Learning Meta Knowledge for Database Checking*

Jeffrey C. Schlimmer
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213
(Jeff.Schlimmer@cs.cmu.edu)

Abstract

Building a large-scale system often involves creating a large knowledge store, and as these grow and are maintained by a number of individuals, errors are inevitable. Exploring databases as a specialization of knowledge stores, this paper studies the hypothesis that descriptive, learned models can be prescriptively used to find errors. To that end, it describes an implemented system called CARPER. Applying CARPER to a real-world database demonstrates the viability of the approach and establishes a baseline of performance for future research.

Keywords: Database consistency, meta knowledge, decision tree learning.

Introduction

Building a large-scale expert system often involves creating and extending a large knowledge base over the course of many months or years. For instance, the knowledge base of the XCON (R1) expert configuration system [Bachant and Soloway, 1989] has grown over the past 10 years from 300 component descriptions and 750 configuration rules to 31,000 component descriptions and 10,000 rules [Barker and O'Connor, 1989]. As knowledge stores like these grow and are maintained by a number of individuals, errors are inevitable; new individuals are unfamiliar with prior encoding conventions, any attempt to comprehend the details of the knowledge store as a whole are thwarted by its size, and the popular method of copy-edit adds entries that propagate errors.

Databases are a special case of knowledge stores and are widely used. For instance, XCON's component descriptions are represented in a database, where attributes describe the one or more entries per component. Errors in databases range from relatively obvious typographical and mathematical errors to subtle usage of legal but incorrect attribute values. In this paper I describe a system called CARPER that learns meta knowledge, or *models*, of how attributes are used to describe entries in a database and then uses these models to detect inconsistencies. After reviewing the state of database learning, I describe the system's architecture and its embedded learning methods and present results from applying it to part of XCON's database.

Related Work

Recently, there has been considerable interest in employing learning in the context of databases. Roughly, the interactions fall under the headings of data capture, query optimization, and consistency checking, though in many cases the same learning methods serve multiple purposes. Data capture is concerned with converting external representations into a consistent and complete form suitable for machine use. Researchers have studied this task in deductive databases, where an inference engine uses domain knowledge to implement a virtual extension of the database, and in pattern completion, where methods learn and apply a function that maps from known to unknown details [Hinton and Sejnowski, 1986].

By far, the greatest effort to date has been invested in applying learning to query optimization, to improve response speed, data accessibility, and response perspicuity. Speed issues are typically addressed by reordering conjuncts, caching responses, and restructuring the representations underlying the database. To improve the accessibility, researchers have recently begun applying learning to the task of reminding the user of previous entries given a new entry or query. One approach uses a case-based method that compresses old entries into a new, abstract one using a nearest-neighbor-like method [Fertig and Gelernter, 1989].

Researchers have also studied how to improve the perspicuity of responses in information and traditional database retrieval. The aim of the former to improve recall and precision of retrievals; inductive learning methods have been used to modify the keywords associated with documents and users in response to user feedback [Belew, 1987, Brunner and Korfhage, 1989].

*This research is supported by a grant from Digital Equipment Corporation and the National Science Foundation under grant IRI-8740522.

In the latter context of traditional database retrieval, researchers have focused on providing *intensional* answers to queries. For instance, [Shum and Muntz, 1989] formally derive a constructive definition of a minimal, abstract response given properties of taxonomy of domain terms. Their method uses an analog of the minimum description length principle to decide between answering a query with a list of individuals or with a counterfactual statement expressed using abstract terms. In a complementary line of research, [Chen and McNamee, 1989] use CART trees to estimate the number of entries that exhibit a particular property (i.e., the cardinality of a set) and the value of numeric attributes. This summary information can speed processing by simplifying the evaluation of query conditions.

Some queries, however, cannot be answered by direct look-up in the database, because individual values are missing or there is no corresponding attribute (or combination of attributes) in the database. For example, a doctor may wish to know which symptoms (represented as attributes in a database of patient records) indicate whether a treatment will be effective or not. In many ways, this is the simplest incarnation of learning in databases: the learning method uses a database as a set of examples from which to do induction. For instance, [Kaufman *et al.*, 1989] uses a suite of learning methods (e.g., characterization, discrimination, clustering, function learning, ...) which users may invoke to answer their questions. [Blum, 1982] presents a more sophisticated and autonomous approach in which simple statistics identify possibly interesting correlations. To test these, background knowledge is used to eliminate possible confounding factors, and if subsequent statistics indicate a true correlation, the finding is reported and included in the knowledge store of confounds.

