Conceptual Learning in Database Design

YANNIS E. IOANNIDIS, TOMAS SAULYS, and ANDREW J. WHITSITT
University of Wisconsin

This paper examines the idea of incorporating machine learning algorithms into a database system for monitoring its stream of incoming queries and generating hierarchies with the most important concepts expressed in those queries. The goal is for these hierarchies to provide valuable input to the database administrator for dynamically modifying the physical and external schemas of a database for improved system performance and user productivity. The criteria for choosing the appropriate learning algorithms are analyzed, and based on them, two such algorithms, UNIMEM and COBWEB, are selected as the most suitable ones for the task. Standard UNIMEM and COBWEB implementations have been modified to support queries as input. Based on the results of experiments with these modified implementations, the whole approach appears to be quite promising, especially if the concept hierarchy from which the learning algorithms start their processing is initialized with some of the most obvious concepts captured in the database.

Categories and Subject Descriptors: H.2.1 [**Database Management**]: Logical Design—*data models; schema and subschema*; H.2.2 [**Database Management**]: Physical Design—*access methods*; H.2.7 [**Database Management**]: Database Administration; I.2.6 [**Artificial Intelligence**]: Learning—*concept learning*

General Terms: Algorithms, Design, Management, Performance

Additional Key Words and Phrases: Adaptive database systems, learning from examples, COBWEB,/UNIMEM

1. INTRODUCTION

In the past few years, many efforts have been made to bridge the gap between the fields of artificial intelligence and database management systems. The goal of several of these efforts is to transfer technology from one field to another for improved functionality and/or performance. The work described in this paper belongs in this category as well, and addresses the issue of using machine learning algorithms for deriving concepts from database queries.

1.1 Relational Database Systems Overview

Before elaborating on the specifics of our work, we first set the stage by giving a brief overview of the relevant aspects of relational database systems.

Authors' addresses: Y. E. Ioannidis and A. J. Whitsitt, Computer Sciences Department, University of Wisconsin, Madison, WI 53706, email: yannis@cs.wisc.edu and whitsitt@cs.wisc.edu; T. Saulys, Raytheon Missile Systems Laboratories, Tewksbury, MA 01876.

Every relational database consists of a set of *relations*, which can be simply thought of as tables with many *tuples* (rows) and *attributes* (named columns). As a canonical example, consider the database (named COMPANY) of an enterprise where information is kept about employees and the departments in which they work. Such a database may consist of the following relations:

EMP (*eno, ename, age, salary, edno*)
DEPT (*dno, dname, floor, mgrno*).

EMP and DEPT are two relations, with five and four attributes respectively, whose intended meaning is rather straightforward from their names. In this database, the EMP relation will contain one tuple for each employee and the DEPT relation will contain one tuple for each department in the enterprise. Queries on this database retrieve data that satisfies a set of conditions including *selections* and *joins*. Selections involve one relation and are of the form (*attribute op value*), where $op \in \{=, <>, \leq, \geq\}$, e.g., $salary > 30K$. Joins combine two relations and are of the form (*attribute1 op attribute2*), where *op* is as in selections, e.g., $edno = dno$. As an example, in the standard database query language SQL [1], the query

select *ename*
from EMP, DEPT
where *edno* = *dno* **and** *floor* = 2 **and** *salary* > 30K

requests the names of employees who work on the second floor and make more than 30K. Note that the qualification of the query (the **where** clause) specifies that the requested data must satisfy two selections and one join.

Each relational database system supports multiple types of disk-based data structures, or *indices*, that are usually based on hashing or B^+-trees. Each relation can be supported by at most one clustered (primary) and an arbitrary number of nonclustered (secondary) indices, which are built based on the values of the relation tuples for some attribute(s). Such a wide variety of indices gives the system the ability to accelerate the processing of an arbitrary number of queries.

One of the most important characteristics of relational systems is their layered architecture. Each layer is associated with a different abstraction of the data in any given database. These abstractions are called *schemas* and from bottom-up they are the following (see Figure 1):

—*Physical schema*: It specifies the primary and secondary indices of the relations. Such details are very important for the internal processing of queries by the system, but are unnecessary to the user, which is why they are hidden by the layers above.

—*Logical schema*: This is also simply referred to as *schema* and specifies the relations in the entire database as they are known to the system and stored internally.

—*External schema*: There are usually several such schemas associated with a database, potentially one for each different user. Each external schema specifies the relations that capture the part of the database that is relevant to the users of the schema in a way that is most convenient to them.

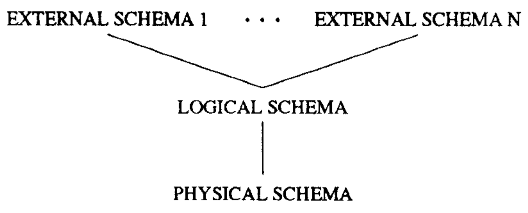


Fig. 1. Layers of schemas in relational databases.

Relations that are part of an external schema but not of the logical schema are called *views*. Users interact with the system at this layer, and they can be ignorant of the specifics of the lower layers. That is, they can use views in their queries ignoring the fact that these are not explicitly stored in the database but are defined in terms of other relations.

The task of specifying the logical and external schemas of the database is called *logical database design* and is performed by the *Database Administrator (DBA)*. For the purposes of this paper, we assume that the logical schema of a database is given and that there is a single external schema, so that logical database design consists solely of the specification of one set of views. In that sense, during logical database design, the primary goal is user productivity. In particular, based on the expected types and frequencies of user queries, the appropriate views are defined so that the queries can be as succinct as possible. For example, if many queries involve the join of EMP and DEPT in the COMPANY database based on the condition $edno = dno$, it may be worthwhile to define the result of the join as a view, so that users do not have to specify it in their queries all the time.

Similarly, the task of specifying the physical schema of the database is called *physical database design*, also performed by the DBA. In this case, the primary goal is efficiency in query processing. Again, based on the expected types and frequencies of user queries, the appropriate types of indices on the appropriate attributes of each relation are built, so that query processing can be as efficient as possible. In this case, the specific profiles of updates to the database is also very important, since indices are modified during updates, and having a large number of them degrades performance. Without loss of generality, for the purposes of this paper, we ignore updates altogether. Continuing the above example, if many queries involve the same join of EMP and DEPT, it may be worthwhile to build an index on the join attribute of one of the relations or even sort the relations on the join attributes, so that queries can be executed without scanning the whole relation multiple times for every query.

1.2 Using Machine Learning for Relation Database Design

The external and physical schemas of a database when the latter is first created may not always be optimal for the set of queries being asked by the users at any given time. Ideally, the system should have the ability to dynamically create, modify, and destroy indices and views. (Actually, deleting views would be rare to avoid invalidation of existing applications.) That is, if

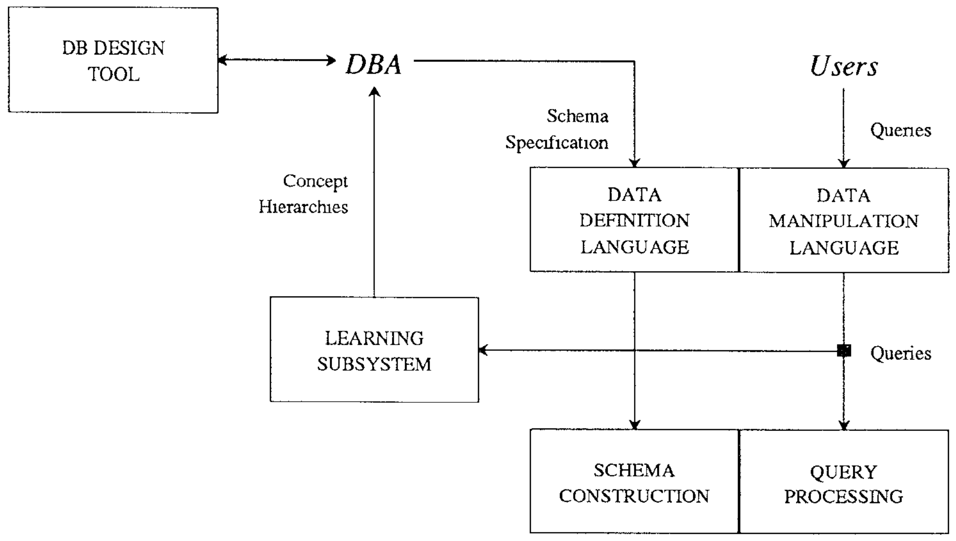


Fig. 2. Database system architecture incorporating machine learning

the system can determine that the queries currently being asked by the database users could be expressed more succinctly by a different set of views or processed more efficiently by a different set of indices, it should be able to make the appropriate suggestions for changes in the database schemas to the DBA. In effect, the database system should be able to fine-tune the user productivity and its own performance as required by its workload.

