

# Cooperative Query Answering Using Multiple Layered Databases

*Research*

Jiawei Han\*      Yongjian Fu  
School of Computing Science  
Simon Fraser University  
Burnaby, B.C., Canada V5A 1S6  
{han, yongjian}@cs.sfu.ca

Raymond T. Ng†  
Department of Computer Science  
University of British Columbia  
Vancouver, B.C., Canada  
rng@cs.ubc.ca

## Abstract

*How can a real-estate agent respond to inquiries quickly and intelligently? The ‘trick’ could be using a simple table to briefly outline the general information and a complete book to reference the details. Such a method can be generalized to the construction of a multiple layered database (MLDB), a useful database organization technique for cooperative query answering, database browsing, query optimization and querying cooperative information systems. In this paper, we study the construction of MLDBs using generalization and knowledge discovery techniques and the application of MLDBs to cooperative/intelligent query answering in database systems.*

## 1 Introduction

Cooperative (or intelligent) query answering refers to a mechanism which answers queries cooperatively and intelligently by analyzing the intent of a query and providing some generalized, neighborhood, or associated answers [5, 11, 2]. Many interesting techniques [14, 6, 5, 10, 13, 15] have been developed for cooperative query answering, by integration of the methods developed in several related fields, such as semantic data modeling [3],

\*Research partially supported by the Natural Sciences and Engineering Research Council of Canada under grant OGP0037230 and the Centre for Systems Science of Simon Fraser University.

†Research partially supported by the Natural Sciences and Engineering Research Council of Canada under grants OGP0138055 and STR0134419.

deductive databases [14], knowledge discovery in databases [4, 15], etc. Cooperative query answering can be realized by generalization and summarization of answers, explanation of answers or returning intensional answers, query rewriting using associated or neighborhood information, comparison of answers with those of similar queries, etc.

In this paper, we propose a new technique: *the construction and application of a multiple layered database*, and explore its potential and effectiveness in cooperative query answering. A **multiple layered database (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.

We have the following motivations to promote the idea of *multiple layered databases*.

First, with the wide availability of database systems and rapid progress of information technologies, a database may store a huge set of data objects with complex structures. A large set of data objects may be organized in classes and class-subclass hierarchies and may contain complex structured or unstructured subobjects, texts, images, and spatial or multimedia data. Moreover, the data may be distributed to different sites and be stored in heterogeneous *multi*-databases. Queries on such kind of databases could be costly to process. An MLDB system may preprocess and generalize some primitive data, resolve certain semantic ambiguities of heterogeneous data, and store the preprocessed data at a more general concept layer, which may facilitate high-level querying and reduce the cost of query processing [19, 22, 18].

Secondly, a database user may not be familiar

with a database schema, a query language, or specific data constraints. It is likely that such a user may pose queries which are not exactly what (s)he wants to know. Such kind of queries are better to be treated as information probes and be answered by providing general or associated information with data distribution statistics, which may help users understand the data better and form more accurate queries [5, 2, 6]. In an MLDB system, probe queries can be mapped to a relatively higher concept layer and be processed in such a layer. Such answers may provide associative and summary information and assist users to refine their queries.

Thirdly, an MLDB may provide a global view of the current contents in a database with summary statistics. It is a natural resource to assist users to browse database contents, pose progressively refined queries, and perform knowledge discovery in databases. Some users may even be satisfied with the examination of the general or abstract data with associated statistical information in a high layer instead of examining the concrete data in every detailed level.

Finally, schema-directed semantic query optimization can be performed in an MLDB. A higher layer database, storing more general and abstract information, could be much smaller than its corresponding lower layer one. Thus, it is faster and less costly to retrieve data in a higher layer database. Moreover, since an MLDB provides statistical information of database contents in the higher layers, it may provide guided assistance for query processing and query optimization of its lower level counterparts.

In this paper, we propose a model for an MLDB and study how to construct and maintain an MLDB and how to perform cooperative query answering using MLDBs.

The paper is organized as follows. In Section 2, the concept of MLDB is introduced. The techniques for construction and maintenance of an MLDB are studied in Section 3. Cooperative query answering using MLDBs is investigated in Section 4. In Section 5, the role of MLDBs in cooperative information systems is briefly discussed. The study is summarized 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., and to other kinds of databases, such as distributed and heterogeneous databases.

**Definition 1** A multiple layered database (MLDB) consists of 4 major components:  $\langle S, H, C, D \rangle$ , defined as follows.

1. *S*: a database schema, which contains the meta-information about the layered 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 layered database.  $\square$

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.