In addition to data capture and query optimization, learning researchers are beginning to address issues of database consistency. At one end of the spectrum, [Li and McLeod, 1989] propose a database which addresses consistency with a sophisticated vocabulary of types and relations between attributes; users can invoke heuristics to learn new entity definitions and categories by rote and to relaxing typing in response to new representational demands. The system autonomously offers advice on where to strengthen typing. Others have proposed systems based on flexible consistency checking [Parsaye *et al.*, 1989], introducing the notion of *if-added* predicates which specify acceptable conditions for new attribute values. This type of meta information can be learned. For instance, [Parsaye *et al.*, 1989] describes a system that inductively learns certainty factor, propositional rules that map from observed values of database attributes to another attribute. These rules encode meta knowledge about the relationships between attribute values, and by including them as *if-added* predicates, the system as a whole is dynamically constructing highly-specific and potentially powerful constraints.

CARPER addresses database consistency in a similar manner, namely applying an inductive learning method to build meta knowledge about the relationships between attribute values and then using that knowledge prescriptively to identify inconsistencies in database entries. The approach offers a low maintenance, high utility consistency system which may be applicable even in poorly understood domains.

The Architecture

CARPER is an extensible system designed to find inconsistencies in databases. Central to the operation of CARPER are attribute *models* that capture how values are used to describe entries. For instance, a simple model might specify that the power consumed by a component should be an integer greater than or equal to zero. A more specific model might specify that power consumption is either 0 or between 15 and 40 watts. An even more specific model might specify that power consumption should be 20 watts if the component draws 3700 milliamps from a 5 volt power supply and 125 milliamps from a 12 volt power supply. CARPER allows the user to specify models, and it supplements these with models it learns by studying entries in the database. CARPER detects problems in the database by applying attribute models to database entries and generating predictions. If a particular entry violates a prediction, CARPER raises an alarm.

Figure 1 depicts the overall organization of CARPER. Some initial, definitional models are constructed by applying a simple YACC parser to the Scribe source of the *Database Dictionary*, a Digital Corp. document that describes XCON's database. These models constraint the domain and range of each of XCON's attributes. More specific models may also be encoded by hand (though none currently are).

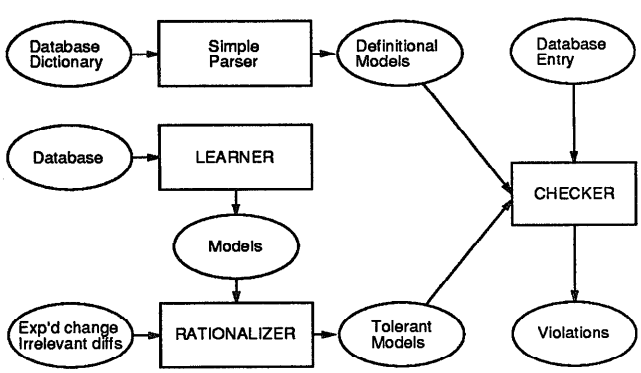

Figure 1: Schematic of Carper's Architecture. Boxes denote processes, and ovals denote data. Arrows indicate the source and sink of each type of data.

CARPER also uses inductive learning methods to

extract models of attribute use directly from entries in the database. These descriptive models are filtered by a rationalization process that takes into account knowledge about how components are likely to change over time (e.g., the average access time of disk drives tends to decrease) as well as meaningful differences (e.g., being off by one in BTUs is insignificant, but being off by one in the number of slots required by a set of boards is worth attending to). At present, this knowledge is sparse and simply encodes the percent increase and/or decrease that may be expected for numerically-valued attributes. The resulting models are more tolerant of variations in entry descriptions and partially compensate for the strict reliance of learning methods on past regularities.

CARPER applies the given and learned models prescriptively to find inconsistencies in database entries. Specifically, for a given entry, for each of its attributes, CARPER first applies the given models. If a prediction is violated, CARPER raises an alarm, prints some summary information, and goes on to check the next attribute of the entry. Otherwise, CARPER applies the learned models, reporting any violated predictions.

This simple model for checking a new entry's attribute values may also be used to find inconsistencies in entries already in the database. Simply remove each existing entry one at a time from the database, rederive any learned models, and check the entry as if it were new. In the experiments reported below, this is how CARPER checked entries from XCON's database. While not as comprehensive as a complete sequential reconstruction or a subset checking approach, it is applicable in situations where edit traces are not available, and it appears to be useful.

