0 database, a layer-1 relation *grading*, viewed as the minimum cooperative layer, can be constructed by generalizing data in a set of shared attributes into concepts which are cooperative in heterogeneous databases. For example, “*semester, year*” can be generalized to *semester* in the form like 932 (i.e., the 2nd semester of 1993). Similarly “*course\_num*” can be generalized to some generic, summarative course\_information, such as DB1 (first DB course), OS2 (second OS course), etc., and *grades* into grade distributions, such as top-5%, etc. Such transformations make database contents exchangeable among different component databases.

The formation of this minimum cooperative layer may involve the negotiation and standardization of higher layer concepts and schemas among multiple components. Each component system should specify the rules of mapping from a certain layer of their MLDB to this minimum cooperative layer. In this example, rules may work out for each component database on how to transform course numbers, grades, etc. to commonly sharable course names, grade distributions, etc.

Name	student_id	semester	course_num	department	university	...	grade
Jane Doe	93350924	932	DB1	BU	SFU	...	top-2%
John Smith	85140298	923	OS2	CS	SFU	...	top-40%
...	...	...	...	...	...	...	...

Name	student_id	semester	course_num	department	university	...	grade
Sam Carey	941CS0135	953	Java	CS	U.B.C.	...	top-15%
...	...	...	...	...	...	...	...

Name	student_id	semester	course_num	department	university	...	grade
Tom Hardy	3H702953	951	C++	CIS	OSU	...	top-30%
...	...	...	...	...	...	...	...

Table 1: A set of cooperative layer-1 relations *grading* from multiple university databases

One example of such transformation is shown in Table 1, which results in a set of layer-1 relations. Notice at this layer, contents of some attributes such as *semester, course\_num, department, grade*, etc. have been generalized and transformed into cooperative concepts. However, not every attribute in such a higher layer relation will have their contents generalized uniformly. For example, values in the attributes *name* and *student\_id* remain unchanged. Such treatment at the construction of minimal cooperative layer is important since such retained values will be useful in many applications, such as, evaluation of individual applicants.

Higher layers constructed on top of this minimum cooperative layer will be shared among the components. Such higher layers may provide high-level, generalized views of the database contents.

An interesting point is that when a user at one university poses a query about the applicants from other universities, it is often preferable not to return detailed primitive data but to return data at the minimum cooperative layer or higher. This is because the primitive level data, such as a course number or a grading system in another university may not be comprehensible by users at

a different institution. Therefore, cooperative query answering at different layers discussed in the last sections may become a valuable tool for cooperating heterogeneous information systems.  $\square$

## 7 Conclusions

In this chapter, a major challenge for building cooperative information systems, the semantic heterogeneity problem, is analyzed, which shows that schema level analysis, though popularly performed in current studies, may not be sufficient to solve the problem, and the data level analysis, i.e., the analysis of database contents, should be explored to ease the semantic heterogeneity problem in cooperative information systems. In particular, a multiple-layer database (MLDB) model is proposed and studied, which is useful in cooperative query answering, database browsing, query optimization, and cooperative heterogeneous databases.

Our study shows that data mining may provide a powerful tool at construction of multiple-layer databases since it facilitates the transformation of low-level heterogeneous data into high-level homogeneous information. Data generalization and layer construction methods have been developed in this study to ensure that new layers can be constructed efficiently, effectively, and consistent with the primitive information stored in the database.

The necessity of data mining in cooperative information systems and the techniques for data mining are studied, including the methods for data generalization, summarization, and the construction and maintenance of multiple-layer databases. Direct and cooperative query answering in such an MLDB and the methods for cooperating heterogeneous databases are studied with the implementation techniques examined and the benefits and limitations analyzed.

Currently, we are further investigating the methods for construction of MLDBs for heterogeneous databases. New techniques, implementations, and performance studies for the construction of MLDBs for heterogeneous databases will be reported in the future.

## References

- [1] R. Agrawal, S. Ghosh, T. Imielinski, B. Iyer, and A. Swami. An interval classifier for database mining applications. In *Proc. 18th Int. Conf. Very Large Data Bases*, pages 560–573, Vancouver, Canada, August 1992.
- [2] W. W. Chu and Q. Chen. Neighborhood and associative query answering. *Journal of Intelligent Information Systems*, 1:355–382, 1992.
- [3] F. Cuppens and R. Demolombe. Extending answers to neighbor entities in a cooperative answering context. *Decision Support Systems*, 7:1–11, 1991.