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 concept 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** Suppose a real-estate 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*, *house\_layout*, *house\_picture*, *house\_video*, *listing\_price*).

2. *customer* (*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 contains 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 layered 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 14; (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, house\_layout, house\_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, ..., house\_picture, listing\_price*).

Secondly, further generalization on *house'* can be performed to produce an even higher layered relation *house''*. For example, generalization may be performed as follows: (1) remove the attributes *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, *customer* can be generalized to *customer', customer''*, etc., which forms multiple layers of a *customer* relation. Multiple layers can also be formed in a similar way for the relations, *sales* and *agent*.

A higher layered relation can also be formed by joining two or more primitive or generalized relations. For example, *customer\_sales'* can be produced by generalization on the join of *customer'*

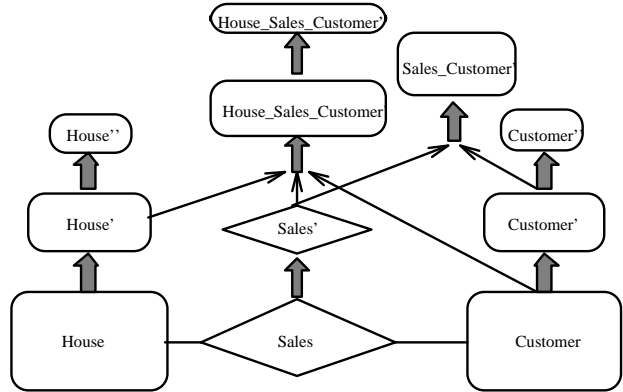


Figure 1: The route map of a real-estate DB.

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-layered relations, such as *house\_sales\_customer'*, etc.

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 the information about the houses with the price range between \$250k and \$300k. The query can be answered intelligently 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 Building MLDBs

An MLDB is constructed by generalization of the layer-0 (original) database. Since a database may contain different kinds of complex 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 [17, 8].

### 3.1 Generalization of different kinds of data

#### 3.1.1 Generalization on simple values

Simple (containing no internal structures) numerical and nonnumerical data are the most popularly encountered attribute values in databases.

Generalization on 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. Also, it could be defined on a single attribute or on a set of related attributes, and it could 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.

Generalization on numerical attributes can be performed similarly but in a more automatic way by the examination of data distribution characteristics [1, 9, 7]. In many cases, it may not require any predefined concept hierarchies. For example, the household income of customers 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 clustering 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.

### 3.1.2 Generalization on structured data

Complex structure-valued data, such as set-valued and list-valued data and data with nested structures, can be generalized in several ways in order to be interesting.

A set-valued attribute may be of homogeneous or heterogeneous types. Typically, a 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.

### 3.1.3 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 [20] 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.

## 3.2 Construction of MLDB

### 3.2.1 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 the *ranges* (of the construction years) such as  $\{below\_5, 6-15, 16-30, over\_30\}$  instead of the exact *construction date*, etc.

A new layer could be formed by performing generalization on one relation or on a join of several relations based on the selected, 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 [4], 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 keys at generating subsequent layers.

It is crucial to identify altered keys since if the altered keys were used to perform joins of different relations, it may generate incorrect information. This is 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 *sc'* (*sales\_customer'*) contain the following tuples.

```
house'(945_Austin, ..., 35(years_old), ...).
house'(58_Austin, ..., 4(years_old), ...).
sc'(945_Austin, mark_lee, 30_40k(income), ...).
sc'(58_Austin, tim_akl, 60_70k(income), ...).
```

Their further generalization may result in the relations *house''* and *sc''* (*sales\_customer''*) containing the following tuples.

```
house''(Austin, ..., over_30(years_old), ...).
house''(Austin, ..., below_5(years_old), ...).
sc''(Austin, 30_40k(income), ...).
sc''(Austin, 60_70k(income), ...).
```

If the join is performed between *house'* and *sc'*, it still produces the correct generalized information *hc'* (*house\_customer'*), which contains two tuples as shown below, and further generalization can still be performed on such a joined relation.

```
hc'(945_Austin, 35, mark_lee, 30_40k, ...).
hc'(58_Austin, 4, tim_akl, 60_70k, ...).
```

However, if join is performed on the altered keys between *house''* and *sc''*, 4 tuples will be generated, which is obviously incorrect. 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 the 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*.

### 3.2.2 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 [4, 12].

**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 layered 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 layered 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 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 the attribute-oriented induction, studied previously [12, 4]. 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.  $\square$

### 3.3 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 layered relation is generalized from one or a set of lower layered relations.
2. A set of *generalization paths*, each of which shows *how* a higher layered relation is generalized from one or a set of lower layered relations.