The Learning Methods

CARPER currently uses two inductive methods to learn models for each attribute in the database. The first is quite simple; it merely records all the values previously used for this attribute. For numerically-valued attributes, this information specifies a range of expected values in new entries. For nominally-valued attributes, it specifies a list of viable alternatives.

If simple range models do not indicate any inconsistency, CARPER uses an inductive method for learning from examples to construct additional models, where other entries are examples, and the classes to be learned are the values of the attribute being checked. In the absence of any information to the contrary, all other attributes may be in prediction. However, the user may indicate a coarse, determination-like relationship between the attributes by assigning them to one or more attribute groups. Attributes within a group may be used to predict others.

Using learned models to predict the value an attribute should have effectively relies on the assumption that the attribute values exhibit some sort of redundancy or inter-correlation. If this is not the case, inductive methods can only construct random models. While one might suppose that databases are designed from the outset to be minimal and non-redundant encodings of information, this is not the case with XCON's component database. For instance, Figure 2 depicts a number of first-order correlations between values of the numerical attributes that characterize Digital's cabinets.

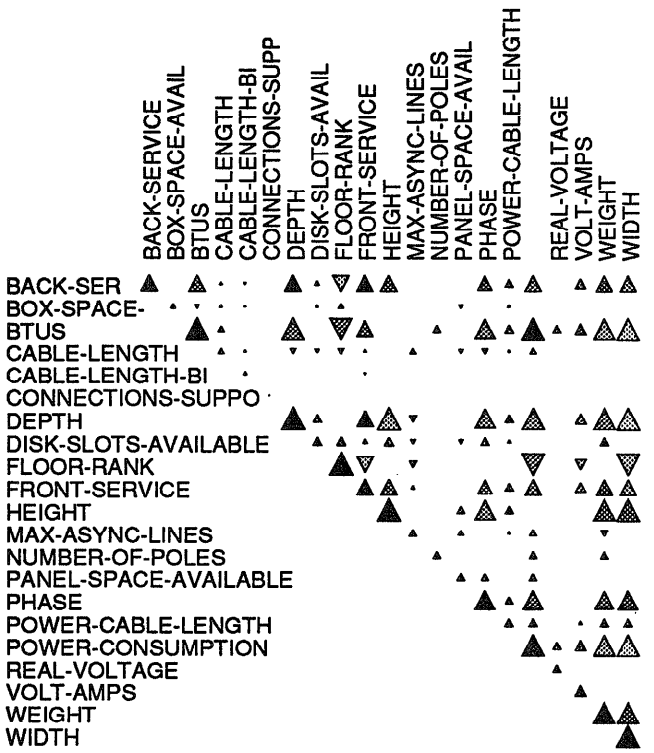

Figure 2: Correlations between numerical attributes describing cabinets. Triangles denote significant correlations, upward is positive, larger covers more data, and darker is stronger.

Learning from examples in CARPER

As an initial research strategy, CARPER uses a variant of C4 [Quinlan, 1987] to generate models that are expressed as Prolog rules. C4 is an efficient and robust inductive method for learning from examples. It initially constructs a decision tree to discriminate between examples from different classes, and then it converts the tree into a set of production rules. To construct a tree, C4 applies an information-theoretic evaluation function to heuristically select the single most discriminative attribute. This serves as the root of the tree, and C4 constructs a branch for each possible value of this attribute. The examples are then partitioned according to their value for this attribute, and the process is recursively repeated at each subtree until either there

are no more attributes to test or until all examples are of the same class. (Some decision tree learning methods use a pruning heuristic to stop tree construction when the examples cannot be reliably discriminated.) To convert the tree into a set of rules, C4 constructs an initial rule for each leaf in the decision tree. The conditions of the rule are the attributes tested along the path from the root to the leaf together with their corresponding values. The action of the rule is the most common class of the leaf. Each rule is individually pruned by iteratively testing the rule without each of its conditions. If its performance is not significantly worse, then the rule is simplified. The set of rules as a whole are also pruned by iteratively testing the rule set without each of its rules.

CARPER makes four modifications to the above scenario. First, pruning a rule set can introduce order dependencies between the rules, and this can make understanding one rule a function of the structure of others. Because rules are used to help explain a database discrepancy to the user, CARPER prunes individual rules but drops the rule *set* pruning step of C4.

