Abstract-Driven Pattern Discovery In Databases

Vasant Dhar and Alexander Tuzhilin

Department of Information, Operations and Management Sciences

Leonard N. Stern School of Business, New York University

44 West 4th Street, New York, NY 10012

vdhar@stern.nyu.edu, atuzhilin@stern.nyu.edu

Abstract-Driven Pattern Discovery in Databases

Vasant Dhar

Alexander Tuzhilin

Information Systems Department
New York University
Stern School of Business
40 West 4th Street, Room 624
New York, NY 10003

vdhar@stern.nyu.edu, (212) 998-4209
atuzhilin@stern.nyu.edu, (212) 998-4213

Abstract

In this paper, we study the problem of discovering interesting patterns in large volumes of data. Patterns can be expressed not only in terms of the database schema but also in *user-defined* terms, such as relational views and classification hierarchies. The user-defined terminology is stored in a *data dictionary* that maps it into the language of the database schema. We define a pattern as a deductive rule expressed in user-defined terms that has a degree of certainty associated with it. We present methods of discovering interesting patterns based on *abstracts* which are summaries of the data expressed in the language of the user.

Keywords: pattern discovery, data abstraction, classification, generalization.

Working Paper Series
STERN IS-93-11

database, attribute values are replaced by the set to which they belong. Han et.al [HCC92] use a similar technique to search for dependencies among the abstracted attribute values and also incorporate a probability measure into the dependency. Our approach generalizes on Walker's and Han et.al's in that attribute values in an abstracted database can also be predicates or views of the original database, depending on multiple attributes. We also allow a variety of functions, such as summation, averaging, etc., to be used in addition to counting for aggregating attribute values. Other differences will be described after presenting our model in Section 5.

In order to describe pattern discovery, we first need a precise definition of a pattern. Certainly, there is no standard definition of the term in the literature. In trying to draw a common thread through a recent collection of papers on "Knowledge Discovery in Databases," Frawley et.al. [FPSM91] define patterns as follows:

> Given a set of facts (data) F, a language L, and some measure of certainty C, a pattern S is a statement S in L that describes relationships among a subset F_S of F with certainty C, such that S is simpler (in some sense) than the enumeration of all facts in F_S.

This definition is intentionally vague to cover a wide variety of approaches. For example, even a set of statistical parameters such as the mean and standard deviation for a collection of numerical values qualifies as a pattern with the above definition. In fact, any abstraction that in some sense summarizes the data would satisfy the above definition of a pattern. In contrast to this, we define a pattern in a more restricted sense, as a rule that has associated with it a degree of certainty. The precise form of the rule will be described in Section 3.

2 Data Dictionary

Consider the following application where a user may be interested in patterns in the data that are expressed not in terms of the schema of the database but in other terms:

Example 1 Assume a credit card agency stores its data in the CREDIT_CARD database that contains CUSTOMER and TRANSACTION files. The schemas of these files are shown in Figure 1. The CUSTOMER file stores all the information about the credit card customers, and the TRANSACTION file stores all the information about the transactions performed by these customers, such as customer name, merchant's name, merchant's type, and the amount and the date of a transaction.

□

1 Introduction

Our interest is in large scale business databases which grow by millions of records daily. While this data is recorded primarily for accounting purposes, executives are interested in leveraging them for other purposes such as analyses of trends in the data. For example, intelligent summaries of credit card and scanner data can greatly aid decisions about production, distribution, pricing, advertising, and promotions. Likewise, securities trading data can be monitored for patterns that might point to fraud or other irregularities. Clearly, with the massive volumes of data that flow into databases daily, the computer will play an increasingly important role in the analysis. The challenge, however, is for it to generate the "interesting" patterns which may be hidden deep in the data, based on the needs or interests of the user.

We have been working with an organization whose core business hinges on the collection and effective analysis of very large amounts of consumer purchase data. A major part of the revenue derives from standard statistical reports that are generated with SQL queries and report generators. Another, more human-intensive "help" service focuses on helping clients with special one of a kind requests that require analysts to dig deeper into the data in an open-ended way in order to detect interesting patterns.

One of our short term objectives is to make the computer do the open-ended probing on its own. This requires generating a broad range of summaries and recognizing relationships among variables that a user might find interesting. The novelty in our method lies in exploiting two types of knowledge, and in accommodating a range of inference methods. The first type is a user-defined vocabulary that provides relational *views* of the data and is used to express generalization relationships among different data types. For example, a credit card company can define Yuppie as a person whose age is less than 35 and who makes more than $80000 a year, or who has a Gold card. Also, we can define Wall-Street-Yuppie and Madison-Avenue-Yuppie as specializations of Yuppie. The second novelty is in how we use *abstracts*, which are summaries of the data expressed in terms of this vocabulary. The vocabulary and abstracts endow the system with the ability to search for patterns in terms of sets that are meaningful to the user, in effect, focusing the search. Finally, the inference procedure applied on the abstracts can be a generate-and-test or a standard statistical procedure such as the Logit model [The71]. The ability to use standard statistical techniques is important given that the data is often inherently noisy and the patterns therefore statistical.