Similar to many extended relational databases, a *route map* can be represented by an extended E-R (entity-relationship) diagram [21], in which the entities and relationships at layer-0 (the original database) can be represented in a conventional E-R diagram [16]; 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 layered entity generalized from a lower layer entity *house*, as shown in Fig. 1. Similarly, *sales\_customer'* is a higher layered relationship, obtained by generalizing the join of *sales'* and *customer'*. 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 [16], 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.

### 3.4 Maintenance of MLDBs

Since an MLDB is resulted from extracting extra-layers from an existing database by generalization, an MLDB will take more disk space than its corresponding single layered 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 more 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).

## 4 Cooperative query answering in an MLDB

A query consists of (*query constants*) and inquired information. Both of them 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.

### 4.1 Direct query answering in an MLDB

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 inquires 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 rela-

tions 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 layered 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 > 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 higher 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 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 lower 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.  $\square$

Then 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 a join 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.*

**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, join cannot be performed at these two high layer relations since the join attributes of

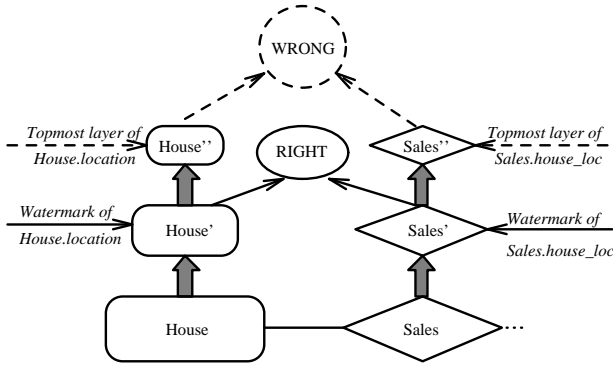


Figure 2: Perform joins at the appropriate layers.

*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. □

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 the range 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 occupies only a portion of the entire generalized range.

**Example 4** Suppose the query on the real-estate database is to describe the big houses in north-vancouver with the price ranged 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. □

## 4.2 Cooperative query answering in an MLDB

Since an MLDB stores general database information in higher layers, many techniques investigated in previous researches on cooperative query answering in (single layered) databases [14, 6, 5, 10, 13] can be extended to cooperative query answering in MLDBs, easily, effectively and efficiently.

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 [5, 11]; whereas different kinds of frequently used generalizations are performed and stored in the higher layers of an MLDB. Also, they often need to compare the “neighborhood” information [6, 5]. The generalized neighborhood tuples are usually stored in the same higher layer relations, ready for comparison and investigation. Moreover, they often need to summarize answer-related information, associated with data statistics or certain aggregations [5, 23]. Interestingly, a higher-layered 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 [22, 24].

Clearly, with these advantages, MLDB may become a valuable tool in cooperative query answering.

Since the cooperative query answering has been studied relatively thoroughly in previous research, instead of “reinventing” the technologies of cooperative query answering, we briefly present some examples to illustrate the use of MLDBs in the implementation of cooperative query answering mechanisms.

**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 layered DB by summarization of the answers (such as giving 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). □

## 5 MLDBs in Cooperative Information Systems

The discussions in the previous sections were focused on the construction and application of MLDBs in *individual* database systems. Actually, the construction of MLDBs will benefit greatly the interoperation of *cooperative information systems* (CISs).

A major challenge to the interoperability of a CIS is the semantic heterogeneity of multiple, au-

onomous 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 information systems.

The interoperability of multiple information systems should be taken as a major concern in the design and construction of the MLDB for a CIS. Although a CIS may still allow each component system to have its own independent multiple layered relations, higher layered relations with its subschemas and concepts shared among component systems in the CIS 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 component information systems in a CIS. The result of such negotiation and standardization may lead to an agreement on a minimum information consistent layer, called the *minimum cooperative layer* of the CIS, in which the schema is a commonly agreed one and each attribute contains (generalized or transformed) concepts (or values) agreed by each component. Each component system should specify the rules of mapping from a certain layer (which could be the primitive layer) of their MLDB to this minimum cooperative layer. Higher layers constructed on top of this minimum cooperative layer will be shared and cooperated among the component systems in the CIS.

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

**Example 6** Suppose a real-estate cooperative information system consists of several real estate databases from several agencies: *Royal LePage*, *Sutton Group*, *Century 21*, *Realty World*, etc. The information at the primitive level of these databases may have incompatible standards. For example, the room size could be specified by each dimension in inches, feet, or meters or by the floor area in square feet or square meters. The total floor area may include bath-rooms and kitchen in some databases but exclude them in others.