Second, C4 is designed to build a set of rules that characterize all possibilities. Because CARPER generates rules to check a specific database entry, it recursively builds subtrees only for the values that correspond to the entry being checked. On XCON's processors, this simple optimization reduces the number of test-example comparisons by 39% (from 41M to 25M), the number of times the evaluation function must be computed by 37% (from 602K to 376K), and the total learning time (on a Sun 4) by 40% (from 2.4 to 1.4 days).

Third, C4's evaluation function is inappropriately biased in favor of attributes with large numbers of values. One solution is to use a number of binary-valued tests to encode the possible values of each attribute, putting all possible tests on the same ground. This is the default strategy for dealing with numerically-valued attributes (e.g., watts-drawn < 5), but it has been criticized as a strategy for dealing with nominally-valued attributes because it may make the intermediate decision tree unreadable. However, the final output in either case is a set of rules whose conditions specify a single value for an attribute. If it were computationally possible, considering a binary test for each possible subset of the attribute's values would probably yield the most concise rules.

Fourth, [Fayyad and Irani, 1991] outlines a refinement for binarizing numerically-valued attributes, noting that C4's evaluation function will always prefer a test that falls between two classes to one that separates examples from the same class. Extending this idea slightly, CARPER effectively enumerates the binary tests corresponding to all possible subsets of a nominal attribute's value and then only considers those that separate examples from different classes. Specifically, for each value of an attribute, CARPER collects the classes of examples with that value. By examining the set of classes each value is associated with, it combines values which map into the same set of classes (cf. Table 1). Discriminating between these values does not further partition example classes. The resulting value combinations serve as a set of binary-valued tests. On XCON's processors, this method reduces the number of tests by 57% from an average of 1,004 to 412.

Table 1: Binarizing a nominally-valued attribute A.

NOMINAL VALUES	V_i V_j V_k V_l	
EXAMPLE CLASSES	C_a C_b C_b C_b	
	C_b C_c C_c	
BINARY TESTS		
EXHAUSTIVE	DEFAULT	HEURISTIC
$A = \{V_i, V_j, V_k\}$	$A = V_i$	$A = \{V_i\}$
$A = \{V_i, V_j, V_l\}$	$A = V_j$	$A = \{V_j, V_l\}$
$A = \{V_i, V_k, V_l\}$	$A = V_k$	$A = \{V_k\}$
$A = \{V_i, V_j\}$	$A = V_l$	
⋮		

Fourth, C4's control structure builds a single tree, always selecting the single most discriminating attribute to test. If there is a tie between two possibilities, C4 chooses randomly. Figure 3 depicts histograms of C4's evaluation function applied to four attribute model learning problems (drawn again from XCON's processors). Both Distribution 1 and Distribution 2 indicate that the problem of ties may be more widespread than previously anticipated. This arbitrary preference for one attribute over another may result in attribute models that both miss existing database errors and falsely alarm on correct entries (cf. Table 2). CARPER follows a simple approach that eliminates this arbitrary selection and minimizes the number of false alarms by constructing attribute models that use each of the highly evaluated possible tests, in effect creating multiple trees instead of one. All tests that are above a fixed percentile (95% in all the results presented here) are used to construct a redundant set of rules. (This is similar to an approach used by [Buntine, 1989].) Because this may lead to a large number of rules, CARPER also imposes a small constant bound on the number of tests that may be selected at any choice point (3 in the results here). When these redundant (and potentially contradictory) rules are used to predict an entry's value for an attribute, all predictions are taken as equally valid, and CARPER only raises an alarm if the entry's value is not among them. Together with the policy of constructing only rules that correspond to values of the entry being checked, this results in about the same number of test-example comparisons, a 4% savings in the number of times the evaluation function must be computed, and a 16% savings in the total learning time. Even with the additional rules, the 16% savings is reflected in total checking time as well.