In this paper, we argue that a viable approach for attaining the above goal is through the introduction of machine learning techniques. The incorporation of machine learning into the database would generally enhance its knowledge on the way in which its contents are accessed. The system would monitor the incoming queries and try to extract the commonalities that they exhibit. This is basically the same as trying to identify concepts given some observations of events or objects that are *examples* (or *instances*) of the concepts to be learned in traditional machine learning. The concepts produced would be given to the DBA who would proceed in appropriate modifications of the database schema, possibly with the help of a database design tool (see Figure 2). Although we believe that technology is not mature enough yet, the concepts learned by the machine learning algorithm could be automatically passed on to a design tool, which would make the database reorganization. In this paper, we concentrate on the specifics of the machine learning algorithms that can be used for database design. Except for a very brief discussion, we do not address the issue of how exactly the DBA will use the output of these algorithms for database design, since for the most part, standard approaches exist that can be used for the task.

There are some differences between physical and logical database design on which we briefly want to elaborate, with respect to how incoming queries must be manipulated. In particular, for physical database design, the specific constants that appear in a query are not important. For example, the query

select *ename*
from EMP
where *salary* > 30K

provides exactly the same information for index building as any other query that has 30K replaced by some other constant. On the contrary, for logical database design, the query constants are very important, because differences in them reflect differences in the views that should potentially be defined. Hence, each query must be manipulated by the learning subsystem in two forms: (a) in its full form as submitted by the user, for logical design, and (b) with each constant replaced by some generic constant, for physical design. Each incoming query must be transformed into its generic form by some of the higher levels of a database system and then have both of its forms passed on to the learning subsystem. The primary functions of the latter do not depend on the specific query form being manipulated. Hence, for this paper, we ignore the above issue and treat queries that are input to the learning algorithm similarly, independent of whether they have real or generic constants.

1.3 Related Work

To the best of our knowledge, there has been very little work on using machine learning algorithms in database systems. Specifically, we are aware of two such efforts, both of which in fact have dealt with database design. The work of Borgida and Williamson proposes the use of machine learning to adequately represent exceptions in databases that are based on semantic data models [3]. The system learns from the encountered exceptional data objects, i.e., those that do not conform to the logical schema of a database, and suggests modifications of the schema that will accommodate them appropriately. The work of Li and McLeod uses machine learning techniques to address the problem of object flavor evolution in object-oriented database systems [12]. Similarly to the previous work, when objects are encountered that cannot be captured by the logical schema of a database, learning is employed to aid in the process of appropriately modifying the schema. The primary difference of both of these efforts from the one described in this paper is that their emphasis is on monitoring data objects so that the logical schema of database can be modified, whereas the emphasis of our work is on monitoring user queries so that the external and physical schemas of a database can be modified.

1.4 Paper Organization

This paper is organized as follows. Section 2 describes the criteria that must be satisfied by a learning algorithm for it to be applicable for database design. It also includes brief descriptions of the UNIMEM and COBWEB algorithms, which we chose based on those criteria as the most appropriate for the task. Section 3 contains the details of how standard implementations of these algorithms were modified to accommodate the special needs of the particular application studied. Section 4 presents the results of several experiments with the two algorithms, focusing on issues that are important to database design. Section 5 compares UNIMEM and COBWEB and attempts to identify the one that is more useful. Finally, Section 6 summarizes the presented work and proposes some directions for future work.

2. MACHINE LEARNING ALGORITHMS

For the rest of the paper, we use the terms *example* and *instance* indistinguishably to denote the basic unit of input to the learning algorithm.

2.1 Criteria for Choosing Machine Learning Algorithms for Database Design

Several criteria that limit the kinds of machine learning algorithms that are applicable for database design can be derived from the nature of the operation of database systems. The first such criterion has been mentioned already. The system must learn the necessary concepts through observation, by examining examples of those concepts. Each meaningful query entered into the system is the example of some concept that the system might find useful to identify. For instance, the aforementioned join of EMP and DEPT based on the condition *edno* = *dno* identifies the concept of *Employment*. To learn this concept, the system must monitor the stream of incoming queries and group together all those that involve this particular join.

The second criterion builds on the first and is that the learning algorithm must be capable of receiving only positive examples as input, since that is all that database queries can represent. This is crucial because many of the machine learning algorithms that learn from examples do so by making use of both positive and negative examples of a concept. Examples of such algorithms include Quinlan's ID3 [16] and Mitchell's VERSION-SPACE [13]. The importance of negative examples is that they provide bounds on how general the concept description can be made and yet still have it cover only proper examples of the concept. With only positive examples, the learning task is more difficult.

The third criterion is that the algorithm must be able to work incrementally and learn concepts as examples arrive. Many potentially useful machine learning algorithms, e.g., Stepp and Michalski's CLUSTER/RD algorithm [17], are batch-oriented, in that they expect to have all of the examples present before they begin executing. One could use such a learning algorithm and run it only periodically, at moments when the system is not so busy, each time using the example queries collected since its previous execution. However, if such times of light load are infrequent, the concepts that the system learns may no longer be so important if the focus of users' queries is gradually drifting from one set of concepts to another, a phenomenon known as *concept drift*. Given that the algorithm runs incrementally, it is important that it be computationally efficient or the system performance will suffer.

The fourth criterion is that the learning algorithm must be able to form its own classifications. The reason is that an example cannot be directly associated with the concept to which it applies, since the former does not carry any information to that effect. This task is called *conceptual clustering* (or *learning without a teacher*). Conceptual clustering involves observing the instances to be classified and recognizing similarities and differences among them, which can then be used to cluster the objects into classes or concepts. Fisher [4] points out that the activity of performing conceptual clustering actually involves two tasks: the *clustering task* and the *characterization task*. The clustering task corresponds to determining useful subsets of a set of objects, i.e., the examples of a given concept. The characterization task involves determining useful concepts for each object class, which is simply learning a concept from examples.

The fifth criterion is that the learning algorithm must be capable of learning multiple concepts simultaneously, since the queries entering the system represent a wide variety of concepts. In addition, the algorithm must perform its task with the examples arriving in some unpredictable, arbitrary order, and not clustered in any particular way. Finally, the concepts it has to learn are likely to be overlapping rather than mutually disjoint. This imposes the constraint that the learning algorithm must be able to classify a given example as a member of more than one concept. For example, the concept of *well-paid employees* implicitly represents the concept of *employment* as well.

Several machine learning algorithms meet enough of the desired criteria to be worthy of consideration for providing the desired functionality to the database system. Two such algorithms, which are good representatives of the class of appropriate algorithms, are discussed in detail in this paper. These are Lebowitz' UNIMEM algorithm [10, 11] and Fisher's COBWEB algorithm [4]. Several other such algorithms are not discussed due to lack of space, e.g., CYRUS [8]. UNIMEM satisfies all five of the above criteria, while COBWEB only fails to satisfy the need for creating overlapping concepts, which is part of the fifth criterion above. Both algorithms form a hierarchy of concepts, with concepts at higher levels of the hierarchy being generalized versions of the concepts at the lower levels of the hierarchy. The main difference between them is in the method by which the concept hierarchy is created and maintained. Regarding their input, both algorithms require that the examples be represented as feature vectors, where a feature is a property whose value has some role in classifying each example in the appropriate concept. Each element of the feature vector is a pair consisting of a feature name and a value for that feature. The two algorithms are briefly described in the next subsections. Detailed descriptions can be found in the original references [4, 10, 11].

2.2 UNIMEM

UNIMEM (UNIversal MEMory) is based on the notion of *Generalization-Based Memory* (*GBM*), a hierarchical arrangement of concepts for describing classes of objects. A GBM hierarchy is built up by generalizing from sets of examples. For each incoming example, memory (the concept hierarchy) is searched for examples with similar features with the goal of potentially forming new concepts. As a result, this hierarchy changes dynamically as new examples are presented to the system.

As discussed in the previous section, the examples with which UNIMEM works are described as feature vectors. A concept is represented by storing *some* of the feature values from its instances. These stored feature values are the ones used to create the concept. Thus, it is expected that these features will have similar values for all of the instances stored at this concept.

To add a new example to GBM, the algorithm performs a controlled depth-first search on the hierarchy, compares the example to those stored at existing concepts, and eventually stores the example either in the (potentially multiple) most specific concepts that it matches or in a newly-created concept. The comparison between two examples consists of matching on their feature values. The more features on which they have the same values, the more similar the examples are considered. When comparing two different feature values, UNIMEM issues a score of 1 if the feature values are identical or a score of 0 if the feature values do not match at all. For *linear* features, on which a total order is defined, a partial match score between 0 and 1 can also be issued, depending on how close in the total order the values of the features are. That score is inversely proportional to the *distance* between the two feature values in the total order. Partial matching is very powerful because it allows concepts to be formed that do not perfectly describe all the instances that they cover. The final match score between two examples is the sum of the scores of the individual pairs of features. The matching routines can also take into account cases where feature values are missing (unknown or inapplicable) by penalizing the overall matching score. This is important in a database application, since most attributes known to the database are not specified in any one query.