To communicate among different databases and

answer (global) queries, the minimum cooperative layer should be built, which maps different standards into a commonly accepted one based on the rules and methods specified for each component database. Once this layer is constructed, the databases can communicate each other with consistent semantics and can answer queries involving the information at different layers.  $\square$

Notice that the minimum cooperative layer and the layers above, even constructed for the whole CIS, 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 higher 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 higher level information, the data transmission across the network can be reduced substantially using the architecture of MLDBs.

## 6 Conclusions

A multiple layered database (MLDB) model is proposed and examined in this study, which demonstrates the usefulness of an MLDB in cooperative query answering, database browsing and query optimization.

An MLDB can be constructed by data generalization and knowledge discovery techniques. Data generalization and layer construction methods have been developed in this study to guarantee new layers can be constructed efficiently, effectively and consistent with the primitive information stored in the database. Direct and cooperative query answering in such a MLDB are studied with the implementation techniques examined and the benefits and limitations analyzed.

Although there have been some studies on the construction of multiple resolution databases [19], more studies are needed in the construction and utilization of multiple layered databases in cooperative query answering. We plan to perform a more detailed performance study and quantitative analysis of the methods related to the construction, maintenance and application of MLDBs and report our further investigations 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] P. Bosc and O. Pivert. Some approaches for relational databases flexible querying. *Journal of Intelligent Information Systems*, 1:323–354, 1992.
  - [3] M. Brodie, J. Mylopoulos, and J. Schmidt. *On Conceptual Modeling*. Springer-Verlag, 1984.
  - [4] Y. Cai, N. Cercone, and J. Han. Attribute-oriented induction in relational databases. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 213–228. AAAI/MIT Press, 1991.
  - [5] W. W. Chu and Q. Chen. Neighborhood and associative query answering. *Journal of Intelligent Information Systems*, 1:355–382, 1992.
  - [6] F. Cuppens and R. Demolombe. Extending answers to neighbor entities in a cooperative answering context. *Decision Support Systems*, 7:1–11, 1991.
  - [7] 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.
  - [8] T. Duong and J. Hiller. Modelling the real world by multi-world data model. In *Proc. Int. Conf. Intelligent and Cooperative Information System*, pages 279–290, 1993.
  - [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. Int. Conf. Intelligent and Cooperative Information System*, pages 359–366, 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. Knowledge discovery in databases: An attribute-oriented approach. In *Proc. 18th Int. Conf. Very Large Data Bases*, pages 547–559, Vancouver, Canada, August 1992.
  - [13] J. Han, Y. Huang, and N. Cercone. Intelligent query answering by knowledge discovery techniques. In *submitted to IEEE Trans. Knowledge and Data Engineering*, 1993.
  - [14] T. Imielinski. Intelligent query answering in rule based systems. *J. Logic Programming*, 4:229–257, 1987.
  - [15] K. A. Kaufman, R. S. Michalski, and L. Kerschberg. Mining for knowledge in databases: Goals and general description of the INLEN system. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 449–462. AAAI/MIT Press, 1991.
  - [16] H. F. Korth and A. Silberschatz. *Database System Concepts*, 2ed. McGraw-Hill, 1991.
  - [17] M. Manago and Y. Kodratoff. Induction of decision trees from complex structured data. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 289–306. AAAI/MIT Press, 1991.
  - [18] J. Mylopoulos, V. K. Chaudhri, D. Plexousakis, and T. Topaloglou. Adapting database implementation techniques to managing very large knowledge bases. In *Proc. Int. Conf. Building and Sharing of Very Large-Scale Knowledge Bases '93*, pages 215–224, Tokyo, Japan, December 1993.
  - [19] R.L. Read, D.S. Fussell, and A. Silberschatz. A multi-resolution relational data model. In *Proc. 18th Int. Conf. Very Large Data Bases*, pages 139–150, Vancouver, Canada, Aug. 1992.
  - [20] C. Shum and R. Muntz. An information-theoretic study on aggregate responses. In F. Bancilhon and D. J. Dewit, editors, *14th Int. Conf. Very Large Data Bases*. Los Angeles, USA, August, 1988.
  - [21] 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.
  - [22] 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.
  - [23] C. Wittemann and H. Kunst. Intelligent assistance in flexible decisions. In *Proc. Int. Conf. Intelligent and Cooperative Information System*, pages 377–381, 1993.
  - [24] M.F. Wolf. Successful integration of databases, knowledge-based systems, and human judgement. In *Proc. Int. Conf. Intelligent and Cooperative Information System*, pages 154–162, 1993.