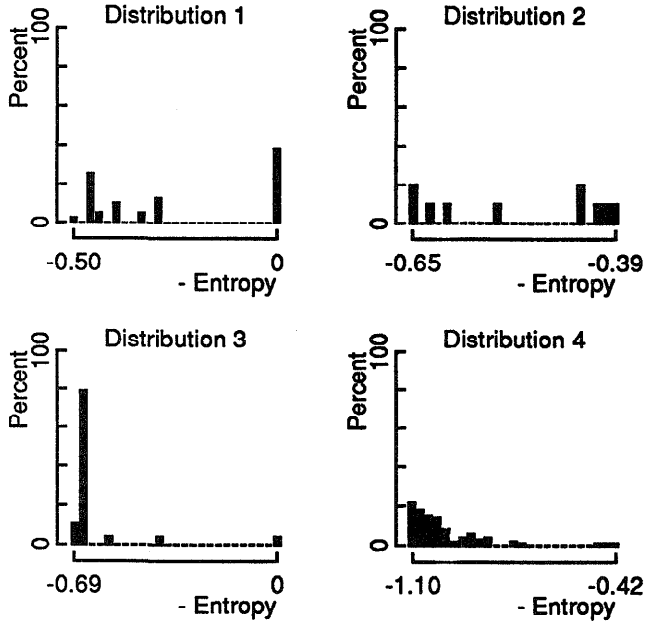

Figure 3: Histograms plotting the distribution of evaluation scores for tests in four model learning subtasks. Larger values (to the right) are preferable.

Table 2: Checking situations when both the attributes tested and the attributes predicted by a model may be in error.

PREDICTING ATTRIBUTE	TESTING ATTRIBUTE CORRECT	TESTING ATTRIBUTE ERROR
CORRECT	No Alarm	False Alarm False Positive
ERROR	Alarm	Missed Error False Negative

Initial Results

CARPER has been used to check XCON's database entries, and in this section I present results from applying it to check XCON's processors. The processors comprise one of 33 classes (others being printers, disk drives, cabinets, etc.) and are characterized by 41 attributes specifically defined for them and 44 more attributes defined for all classes. The attributes include nominally and numerically-valued attributes as well as three additional types: context-sensitive, list, and relational attributes. The latter two encode repetitious structures and relations between database entries, and account for 5 and 2 of the 85 attributes, respectively. To date, CARPER simply ignores these attributes. It does however, recode the nine context-sensitive attributes into nominally-valued attributes. These are first-order predicate calculus attributes which encode conditional properties of the components depending on the overall computing system being configured. A common example is the attribute denoting whether the component is sold as a single component or part of some larger package, and the value is conditioned on both the CPU and its operating system. Each instance of these predicate statements is reified into a proposition. It is not yet clear whether this simplifying step is appropriate.

For each processor, CARPER first checks its definitional models; these are based on the attribute's class definition (effectively its domain), its range, and whether or not it must have a value. The first two items of information are taken from the on-line *Database Dictionary*, and the third is provided by XCON's developers. Of the 4,338 attribute-values in the 87 processors, CARPER identifies 37 (1%) that violate definitional models. If the definitional models are satisfied for a particular entry, CARPER checks its simple range models. These models enumerate previously observed values as well as whether range or membership checking is more appropriate. CARPER identifies 93 attribute-values (2%) that violate range models. If both the definitional and simple range models are satisfied, CARPER consults the third, learning-from-examples model generated by the C4-like learning method described in Section . CARPER identifies 210 attribute-values (5%) that violate models that are learned by selecting the single best test. If redundant models are instead learned by selecting a number of good tests, CARPER identifies 131 alarms (3%), or approximately 38% fewer.

To get an idea of the utility of these alarms, 7 XCON developers examined CARPER's output for 20 processors drawn randomly from those with one or more alarm. Each alarm was rated as either being useful, useless, or unknown. The former include alarms that identify actual errors in database entries as well as those that point out inconsequential but real inconsistencies. For instance, XCON's rules automatically convert a value of NIL into either 0 or 1 if a numerical value is required in a rule's computation. Alarms from CARPER indicating that 0 was expected instead of NIL are counted as useful.

The XCON developers rate all of the definitional and range model alarms as useful. Of the 79 alarms raised on these 20 processors by models learned by selecting the single best test, 26 are rated as useful, 49 as useless, and 7 as unknown, yielding a hit rate of 49-57%. Learning redundant models raises the hit rate to 63-76% with 49 alarms, 19 useful, 25 useless, and 5 unknown, sacrificing 27% of the useful alarms for a savings of 43-46% of useless alarms. Table 3 summarizes these results. These results are surprisingly good given that this instantiation of CARPER uses meager knowledge to build definitional models and a relatively generic learning from examples method to build its learned models.

Table 3: CARPER's alarms and their rating by XCON developers.