Feature *confidence values* are stored with every feature in a concept. These confidence values are modified with every new example entering the system depending on how well its values for those features match against those of the concept. Features can be deleted from a concept (if their confidence values fall below some threshold) or be kept permanently in a concept (if their confidence values rise above some threshold). A concept is deleted if too many of its original features are deleted. The main advantage of keeping feature confidence values is that they can allow overly-specific concepts to be removed from the concept hierarchy.

2.3 COBWEB

COBWEB was developed by Fisher [4] and is another algorithm that performs conceptual clustering in a way that is applicable to database design. Similarly to UNIMEM, COBWEB monitors a stream of incoming examples and incrementally adds them to a dynamic concept hierarchy. The primary differences between the two algorithms are that COBWEB uses conditional probabilities when deciding where in the hierarchy to create new concepts (clustering) and probabilistic pattern matching when matching a new example to existing concepts in the hierarchy (characterization) [4].

A concept is represented by a feature vector that contains *all* known features. Each feature is associated with a list of *all* values for that feature in the examples stored in the concept, together with the number of occurrences of each value among all these examples. Unlike UNIMEM, each example in COBWEB is stored in only one location in the concept hierarchy, which in fact must be a leaf. Also, unlike UNIMEM, there are no feature confidence values, so concepts are never deleted and the hierarchy can become quite large if many queries are processed. Thus, COBWEB is less efficient in processing examples than UNIMEM.

The metric used for identifying the usefulness of a concept is known as *category utility*. Given a new example, COBWEB examines various possibilities of changing the concept hierarchy to accommodate the example, e.g., splitting nodes, merging nodes, or classifying the example in an existing node, and chooses the one with the highest category utility. Based on a way of predicting basic-level categories in human classification, category utility is essentially a function that rewards similarity between objects within the same class and dissimilarity between objects in different classes. Category utility is defined as the *increase* in the expected number of attribute values that can be correctly guessed given a partition of concepts, over the expected number of correct guesses with no such knowledge. It is calculated using conditional probabilities of the form $P(A_i = V_{ij}|C_k)$, i.e., probabilities that the ith attribute A_i of an example takes on the jth value V_{ij} among its potential values, given that the example is a member of the kth concept C_k. These probabilities are derived from the feature value counts that are stored with the concept nodes and are also used to deal with missing feature values. Fisher [4] points out that using these probabilities for conceptual clustering is useful in a predictive sense as well as in a classification sense. If the concepts derived by the algorithm are truly meaningful, then they should reflect the types of instances that will be entering the system in the future.

2.4 Example

In this section, we use the COMPANY database as an example to illustrate the actions of the two algorithms, UNIMEM and COBWEB, as they receive a sample stream of queries as input. Before we proceed with the example, we should mention one major modification that must be done to both algorithms so that the class of queries that is acceptable to them is not overly restricted. The algorithms as described above cannot handle joins, because unlike equality selections, joins are not directly mappable to a feature with a value specified for it. Hence, the matching that the algorithm performs must be extended to allow an attribute to take on the name of another attribute as a value, independent of the type of values that the feature normally has. For example, if *eno* normally takes on integer values, it must be able to also take on values such as *mgrno* to express the join *eno* = *mgrno*. In our work, to solve this problem, the legal values for each attribute have been enhanced to include the names of all other attributes that could be joined with it. Then, matching for joins can be treated similarly to constant selections. For example, the following SQL query

```
select ename
from EMP, DEPT
where edno = dno
```

can be represented by the following feature vector:

$$((\textit{ename}\,?)(\textit{edno} = \textit{dno})(\textit{dno} = \textit{edno})).$$

This notation, which explicitly represents the joins from the perspective of both join attributes, ensures that joins will always be treated consistently. For simplicity, we assume that the attributes being printed in the query results (those in the **select** clause) do not play a major role in the formation of concepts, so they are omitted in the following examples, unless they are also involved in a join or a selection.

Consider the following stream of queries on the COMPANY database (in feature vector notation):

1. (*name employment*)(*edno* = *dno*)(*dno* = *edno*)
2. (*name well-paid-emp*1)(*edno* = *dno*)(*dno* = *edno*)(*age* = 20)(*salary* = 200K)
3. (*name senseless*)(*edno* = *floor*)(*floor* = *edno*)(*age* = 20)
4. (*name young-emp*)(*edno* = *dno*)(*dno* = *edno*)(*age* = 20)
5. (*name well-paid-emp*2)(*edno* = *dno*)(*dno* = *edno*)(*age* = 20)(*salary* = 200K)

The *name* feature has been added in this example to more easily identify what concept is represented by each query. In a system where the user is meant to be unaware that learning takes place, this would have to be generated and assigned by the system. For the purposes of this example, we assume that one similar feature value between examples is sufficient for creating a new concept.

We first examine the actions of UNIMEM as queries enter the database system one at a time and are passed on to it. When the first query is processed, the current concept hierarchy is empty (having only the root node). Therefore, instance (1) is stored at the root node where it can never conflict with the features of any other example (no features are necessary to be a member of the root concept). The concept hierarchy that is generated after the process is completed is shown in Figure 3.

When the second query enters the system, the root is the only node in the concept hierarchy. Instance (2) is compared with the instance stored at this node, namely instance (1), to determine whether or not they have enough feature values in common to form a new concept. Since they match exactly on values of the features *edno* and *dno*, a new concept node is formed whose stored features are *edno* and *dno*, and both instance (1) and instance (2) are stored in the new node. Instance (1) is removed from its place at the root node, and finally the new node is added to the hierarchy as a child of the original node. At this point, the concept hierarchy looks as in Figure 4. The numbers in brackets represent the algorithm's confidence values for those feature-value combinations.

The final state of the concept hierarchy, after the remaining three queries enter the system and are all processed, is shown in Figure 5. Note that this particular set of queries formed a nice *linear* hierarchy in which the concepts

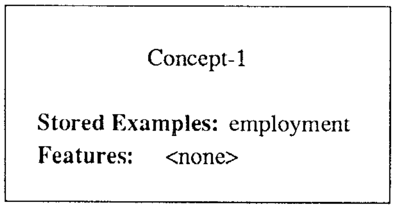


Fig. 3. Concept hierarchy generated by UNIMEM after query (1).

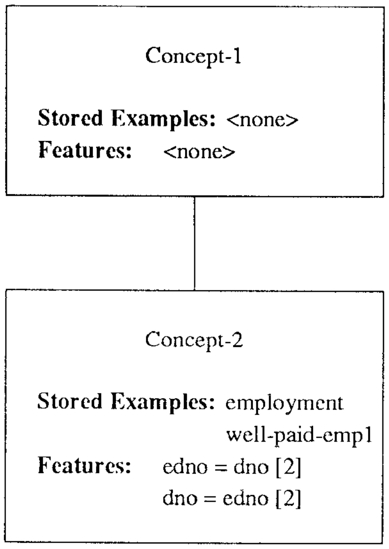


Fig. 4. Concept hierarchy generated by UNIMEM after query (2).

become more specialized deeper in the hierarchy. Concept-2 represents the notion of employment as evidenced by the *edno* = *dno* join. Concept-3 is slightly more specialized and represents the concept of young employees by including the *age* = *20* feature as well as the *edno* = *dno* join, which it inherits from Concept-2, its parent in the concept hierarchy. Finally, Concept-4 adds another new feature to the above, namely *salary* = *200K*, to represent the notion of well-paid employees. Not all streams of instances will result in such a linear pattern. If more queries that involved an entirely different set of relations were introduced, e.g., authors, books, and book subjects, those concepts would form an entirely separate subtree.

An important point is that queries like (3) could be removed from the root of the concept hierarchy if they do not match with any other queries after some specified significant length of time. For a query to not be incorporated into any concepts is an indication that it is insignificant with respect to the concepts that it represents and can be removed from the concept hierarchy without any loss of important information.

We next demonstrate how COBWEB performs on the exact same incoming stream of queries used above. For the sake of brevity, only those attribute-

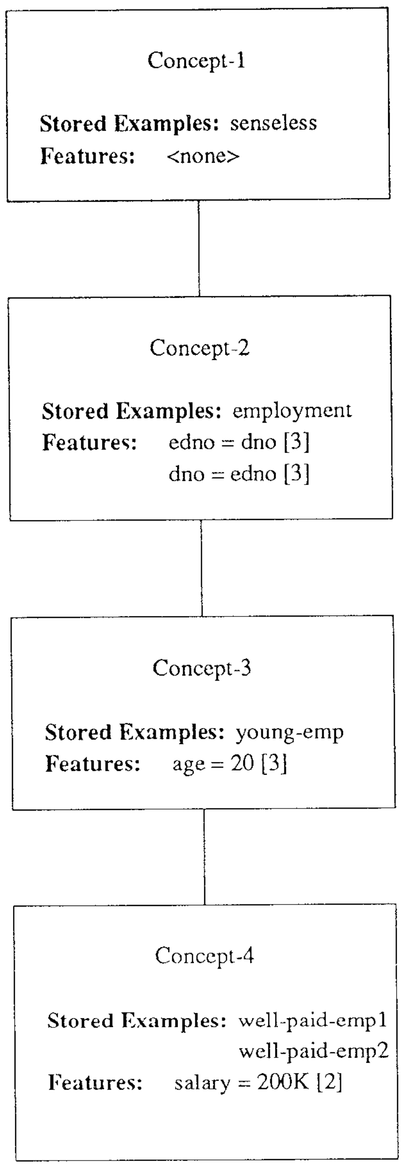

Concept-1
Stored Examples: senseless
Features: <none>
Concept-2
Stored Examples: employment
Features: edno = dno [3]
dno = edno [3]
Concept-3
Stored Examples: young-emp
Features: age = 20 [3]
Concept-4
Stored Examples: well-paid-emp1
well-paid-emp2
Features: salary = 200K [2]

Fig. 5. Concept hierarchy generated by UNIMEM after queries (3)-(5).

value pairs with a nonzero probability are shown in the concepts. In addition, the probabilities are shown directly rather than the feature value counts in terms of which probabilities are calculated.