The idea of using an abstracted database was first proposed by Walker [Wal80]. His approach made use of the fact that the domain of an attribute can be abstracted, i.e. for the PET attribute, dogs and cats are mammals, snakes and turtles are reptiles and so on. In Walker's abstracted

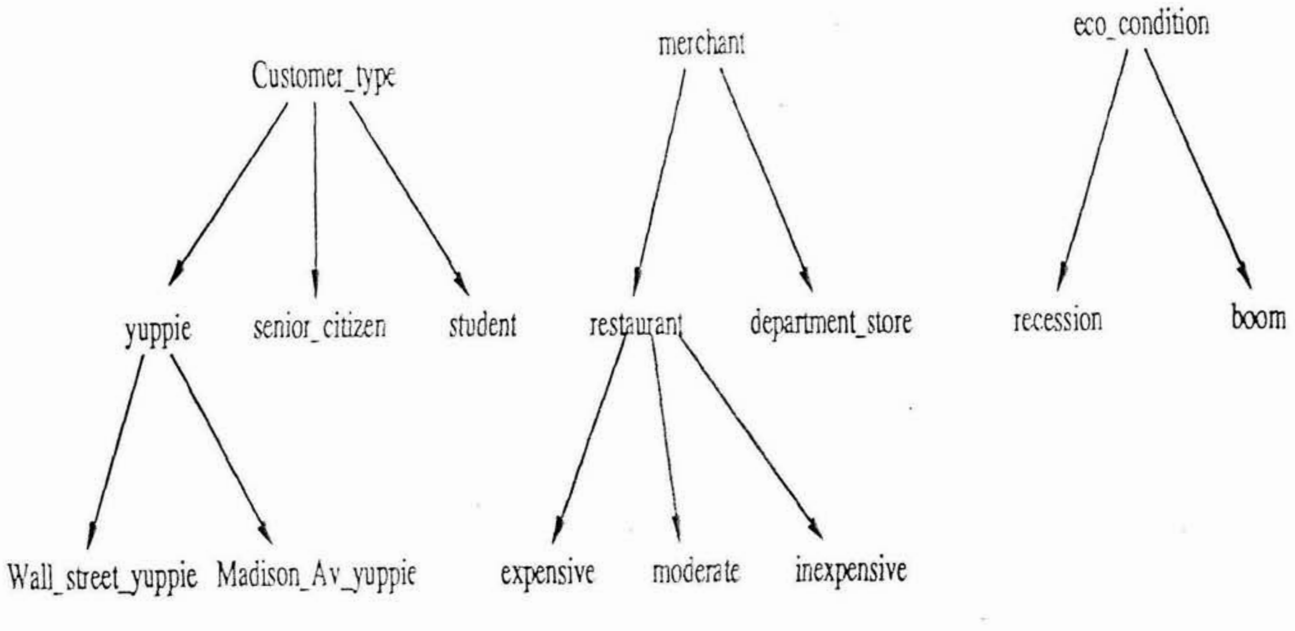

Figure 3: Classification Hierarchy for CREDIT_CARD Application.

i.e. $P < P'$ if P logically implies P'. For example, `Wall_street_yuppie` < `yuppie`.

Based on this partial order, we can build a classification hierarchy of user-defined predicates. The classification hierarchy for the vocabulary from Figure 2 is shown in Figure 3. Notice that siblings may not be mutually exclusive in a hierarchy. For example, a senior citizen can (sometimes) be a student.

However, we assume that the children of a user-defined predicate in the hierarchy form a collectively exhaustive set for the parent. This can be achieved by *implicitly* assuming an extra predicate *other* for each node in the hierarchy as the "catch all" condition. For example, we can implicitly define *other_yuppies*, *other_merchants*, etc.

The hierarchy enables the user to specify the level of analysis at which the system should focus. For example, a marketing manager can be interested in patterns at a national level, whereas a branch manager might be interested in a specific region.