LEARNING	USEFUL	USELESS	UNK	TTL
Single test	26	46	7	79
Redundant	19	25	5	49

Observations

These results demonstrate the viability of CARPER's general describe-prescribe paradigm. On the whole, XCON's developers are encouraged by the output from this initial version, but there is considerable room for improvement. Specifically, CARPER makes explicit use of the notion of the class of a database entry, learning models from only one class at a time. Because CARPER frequently false alarms (21 of 46 times) on two attributes that are known to be closely related a special subclass attribute, this indicates two new directions. First, CARPER's checking framework needs to be extended to naturally accommodate other explicit restrictions on which entries to use in learning models. Second, restrictions delimit subsets of entries that share additional attribute values but vary on others in systematic ways. This is particularly true of the subclass attribute, where all entries in a particular subclass correspond to the same basic mechanism packaged in different ways to adhere to power and containment environments. This information could be of considerable use in checking entries and should be learnable by inductive methods.

CARPER has been presented in the context of a specific database, but its methods are generally designed to be domain independent and extensible by domain specific knowledge. Although it can operate without any domain knowledge, early experiments indicated that this leads to unacceptably poor output, and the results presented here confirm the hypothesis that additional knowledge of the general checking task and the specific database may lead to substantially improved performance.

Acknowledgements Thanks to Cecilia Garbarino, John McDermott, Tom Mitchell, and Bonnie Ray for their consistent and productive involvement, to the AAAI reviewers for their constructive comments, and to the CMU SCS 'gripe' group for providing a reliable computing environment.

References

Bachant, J. and Soloway, E. 1989. The engineering of XCON. *Communications of the ACM* 32:311–317.

Barker, V. E. and O'Connor, D. E. 1989. Expert systems for configuration at Digital: XCON and beyond. *Communications of the ACM* 32:298–310.

Belew, R. K. 1987. *Adaptive information retrieval: Machine learning in associative networks.* Ph.D. Dissertation, University of Michigan, Cognitive Science and Machine Intelligence Laboratory.

Blum, R. L. 1982. Induction of causal relationships from a time-oriented clinical database: An overview of the RX project. In *Proceedings of the Second National Conference on Artificial Intelligence*, Pittsburgh, PA. AAAI Press. 355–357.

Brunner, K. P. and Korfhage, R. R. 1989. An automatic improvement processor for an information retrieval system. In Kerschberg, L., editor 1989, *Proceedings from the Second International Conference on Expert Database Systems*, NY. Benjamin/Cummings. 449–468.

Buntine, W. 1989. Learning classification rules using Bayes. In *Proceedings of the Sixth International Workshop on Machine Learning*, Ithaca, NY. Morgan Kaufmann. 94–98.

Chen, M. C. and McNamee, L. 1989. Summary data estimation using decision trees. In *Proceedings of the IJCAI Workshop on Knowledge Discovery in Databases*, Detroit, MI. 49–56.

Fayyad, U. M. and Irani, K. B. 1991. On the handling of continuous-valued attributes in decision tree generation. *Machine Learning* in press.

Fertig, S. and Gelernter, D. 1989. Musing in an expert database. In Kerschberg, L., editor 1989, *Proceedings from the Second International Conference on Expert Database Systems*, NY. Benjamin/Cummings. 605–620.

Hinton, G. E. and Sejnowski, T. J. 1986. Learning and relearning in Boltzmann machines. In Rumelhart, D. E. and McClelland, J. L., editors 1986, *Parallel distributed processing (vol. 1)*. MIT Press, Cambridge, MA.

Kaufman, K. A.; Michalski, R. S.; and Kerschberg, L. 1989. Mining for knowledge in databases: Goals and general description of the INLEN system. In *Proceedings of the IJCAI Workshop on Knowledge Discovery in Databases*, Detroit, MI. 158–172.

Li, Q. and McLeod, D. 1989. Object flavor evolution through learning in an object-oriented database system. In Kerschberg, L., editor 1989, *Proceedings from the Second International Conference on Expert Database Systems*, NY. Benjamin/Cummings. 469–495.

Parsaye, K.; Chignell, M.; Khoshafian, S.; and Wong, H. 1989. *Intelligent databases.* Wiley, NY.

Quinlan, J. R. 1987. Generating production rules from decision trees. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy. Morgan Kaufmann. 304–307.

Shum, C. D. and Muntz, R. 1989. Implicit representation for extensional answers. In Kerschberg, L., editor 1989, *Proceedings from the Second International Conference on Expert Database Systems*, NY. Benjamin/Cummings. 497–522.