As the first query enters the system, COBWEB is called to put it into the hierarchy. Since the hierarchy is empty, there is no choice but to create a new concept at the root and place this instance into that concept. The resulting concept node is shown in Figure 6. When the second query enters the system, the unique node in the hierarchy is split into two nodes, one for each

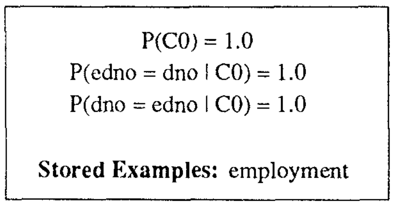


Fig. 6. Concept hierarchy generated by COBWEB after query (1).

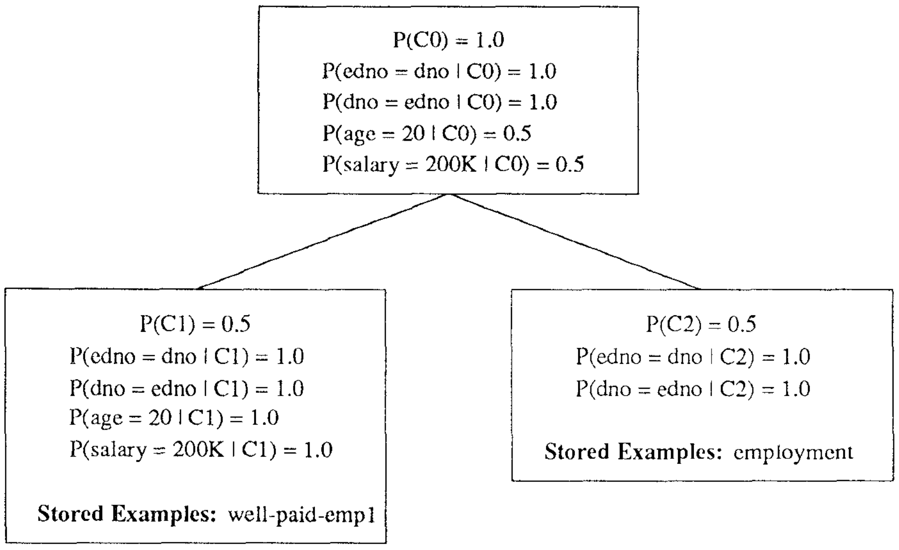


Fig. 7. Concept hierarchy generated by COBWEB after query (2).

instance, since that gives the best category utility. The resulting hierarchy is shown in Figure 7. Finally, the hierarchy that results after the remaining three instances in the query stream are processed is shown in Figure 8. Note that the concept hierarchy produced by COBWEB is similar to, but not exactly the same as, that produced by UNIMEM. COBWEB forms a shallower, bushier tree, since some of the nodes have a greater branching factor.

2.5 Use of Concept Hierarchies

In this subsection, we want to elaborate on how the specific algorithms, UNIMEM and COBWEB, and the concept hierarchies that they produce can be used within database systems for database design. In particular, we want to provide some guidelines that can be used by the DBA in interpreting the concept hierarchies produced and making beneficial changes to the external and physical schemas of a database. We primarily focus on UNIMEM, because it is the easiest one to explain, and only briefly discuss COBWEB at the end.

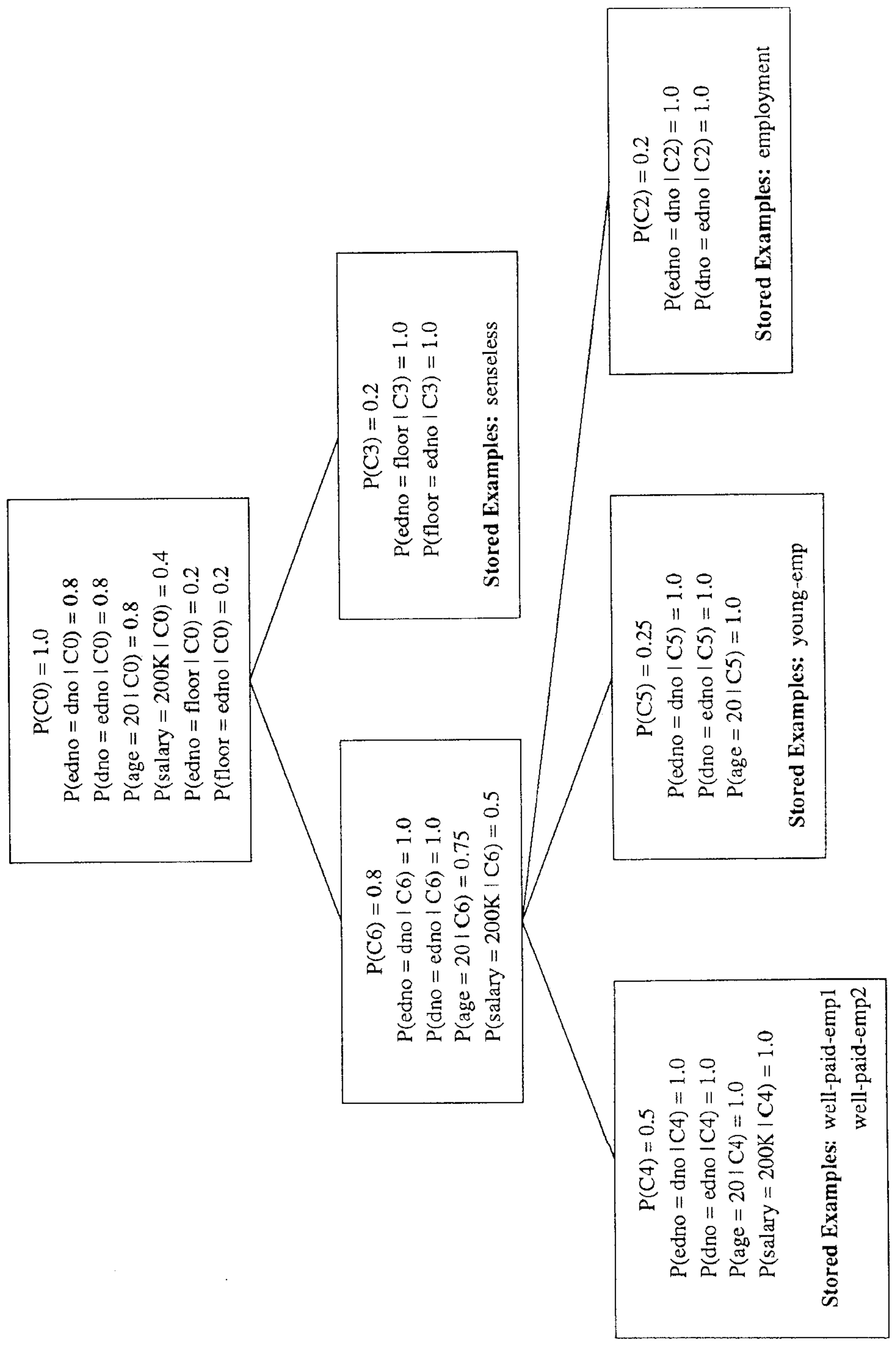


Fig. 8. Concept hierarchy generated by COBWEB after queries (3)-(5).

We start with physical database design. First, any attributes that figure prominently in the higher levels of the UNIMEM hierarchy (called *high attributes*) are good candidates for indices if they do not already have them, since they are involved in the most interesting concepts and are the most likely to appear in queries. Among those, attributes with high confidence values are the best choices for building indices. For each relation, the highest such attribute should be chosen for a clustered index, and some of the remaining ones for nonclustered indices. Second, the specific form in which an attribute appears in a concept indicates the appropriate type of index: if the candidate attribute appears in most concepts in equality selections and joins, hashing is to be preferred; if it appears in many concepts in nonequality selections, a B^+-tree is to be preferred. Third, if more advanced features are supported by the database system, the concept hierarchy can provide guidelines for how they can be used as well. If join indices are supported [18], then they should be built on high attributes that appear in joins. If multidimensional indices are supported [7], then they should be built on sets of high attributes that appear in many concepts together.