2.3 Abstraction Functions

The third component of the data dictionary is the set of user-defined abstraction functions. An *abstraction function* of an attribute maps the domain values of the attribute into some other domain. For example, the abstraction function *year* maps a date into a year by "extracting" the year from

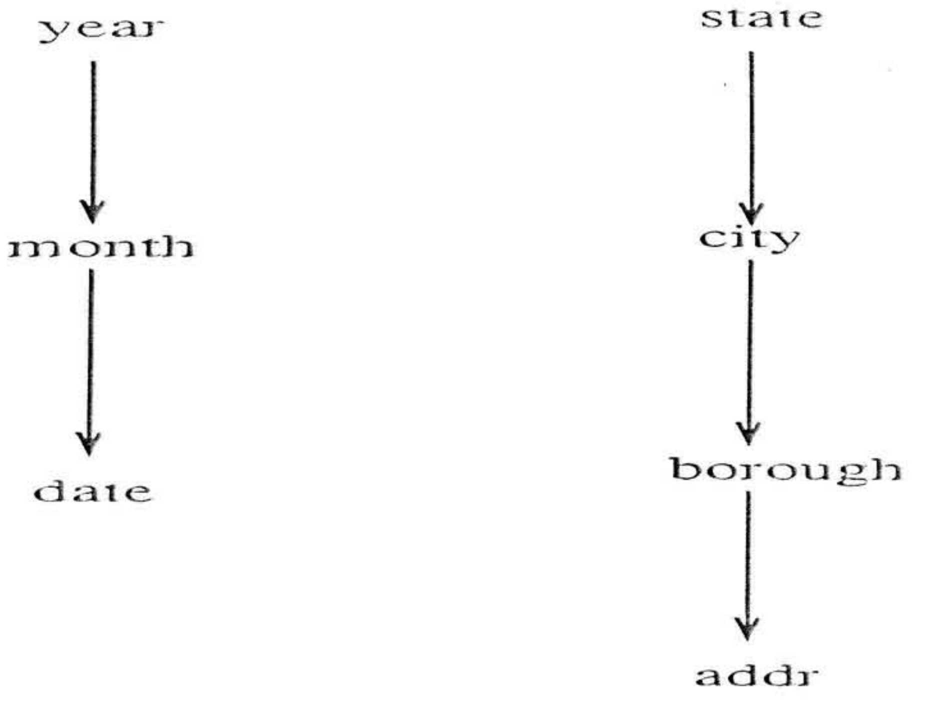

Figure 4: Abstraction Hierarchy for *Date* and *Addr* Attributes of the CREDIT_CARD Database.

the date. Similarly, the *city* function extracts the name of the city from a street address.

Furthermore, abstraction functions can be grouped into *abstraction hierarchies* by composition of abstraction functions. Examples of some of the abstraction hierarchies are presented in Figure 4.

In conclusion, a data dictionary for searching patterns contains the vocabulary consisting of user-defined predicates, a classification hierarchy based on the vocabulary, and a set of abstraction functions grouped into abstraction hierarchies. In the next section, we show how patterns are defined based on the data dictionary.

3 Patterns

We define a pattern along the lines of the machine learning literature [Win84] as a rule expressed in terms of the vocabulary with some degree of strength attached to it, i.e. as

$$P_1 \text{ and } \ldots \text{ and } P_n \rightarrow Q_1 \text{and} \ldots \text{ and } Q_k \qquad \text{(with strength } p) \tag{1}$$

where P_i, $i = 1, \ldots, n$ and Q_j, $j = 1, \ldots, k$ could be database relations, user-defined predicates, GROUP_BY constructs, or relational operators $=$, $<$, $\leq$, etc.[4]. Furthermore, we allow negations of predicates in the body but *not* in the head of a rule. Finally, p is a "measure of *strength*" that a certain pattern holds. We illustrate these concepts using some examples of patterns and then formally define the GROUP_BY construct and the strength of a pattern later in the paper.

Example 2

The pattern "`New York yuppies most likely live in Manhattan and have a Gold American Express card`" can be expressed as

```
CUSTOMER(name,addr,income,profession,age,card_type,mar_stat) and
yuppie(name) and city(addr) = ''New York'' →
borough(addr) = ''Manhattan'' and card_type = ''Gold AmEx''    (with strength 90%)
```

Intuitively, "strength 90%" means that 90% of New York yuppies live in Manhattan and have the Gold American Express card. We define the strength of a pattern in Section 4. Notice that the right-hand side of the rule contains a conjunction of two operators. Also notice that this pattern cannot be split into two patterns "New York yuppies live in Manhattan" and "New York yuppies have a Gold American Express card" because it is impossible to compute the strength of the combined pattern from the strengths of the two individual patterns. For example, as will be shown later in Section 4, we cannot determine what percentage of New York yuppies lives in Manhattan and has Gold American Express card from knowing the percentage of New York yuppies living in Manhattan and the percentage of New York yuppies carrying Gold AmEx.

□

The next rule presents an example of a pattern with a relational predicate appearing in the head of a rule.