- [4] S. Dao and B. Perry. Applying a data miner to heterogeneous schema integration. In *Proc. First Int. Conf. on Knowledge Discovery and Data Mining*, pages 63–68, Montreal, Canada, Aug. 1995.
- [5] B. de Ville. Applying statistical knowledge to database analysis and knowledge base construction. In *Proc. 6th Conference on Artificial Intelligence Applications*, pages 30–36, Santa Barbara, CA, 1990.
- [6] A. Elmagarmid and C. Pu (editors). Special issue on heterogeneous databases. *ACM Computing Survey*, 22, 1990.
- [7] M. Ester, H.-P. Kriegel, and X. Xu. Knowledge discovery in large spatial databases: Focusing techniques for efficient class identification. In *Proc. 4th Int. Symp. on Large Spatial Databases (SSD'95)*, pages 67–82, Portland, Maine, August 1995.
- [8] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996.
- [9] D. Fisher. Improving inference through conceptual clustering. In *Proc. 1987 AAAI Conf.*, pages 461–465, Seattle, Washington, July 1987.
- [10] T. Gaasterland. Restricting query relaxation through user constraints. In *Proc. 1st Int. Conf. Cooperative Information Systems*, pages 359–366, Toronto, Canada, 1993.
- [11] T. Gaasterland, P. Godfrey, and J. Minker. Relaxation as a platform for cooperative answering. *Journal of Intelligent Information Systems*, 1:293–321, 1992.
- [12] J. Han, Y. Cai, and N. Cercone. Data-driven discovery of quantitative rules in relational databases. *IEEE Trans. Knowledge and Data Engineering*, 5:29–40, 1993.
- [13] J. Han and Y. Fu. Dynamic generation and refinement of concept hierarchies for knowledge discovery in databases. In *Proc. AAAI'94 Workshop on Knowledge Discovery in Databases (KDD'94)*, pages 157–168, Seattle, WA, July 1994.
- [14] J. Han and Y. Fu. Exploration of the power of attribute-oriented induction in data mining. In U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 399–421. AAAI/MIT Press, 1996.
- [15] J. Han, Y. Fu, W. Wang, J. Chiang, W. Gong, K. Koperski, D. Li, Y. Lu, A. Rajan, N. Stefanovic, B. Xia, and O. R. Zaiane. DBMiner: A system for mining knowledge in large relational databases. In *Proc. 1996 Int'l Conf. on Data Mining and Knowledge Discovery (KDD'96)*, pages 250–255, Portland, Oregon, August 1996.
- [16] J. Han, Y. Huang, N. Cercone, and Y. Fu. Intelligent query answering by knowledge discovery techniques. *IEEE Trans. Knowledge and Data Engineering*, 8:373–390, 1996.

- [17] T. Imielinski. Intelligent query answering in rule based systems. *J. Logic Programming*, 4:229–257, 1987.
- [18] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: an Introduction to Cluster Analysis*. John Wiley & Sons, 1990.
- [19] H. F. Korth and A. Silberschatz. *Database System Concepts, 2ed*. McGraw-Hill, 1991.
- [20] R. Ng and J. Han. Efficient and effective clustering method for spatial data mining. In *Proc. 1994 Int. Conf. Very Large Data Bases*, pages 144–155, Santiago, Chile, September 1994.
- [21] H. Schek, A. Sheth, and B. Czejdo (editors). Interoperability in multidatabase systems. In *Proc. Third International Workshop on Research Issues in Data Engineering*, April 1993.
- [22] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22:183–236, 1990.
- [23] C. Shum and R. Muntz. An information-theoretic study on aggregate responses. In *Proc. 14th Int. Conf. Very Large Data Bases*, pages 479–490, Los Angeles, USA, August 1988.
- [24] T. J. Teorey, D. Yang, and J. P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Comput. Surv.*, 18:197–222, 1986.
- [25] S.V. Vrbsky and J. W. S. Liu. An object-oriented query processor that returns monotonically improving answers. In *Proc. 7th IEEE Conf. on Data Engineering*, pages 472–481, Kobe, Japan, April 1991.
- [26] C. Wittemann and H. Kunst. Intelligent assistance in flexible decisions. In *Proc. 1st Int. Conf. Cooperative Information Systems*, pages 377–381, Toronto, Canada, 1993.
- [27] M.F. Wolf. Successful integration of databases, knowledge-based systems, and human judgement. In *Proc. 1st Int. Conf. Cooperative Information Systems*, pages 154–162, Toronto, Canada, 1993.
- [28] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: an efficient data clustering method for very large databases. In *Proc. 1996 ACM-SIGMOD Int. Conf. Management of Data*, pages 103–114, Montreal, Canada, June 1996.