As an example of the above, let UNIMEM be the learning algorithm used by the system and consider queries (1)–(5) on the COMPANY database. Assuming that they are fairly typical of the kinds of queries that the system receives, the DBA can use the generated concept hierarchy and identify that clustered hashed indices would be clearly useful on the join attributes *edno* and *dno*. In addition, the DBA can also determine that it would be better to put a nonclustered index on the *age* attribute than on the *salary* attribute, since the former is higher up in the hierarchy than the latter, i.e., it participates in more queries (concepts).

We continue with logical database design. For this, the approach is similar to physical database design, but the options are much fewer. Concepts that appear in the higher levels of the hierarchy are good candidates for being captured by views. The qualification of the view definition will include all features in the concept that have high confidence values. The attributes of the view will include all features that (a) are prominently identified by the concept as part of the output of queries, and (b) are part of close descendents of the concept (one or two levels lower) in the hierarchy. This will increase the usability of the view. Note that for logical database design, the algorithm must keep track of the printed attributes of queries, unlike our practice in the example of Section 2.4.

Continuing on with the above example, consider the concept hierarchy generated by UNIMEM (Figure 5). Clearly, a view that joins EMP and DEPT based on the condition *edno* = *dno* and contains *eno*, *age*, and possibly *salary*, is very useful, since it can be used by almost all queries that have produced the concept hierarchy.

We conclude with a use of concept hierarchies that is unrelated to database design, i.e., identification of ill-formed queries. For example, consider the "senseless" query (3) and again let UNIMEM be the learning algorithm used by the system. If this query were submitted after a rich concept hierarchy had been created by UNIMEM, it would probably still end up being stored at the root node with all the other unusual queries. UNIMEM could notify the user that this query does not match any of the concepts that it has identified, and ask for confirmation that the submitted query is indeed the one intended by the user. If the query is correct, the system can continue processing it and produce an answer. If the query is incorrect, however, the user can fix the mistakes and have the system evaluate the corrected query, instead of wasting valuable resources on the original one. In fact, this can be done with the user examining the parts of the concept hierarchy that are closest to the submitted query, as an aid to identify possible misconceptions on the user's part. (This is reminiscent of Motro's work on dealing with queries having null answers [14].) Such behavior on the part of the database system may also increase the users' confidence in the system, since the latter would appear to have knowledge about the semantics of the information that it contains.

The difficulty in explaining how the concept hierarchy generated by COBWEB can be used for database design stems from the fact that useful information is scattered throughout the hierarchy. The root is important, because the features that have high conditional probabilities there are the ones that should be chosen for indexing. The same features are also the ones that should be used to define views, but the root does not contain any information about how these should be combined. The DBA must traverse the subtrees of the hierarchy rooted at nodes with high probabilities to identify which of these features are often used together in queries. Finding such features together in internal nodes where the conditional probabilities of the features are *very high* (close to 1.0) is a strong indication that they should be combined in views. In the above example, the join of EMP and DEPT that is based on the condition $edno = dno$ is found in the internal node C1 and both features that correspond to it have conditional probability 1.0. Hence, defining a view that captures it would be very useful. We should mention that the leaves of the COBWEB hierarchy have almost no importance, since their union simply corresponds to a list of all queries that have ever been submitted to the system. As we move, however, from the leaves towards the root in a bottom-up fashion, the significance of finding multiple features with very high conditional probabilities increases.

3. IMPLEMENTATION OF LEARNING ALGORITHMS

For the experiments described in the next section, we have used LISP implementations of both UNIMEM and COBWEB that have been developed in the labs of Jude Shavlik of the University of Wisconsin and Alberto Segre of Cornell University, respectively. Both were originally written to accept standard feature vectors as input and needed extensive modifications to accept database queries as input, and to properly match these queries with the concept hierarchy.

For this work, a database query is defined to be a conjunction of one or more *atoms* of the following form: $(attr_i \; op_i \; value_i)$, where $attr_i$ is the name of an attribute of a relation, op_i is a comparison operator in $\{=, <>, \leq, \text{or} \geq\}$, and $value_i$ is either a constant that is type-compatible with $attr_i$ or the name of an attribute with which $attr_i$ can be joined.

The current versions of the algorithms impose the following restrictions on the form of an input query:

—Only conjunctive queries are allowed.

—Only equality joins are allowed.

—At most one copy of each relation is allowed.

—Each attribute is allowed to appear in at most one atom, unless the user can provide a consistent renaming of the attribute.

The original learning algorithms were modified as follows.

(A) A scheme was developed for efficient feature value matching so that the database performance does not degrade. This relates to the comments made earlier regarding the representation of join attributes. As demonstrated in the examples, the scheme consists of making all attributes that may possibly be joined with a particular attribute potential values for the latter. Thus, with the appropriate internal renaming of attributes, the learning algorithms can treat join-term values like any other constants and apply a simple pattern matching algorithm on feature values instead of unification of variables. This solution works well for the subset of relational algebra queries to which we have limited ourselves, i.e., queries in which at most one copy of each relation is used. More sophisticated techniques will be necessary to remove this limitation.

(B) The problem of features whose values are unknown or inapplicable was solved. For UNIMEM, the solution was provided by the algorithm itself: a low value (0.10) was used as the partial match score for a missing feature. In essence, this captured the fact that missing feature values should be given little weight in a match score, but more weight than two feature values that are in complete conflict, which earns a score of 0. The solution for COBWEB is more difficult. The feature vector representation of COBWEB requires that each feature have a value for every example. The solution was to use the value UNKNOWN for each feature that has no value. This introduces a bias into the algorithm, since the category utility score increases if one feature in a concept has a large number of UNKNOWN values. This bias, however, did not seem significant on the queries that were tested with the algorithm.

(C) An efficient way to store multiple copies of the same example into the concept hierarchy was devised. In COBWEB, if an example is identical to another example in some leaf concept node, only the feature value counts are updated; the new example is not stored in a separate leaf. In UNIMEM, a similar approach was adopted. If a concept contains an example that is identical to the current example, the feature confidences at the concept node are automatically updated with no further action. In both cases, this complicates the comparisons required when performing the matching to create a concept or adjust feature confidences. Our experiments have shown, however, that such complications are easy to overcome and the performance trade-offs are in favor of doing so. This approach is also taken in the CLASSIT algorithm [6, 9], which is an extension of COBWEB.

(D) The flexibility of the algorithms was expanded when matching linear features, on which a total order is defined. For example, when comparing the ages of two people, it can generally be assumed that a difference of one year is insignificant, i.e, those people can be considered to have the same age when considering feature value matching. In our implementation, such "approximate" matching was only supported for the case of linear features that take numerical values but not for *nominal features*, which take symbolic values.

Currently, the user can specify one of three methods by which the algorithm can take advantage of the total order of linear features in the matching process. The first method is the specification of a *radius*, a notion that was introduced by Motro in a system that could deal with vague queries [15]. A radius indicates the minimum acceptable distance (difference) between two numerical values for them to be considered equal. For example, for the *age* attribute, the chosen radius could be equal to 5, which indicates that two people whose ages are within five years are to be considered as having the same age by the learning algorithm. In COBWEB, there may be several attribute values in a concept within the same radius as the new instance attribute value. Thus, it is necessary to find the closest match between the new feature value and each of the existing feature values in the concept when deciding how to update the feature value counts in the concept.

The second method can be used only for UNIMEM, which as we mentioned in Section 2.2, can return a value between 0 and 1 to indicate a partial match between two particular feature values. The algorithm requires that the minimum $V_{\min}$ and maximum $V_{\max}$ possible values for the range of feature values be specified. Then, for feature values V_1 and V_2, their match score is equal to

$$1 - \frac{|V_1 - V_2|}{V_{\max} - V_{\min}}.$$

For example, consider the *age* attribute with allowable values in the range [0, 100]. If the feature values are 20 and 25 respectively, then the partial match score is equal to 0.95, which indicates that the values are reasonably close.

The third method requires partitioning the whole range of feature values in a small number of subranges. For example, for the *floor* attribute, the partitions may specify that floors below the fifth are *low*, those between the fifth and tenth are *medium*, and those above the tenth are *high*. Then, the system will use only the three values *low*, *medium*, and *high* when making decisions about matching feature values. We refer to this type of feature as a *partitioned feature*.

(E) The algorithm was enhanced to accept queries with range selections on linear attributes. The need for these can be illustrated by the above examples where query (2) tried to model the notion of a well-paid employee by specifying that the salary was *equal* to 200K. A more realistic formulation of the concept would be as an employee whose salary is *greater than or equal* to

200K. To properly implement range selections, feature values are represented as a list of the following format: (*start-value end-value*). For equality joins and selections, *start-value* is equal to *end-value*. For selections that have the $<$, $\leq$, $>$, or $\geq$ operator with one of the endpoints of the range being unknown, the unknown value is represented by a ?. For example, the selection (age > 25) is represented by the value (25 ?). If, however, it has been specified to the system that the legal range for age is between 0 and 100, the preceding selection is represented by (25 100). In addition, if age is a partitioned attribute with partitions ((0 17 *young*) (18 65 *middle*) (65 100 *old*)), the above selection is represented by the value (*middle old*).

Matching for range selections is done by checking how close the endpoints of the feature values are. Suppose that feature value ($start_1$ end_1) is being compared to feature value ($start_2$ end_2). If $start_i$ is not equal to end_i for either feature value, then the value $|start_1 - start_2| + |end_1 - end_2|$ is computed. This value can be compared to some specific radius, or it can be used in UNIMEM to compute a partial match score. Note that, if one of the endpoint values is ?, the corresponding endpoint value in the other feature must also be the same in order for a nonzero feature value match score to be generated.

4. ANALYSIS

In this section, we present the results of experiments that we perform with the enhanced versions of UNIMEM and COBWEB described above. The schema of the BANK database that was used to study sample concept hierarchies produced by the algorithms is shown below.

ACCOUNT (*actno, bal, cust_name*)
TRANSACTION (*tno, type, bal_change, tactno, tatmno*)
CUSTOMER (*ssno, name, age, street, city, num_kids, married*)
ATM (*atmno, num_trans, location, disabled_time*)

For its linear attributes, combinations of all three methods for handling approximate matching were used. Below we show the templates for the queries used in the experiments (in English) with the actual values having been replaced by the generic x. The queries were chosen so that they provide a good mix of selections and joins.

(1) Balance of account x.
(2) Transactions of account of customer x.
(3) Address, social security number, and transactions of account of customer x.
(4) Customers with withdrawals of more than x.
(5) Customers with withdrawals of between x and y.
(6) Customers with accounts whose balance is greater than x.
(7) Age of customers with accounts whose balance is greater than x.
(8) Social security numbers of customers with accounts whose balance is greater than x.

(9) Marital status of customers with accounts whose balance is greater than x.
(10) ATMs with transactions on accounts whose balance is greater than x.
(11) Disabled times of ATMs with transactions on accounts whose balance is greater than x.
(12) Number of transactions and disabled times of ATMs with transactions on accounts whose balance is greater than x.
(13) Balance of customers with more than x kids.

For each algorithm, twelve different concept hierarchies were created. Each time a set of 1000 sample queries chosen from the templates above were provided as input, each template having an equal number of representatives in the set. These sets are denoted by $qset_i$, where $1 \le i \le 12$. In $qset_1$, the queries from each template were clustered together, i.e., they were one after the other, without any interleaving among templates. In $qset_2$, the templates were maximally interleaved, i.e., a sequence of one query per template was finished before another sequence of the same form started. In each of the remaining ten query sets, $qset_3$ to $qset_{12}$, queries were randomly ordered. Our analysis of the concept hierarchies generated from the above query sets focus on the following four aspects: (a) consistency of the generated concept hierarchies across different input orders for the queries, (b) the effect of starting with a nonempty concept hierarchy based on well-educated estimates about what the interesting concepts are, instead of starting with an empty one, (c) size of the concept hierarchies generated by the algorithms and its variations as the number of observed examples increases, and (d) efficiency of the learning algorithms. The next few paragraphs discuss the results of our experiments with respect to all the above aspects.

In general, it is expected that the order in which queries are presented is significant for both learning algorithms. This is due to the fact that the algorithms create different initial concepts each time, which then affects both the placement of the examples that follows in the hierarchy and the updates of the feature confidence values or conditional probabilities. To establish a better understanding of that effect, we compared the final concept hierarchies that were generated for the twelve query sets used in the study. The hierarchies produced were quite large and, therefore, difficult to examine. Below, we present the results on the comparison of the concept hierarchies produced by two of the randomly ordered query sets. The derived conclusions, however, are in general applicable to the whole suite of conducted experiments.

Based on the discussion in Section 2.5, the top levels of the concept hierarchy are generally the most important for making decisions about dynamic changes to the external and physical database schemas. All generated hierarchies in our experiments were of depth 3 to 5, so we only compared the two levels in each hierarchy that were immediately below the root. The root was excluded because, in UNIMEM, it captures no meaningful concepts, whereas in COBWEB, it is independent of the order of arrival of queries, so it is the same for all input query sets that contain the same queries.

Table I. Comparison of the Top Levels of Two Concept Hierarchies

Algorithm	Level 1		Level 2	
	same	different	same	different
UNIMEM	3	2	5	11
COBWEB	4	0	8	2

The results are summarized in Table I, where for each algorithm, we present the number of nodes that were the same in both hierarchies and also the number of those that were different, separately for each hierarchy level of interest.

The difference between the two hierarchies in each case is clearly evident by the entries in the "different" columns. However, there are significant similarities as well. In particular, both hierarchies had the most common joins captured by concepts in their top levels, which is very desirable due to the important role of joins in relational databases. These would be the join between CUSTOMER and ACCOUNT based on *name = cust_name* and the join between ACCOUNT and TRANSACTION based on *actno = tactno*. This was to be expected, since the query representation that we chose introduces a bias towards joins by requiring that each join is represented twice, once for each join attribute. Most of the other common concepts in the top levels captured groups of queries with the same template.

A comparison of the two learning algorithms, clearly, shows that the concept hierarchies produced by COBWEB are more consistent than those produced by UNIMEM. This is seen by the larger numbers for COBWEB in the "same" columns of Table I, but primarily by the much lower numbers for COBWEB in the "different" columns. Because of its more advanced techniques, COBWEB generated almost no spurious or unrelated concepts at the top levels. This is unlike UNIMEM, which generates several more concepts than COBWEB, many of which are insignificant and may eventually be deleted.

Given the sensitivity of UNIMEM to the order of the incoming queries, we focus on that algorithm and elaborate on the issue further. An interesting question that arises is whether or not there is an optimal order in which queries should be received by the system so that only the most meaningful concepts are formed. In that respect, it is clearly optimal for all queries on the important concepts to be presented first and in clusters as in $qset_1$. This ensures that each such concept is identified and permanently established in the hierarchy before unrelated queries can cause the relevant features to be discarded. (In fact, this order has been determined to be optimal for COBWEB as well as for the ITERATE algorithm, which is an extension of COBWEB, by Biswas et al. [2]).

The above observation inspired the following approach for achieving more consistent results when the learning algorithm is given queries in different orders. In a preprocessing phase, the concept hierarchy is initialized by a carefully chosen set of queries in a specific order, so that important concepts are well established in the hierarchy. When the actual queries arrive during normal processing, the initialized concept hierarchy should play the role of a robust basis so that the subsequent building of concepts will be relatively uniform and uninfluenced by the order of incoming queries. Such a preprocessing phase is easily implementable in database applications, because most often the DBA has right from the beginning quite accurate knowledge about some of the most important concepts that are captured in a database, e.g., the natural joins between relations. The desired concept hierarchy can be generated by simply choosing an appropriate set of queries and "spoon-feeding" it to the database system.

We experimented with the above method by introducing to UNIMEM a specific set of 28 queries in a given order during a preprocessing phase. The resulting concept hierarchy had as children of the root four important joins that semantically connect the relations of the BANK database. These are the join between CUSTOMER and ACCOUNT based on $name = cust\_name$, the join between ACCOUNT and TRANSACTION based on $actno = tactno$, the join between TRANSACTION and ATM based on $tatmno = atmno$, and the join between CUSTOMER and ATM based on $city = location$. The parameters of the algorithm were chosen so that the corresponding concepts were never deleted. After the preprocessing phase, we ran the exact same experiments as before with all twelve query sets starting from the initialized concept hierarchy. We again only present the results from the comparison of the concept hierarchies produced by the two specific query sets used before, although the derived conclusions are more generally applicable. In a similar format to Table I, these results are given in Table II, where the corresponding numbers for UNIMEM without the preprocessing phase are repeated for ease of comparison.

Clearly, the preprocessing phase is very valuable to UNIMEM. The common nodes in *both* levels increased (although only in the top level were nodes fixed during initialization), while the differences decreased dramatically. Moreover, note that the total number of nodes in the top two levels decreased (as evidenced by the decreased sum of commonalities and differences), which is an indication of the decrease in the generation of spurious concepts. We conclude that when using UNIMEM for database design, initializing the concept hierarchy with a small set of carefully chosen concepts is essential for consistency in the final result. Therefore, the rest of the analysis concerns the experiments that involved this preprocessing phase.

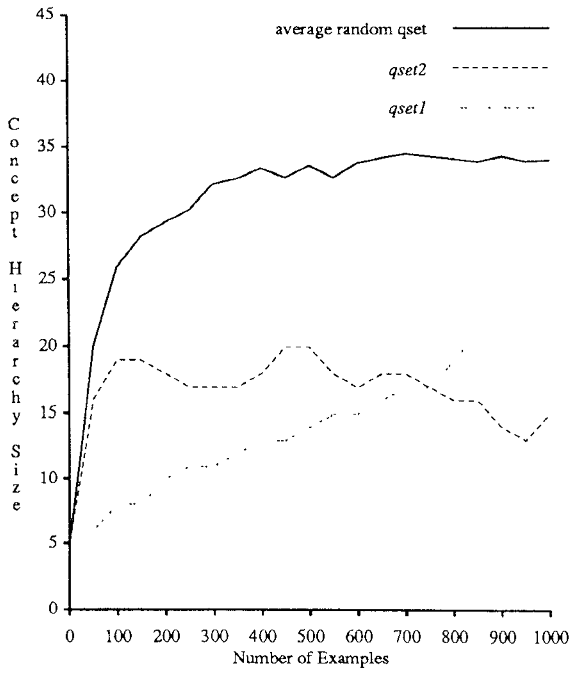
To measure the changes in the size of the generated concept hierarchies as the number of incoming examples increases, we instrumented the algorithms so that the appropriate measurements are taken every 50 examples. Figure 9 shows typical behaviors observed among the twelve different tested sets of queries. We show the behavior of both algorithms for the deterministically constructed query sets $qset_1$ and $qset_2$ as well as for the average among the randomly generated sets. In general, UNIMEM generates hierarchies that are larger than those of COBWEB. This is due to the fact that UNIMEM

Table II. Comparison of the Top Levels of Two Concept Hierarchies with Initialization

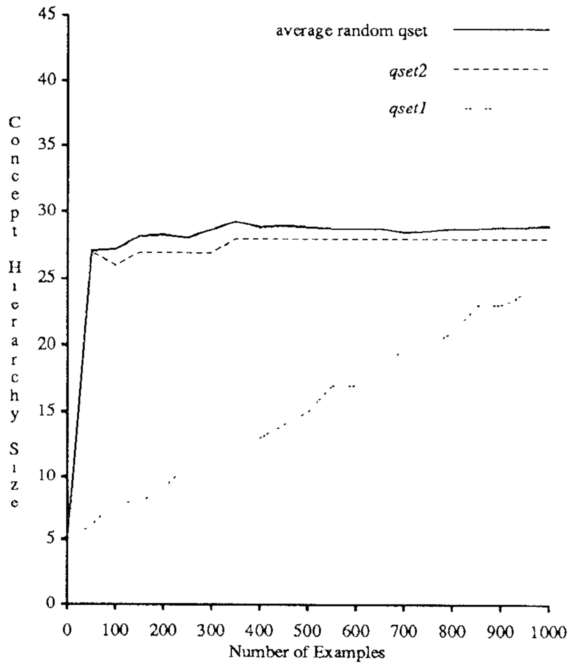
Algorithm	Level 1		Level 2	
	same	different	same	different
UNIMEM with initialization	5	0	7	4
UNIMEM without initialization	3	2	5	11

stores examples in multiple concepts and allows overlapping concepts, whereas COBWEB does neither of these. The difference in size can be quite significant, and in our experiments, it occasionally reached close to a factor of 2. For $qset_1$, both algorithms have a similar behavior: the size of the hierarchy grows monotonically for the most part with the number of incoming queries. This is to be expected, since in $qset_1$, many similar examples are given to the system consecutively, so the corresponding concepts are learned one at a time and become solidly established in the hierarchy. For the remaining query sets, the behavior is quite different. For UNIMEM, although the size of the hierarchy grows overall, there is significant variation as examples are received as input, and the size can grow or shrink at times in unpredictable ways. In contrast, COBWEB is much more stable, reaching very close to its final size after few examples and exhibiting only minor changes beyond that point. This is partly due to the fact that COBWEB does not delete any concepts from the hierarchy, but also due to the more sophisticated techniques that it employs for concept formation. Another interesting observation is that, for UNIMEM, there is significant variation on the behavior among the different query sets, whereas COBWEB again is very stable, exhibiting only minor differences among them.

Finally, the processing time results of the experiments are shown in Figure 10. The x-axis corresponds to the query set, whereas the y-axis (in logarithmic scale) corresponds to elapsed time in seconds when the algorithms are executed on a DecStation 3100, which is approximately a 14 MIPS machine. In the figure, in addition to the total processing time of each algorithm for 1000 queries, we show the time spent to initialize the concept hierarchy in a preprocessing phase as discussed above. There are three major conclusions that one can draw from these results. First, the initialization time is insignificant compared to the total processing time (approximately two orders of magnitudes less), so performing it does not degrade performance in any way. Second, processing a randomly ordered set of queries is always slightly more expensive than processing a set of queries in a clustered order (set $qset_1$). This is due to the fact that with $qset_1$, the concept hierarchies are formed in a regular way without many changes (see Figure 9), which makes searching them more efficient. We should also note, however, that the difference is not all that significant and is similar in magnitude to the variations that are observed among different randomly ordered query sets. Hence, based on Figure 10, it is safe to conclude that query ordering does not affect



(a)



(b)

Fig. 9 Concept hierarchy sizes as a function of the number of incoming queries: (a) UNIMEM; (b) COBWEB.

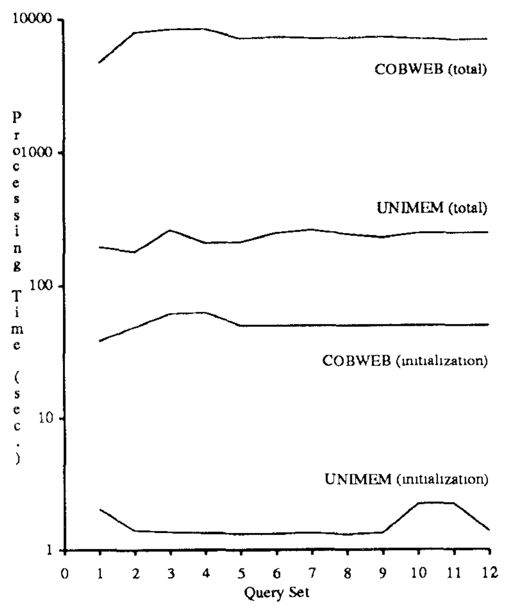


Fig. 10. Processing times of the learning algorithms.

performance very much. Third, as expected from the discussions in Section 2, UNIMEM is much more efficient than COBWEB, by more than an order of magnitude in all query sets. This is explained as follows. With respect to searching effort alone, UNIMEM is less efficient than COBWEB, because it can store examples in multiple places, which implies both that the generated hierarchies are larger in general than those of COBWEB and that larger pieces of them must be searched with every example. The efficiency of COBWEB's searches, however, is offset by the extreme cost of calculating the conditional probabilities, which is much higher than the cost of UNIMEM's comparisons of match scores. The problem is exacerbated by the fact that COBWEB requires a value for every known feature in the database (with UNKNOWN being used as the value of missing features). For our experiments, this implies that for every incoming example and for many nodes in the hierarchy, conditional probabilities for twenty-two features have to be calculated. Hence, as the number of incoming queries increases, the hierarchy grows larger and the algorithm becomes very expensive. This problem has been addressed by Gennari in the CLASSIT-2 algorithm [5], where an attention mechanism is used to determine the minimum number of features that need to be manipulated for deciding the appropriate modifications to the concept hierarchy.

5. SYSTEM COMPARISON

As mentioned in Section 2.1, for the most part, both UNIMEM and COBWEB satisfy all criteria that were given as mandatory for a learning algorithm to be useful for database design. In this section, we compare the two algorithms on their small differences with respect to those criteria and also on other important issues that were brought up by our experiments in an attempt to identify the one that is more appropriate for the desired task.

The main mandatory criterion on which the algorithms differ is the ability to deal with overlapping concepts. Many real-world concepts, like those represented by the queries in the earlier examples, are interrelated. Hence, this property is very important and is a strong point in favor of UNIMEM, which possesses it. On the other hand, COBWEB does not create overlapping concepts, and given the methods that it uses to identify concepts, it is unlikely that it could be easily modified to do so.

The next issue is the processing time consumed by the learning algorithms. The learning component of the system should be transparent to the user, so it must run in a fraction of the time that is needed by the system to answer a typical query. Clearly, the results of Figure 10 are overwhelmingly in favor of UNIMEM again, since COBWEB needs at least an order of magnitude more time for its expensive conditional probability calculations.

Another issue is the sensitivity of the resulting hierarchy to the order of incoming queries. In this respect, the two algorithms are roughly equivalent. Clearly, COBWEB is much more robust when starting from an empty concept hierarchy, but UNIMEM improves significantly and comes close to COBWEB when starting from a carefully initialized hierarchy. As we have mentioned already, for the specific application of database design, we believe that a preprocessing phase is very useful, essentially allowing the DBA to provide heuristics that will guide the learning algorithm. Hence, assuming that it is incorporated as part of the learning algorithm, either algorithm appears to be adequate.

A related issue to the above is the ability to delete concepts from the hierarchy. UNIMEM has this ability, whereas COBWEB does not, keeping all examples that it receives forever. On the other hand, such an ability is much more important to UNIMEM, which tends to generate many spurious concepts, than to COBWEB, which is more conservative. Hence, with respect to the quality of the produced concept hierarchies, it is unclear if UNIMEM, which allows concept deletion, should be considered superior to COBWEB, which does not. Of course, with respect to efficiency, COBWEB's practice can still lead to extremely large concept hierarchies, which is undesirable.

A final issue that could make a difference is the learning algorithm's predictive ability. The purpose of identifying concepts is to help the system in fine-tuning the design of databases. This assumes that the identified concepts are amenable to analysis that determines the most important ones, which should play major roles in the external and physical schemas of a database. For this task, the learning algorithms' differing representations play a major role. COBWEB makes the analysis very direct, since it maintains information that is necessary to calculate probabilities. These probabilities are very good measures of which concepts are most likely to appear in forthcoming queries. UNIMEM constructs essentially the same hierarchy and keeps some of the same information, but not in such a useful form for prediction as COBWEB does. Hence, in that sense, COBWEB appears to be preferable to UNIMEM.

The above discussion makes it clear that the question of superiority between UNIMEM and COBWEB cannot be conclusively answered. Both algorithms have different strengths. UNIMEM appears to be more efficient, has the power of overlapping concepts, can remove spurious concepts from its concept hierarchy, and takes advantage of a preprocessing phase that initializes the hierarchy. On the other hand, by using conditional probabilities, COBWEB generates more consistent and meaningful concept hierarchies, and appears to have better predictive power for making dynamic database redesigns. Hence, which of the two algorithms is more appropriate should depend on the importance of the various factors for the database concerned.

A hybrid algorithm might also be considered, using the UNIMEM hierarchy and also storing the conditional probabilities at each node in the hierarchy. This algorithm will probably feature the strengths of both algorithms and none of their weaknesses. Such a combination, however, requires very careful design and implementation, since there are several incompatibilities between the two algorithms that underly their basic functions. Thus, the viability of the hybrid algorithm is very much an open question.

6. SUMMARY AND FUTURE WORK

This paper has examined the idea of incorporating machine learning algorithms into a database system for monitoring its stream of incoming queries and dynamically modifying its physical and external schemas for improved system performance and user productivity, respectively. Two existing algorithms, UNIMEM and COBWEB, have been studied, since they appear to be suitable choices for the task. Standard UNIMEM and COBWEB implementations have been modified to support queries. These modified implementations provided a good basis for studying the practicality of using this approach.

The main conclusion from this work is that, given the right application domain, a database system with built-in learning would be advantageous. Learning would be most useful in cases where (a) the database contains a large amount of diverse information, so that many different concepts are identifiable, and (b) the importance of various concepts fluctuates over time, so that dynamic tuning is necessary. Another very important conclusion is that starting the operation of the system after a carefully constructed concept hierarchy has been established has very beneficial effects on the sensitivity of the final hierarchies to the order of incoming queries and the meaningfulness of the learned concepts. The initialized hierarchy should contain concepts that are expected to remain important throughout the life of the database, while the learning algorithm primarily deals with fluctuations on the importance of the remaining concepts.

On the other hand, in their current form, the learning algorithms may not be ready for incorporation into industrial-strength database systems for widespread use. On the positive side, it appears that concepts are usually grouped by the joins appearing in their stored queries, which is useful information for building indices and views. The inconsistency, however, between the concept hierarchies that result from differences in the order of incoming queries (even after a preprocessing phase) makes it difficult to predict the structure of the concept hierarchies. Also, another problem is the large storage requirements of the algorithms (every distinct query must be stored in the concept hierarchy for some period of time), which may adversely affect their running time and the overall system performance. Thus, although the potential exists for using this methodology to make effective decisions on dynamic database reorganizations, it may not be materialized until the aforementioned problems are overcome.

There are potentially several directions for future work. Database design is only one part of the database lifecycle that may benefit from machine learning. Others include monitoring the sizes of the actual results of queries so that predictions by the query optimizer in future queries can be more accurate, or deciding on what values of derived attributes should be cached so that they are not recomputed whenever they are requested by a query (which can actually be thought of as part of physical database design as well). An important investigation would be to incorporate learning techniques for improving the functionality or performance of a database system with respect to any of these aspects. For such an effort, different approaches to machine learning from the one that we have explored in this paper would most likely be needed.

Another important direction would be to develop learning algorithms whose output does not depend as much on the input example order. We want to once again emphasize that this is a key issue that needs to be addressed before such learning techniques can become practical. As mentioned before, one possible alternative would be a combination of UNIMEM and COBWEB that tries to preserve the nice features of both algorithms. As a complement to such an effort, experimentation with real-world database systems operating in a production environment would be required to assess the applicability of such tools.

Finally, some items of more immediate concern are extensions of the algorithms described in the paper so that their limitations (such as those mentioned in Section 3) are removed and the range of queries that can be dealt with by the learning algorithm expands.

REFERENCES

1. Astrahan, M. M. Blasgen, M. W., Chamberlin, D. D., Eswaran, K. P., Gray, J. N., Griffiths, P. D., King, W. F., Lorie, R. A., McJones, P. R., Mehl, J. W., Putzolu, G. R., Traiger, I. L., Wade, B. W., and Watson, V. System R: A relational approach to data management. *ACM Trans. Database Syst. 1*, 2 (June 1976), 97–137.
2. Biswas, G., Weinberg, J., Yang, Q., and Koller, G. Conceptual clustering and exploratory data analysis. In *Proceedings of the 8th International Workshop on Machine Learning* (Evanston, Il, June 1991), Morgan Kaufmann, Los Altos, Calif., 1991, pp. 591–595.
3. Borgida, A., and Williamson, K. E. Accommodating exceptions in database, and refining the schema by learning from them. In *Proceedings of the 11th International VLDB Conference* (Stockholm, Aug. 1985), Morgan Kaufmann, Los Altos, Calif., pp. 72–81.
4. Fisher, D. H. Knowledge acquisition via incremental conceptual clustering. *Mach. Learn. 2*, 2 (1987), 139–172.
5. Gennari, J. H. Focused concept formation. In *Proceedings of the 6th International Workshop on Machine Learning* (Ithica, N.Y., June 1989), Morgan Kaufmann, Los Altos, Calif., 1989, pp. 379–382.

6. GENNARI, J. H., LANGLEY, P., AND FISHER, D. Models of incremental concept formation. *Artif. Intell. 40*, 1-3 (Sept. 1989), 11–61.
7. HINTERBERG, H., NIEVERGELT, J., AND SEVCIK, K. C. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst. 9*, 1 (Mar. 1984), 38–71.
8. KOLODNER, J. L. Reconstructive memory: A computer model. *Cognitive Sci. 7*, 4 (Oct.–Dec. 1983), 281–328.
9. LANGLEY, P., THOMPSON, K., IBA, W., GENNARI, J. H., AND ALLEN, J. A. An integrated cognitive architecture for autonomous agents. Tech. Rep. 89-28, Univ. of Calif., Irvine, Sept. 1989.
10. LEBOWITZ, M. Concept learning in a rich input domain: Generalization-based memory. In *Machine Learning: An Artificial Intelligence Approach, Vol. II*. J. G. Carbonell, R. S. Michalski and T. M. Mitchell, Eds., Morgan Kaufmann, Los Altos, Calif., 1987, pp. 193–214.
11. LEBOWITZ, M. Experiments with incremental concept formation: UNIMEM. *Mach. Learn. 2*, 2 (1987), 103–138.
12. LI, Q., AND MCLEOD, D. Object flavor evolution through learning in an object-oriented database system. In *Expert Database Systems, Proceedings from the Second International Conference*, L., Kerschberg, Ed., Benjamin/Cummings, Menlo Park, Calif., 1989, pp. 469–495.
13. MITCHELL, T. M. Generalization as search. *Art. Intell. 18*, 2 (1982), 203–226.
14. MOTRO, A. Query generalization: A method for interpreting null answers. In *Expert Database Systems, Proceedings from the First International Workshop*, L. Kerschberg, Ed., Benjamin/Cummings, Menlo Park, Calif., 1986, pp. 597–616.
15. MOTRO, A. VAGUE: A user interface to relational databases that permits vague queries. *ACM Trans. Off. Inf. Syst. 6*, 3 (1988), 187–214.
16. QUINLAN, J. R. Induction of decision trees. *Mach. Learn. 1*, 1 (1986), 81–106.
17. STEPP, R. E., AND MICHALSKI, R. S. Conceptual clustering: Inventing goal-oriented classifications of structured objects. In *Machine Learning: An Artificial Intelligence Approach. Vol. II*, J. G. Carbonell, R. S. Michalski and T. M. Mitchell, Eds., Morgan Kaufmann, Los Altos, Calif., 1987, pp. 472–498.
18. VALDURIEZ, P. Join indices. *ACM Trans. Database Syst. 12*, 2 (June 1987), 218–246.

Received July 1991; revised December 1991; accepted February 1992