Example 3 The pattern "`expensive restaurants in the Greenwich Village are mostly visited by yuppies`" can be expressed as

```
CUSTOMER(name,addr,income,profession,age,card_type,mar_stat) and
TRANSACTION(name,merchant,type,amount,data) and
expensive_restaurant(merchant) and city(merchant) = ''New York'' and
school_district(merchant) = ''Greenwich Village''
→ yuppie(name)      (with strength 60%)
```

In this example we also used the relation CUSTOMER besides the relation TRANSACTION be-

[4] We will also introduce additional statistical relational operators in Section 4.

cause yuppie is defined in terms of that relation.

□

We next provide an example of an aggregate pattern having the **GROUP_BY** construct.

Example 4 Consider the pattern ``yuppies spend more money in expensive restaurants than in any other type of restaurant (over the years).''

To express this pattern succinctly, we define the following three macros that specify total yearly spendings of yuppies in expensive, moderate and inexpensive restaurants respectively (the first parameter X in these macros specifies the year and the second Z specifies the amount spent):

YEARLY_YUPPIE_SPENDING_EXP_REST(X,Z) ≡
GROUP_BY([CUSTOMER(name,addr,income,profession,age,card_type,mar_stat) and
TRANSACTION(name,merchant,type,amount,data) and yuppie(name)
and expensive_restaurant(merchant) and year(date) = X], [X], [Z = SUM(amount)])

where the predicate **GROUP_BY**([expression], [X], [Z = SUM(amount)]) is similar to the **GROUP_BY** statement in SQL and specifies the sum Z of all the amounts taken over all the tuples in expression with the same value of X[5].

We define the other two macros YEARLY_YUPPIE_SPENDING_MOD_REST(X,Z) and YEARLY_YUPPIE_SPENDING_INEXP_REST(X,Z) similar to YEARLY_YUPPIE_SPENDING_EXP_REST(X,Z) by replacing predicate expensive_restaurant(merchant) in its definition by moderate_restaurant(merchant) and inexpensive_restaurant(merchant) respectively.

Then our pattern can be expressed in terms of these macros as

YEARLY_YUPPIE_SPENDING_EXP_REST(X,Z) and
YEARLY_YUPPIE_SPENDING_MOD_REST(X,Z′) and
YEARLY_YUPPIE_SPENDING_INEXP_REST(X,Z″)
$\rightarrow Z' < Z$ and $Z'' < Z$ (with strength 85%)

assuming that in 85% of the years yuppies spend more in expensive than in moderate and inexpensive restaurants.

This pattern could have been expressed in a more succinct form if we defined predicate YEARLY_YUPPIE_SPENDING(X,Y,Z) that defines the total amount Z spent by yuppies in the year X in a restaurant type Y (expensive, moderate, inexpensive). However, any definition of

[5]This form of **GROUP_BY** was originally proposed in [MPR90].

such a predicate would require the use of the second-order logic with quantification over predicates `expensive_restaurant`, `moderate_restaurant`, `inexpensive_restaurant`.

□

In general, the GROUP_BY construct is defined as follows [MPR90]. Let ϕ be a first-order formula, $X_1, \ldots, X_n, Y$ be variables appearing in ϕ and *aggr* be one of the standard aggregation functions COUNT, SUM, AVG, MAX, MIN, etc. Then **GROUP_BY**$(\phi, [X_1, \ldots, X_n], [Z = aggr(Y)])$ is a predicate depending on variables $X_1, \ldots, X_n, Z$ such that all the tuples in ϕ with the same values of $X_1, \ldots, X_n$ are grouped together, and the aggregation function is computed for the attribute Y within each group.

4 Searching for Patterns

Consider the pattern from Example 4 which states that yuppies spend more in expensive restaurants than any other type of restaurant. How can this pattern be detected? Since this pattern is based on total spendings of yuppies in different types of restaurants over a period of years, it is not apparent from the original relations CUSTOMER and TRANSACTION. To discover this pattern, we need an *abstract* of these relations that contains cumulative spendings of yuppies and maybe other customer types in different types of restaurants over the years. Figure 5 provides an example of such an abstract called SPENDING[6]. The CUST_TYPE column of SPENDING comes from the classification hierarchy (see Figure 3) and its values `senior_citizen`, `student`, and `yuppie` are relational views from the data dictionary (see Figure 2). Similarly, the REST_TYPE column comes from the classification hierarchy (see Figure 3), and its values `expensive`, `moderate`, and `inexpensive` are relational views defined in the data dictionary (see Figure 2). Finally, the column TOTAL_AMOUNT is obtained by adding the amounts of all the transactions of members of a certain customer group in a certain restaurant type in a certain year.

We call this kind of a table an *abstract* because it summarizes and abstracts the data in terms of high-level categories. For example, the first row in Figure 5 says that the amount that the CUST_TYPE category `senior_citizen` spent in the REST_TYPE category `expensive` in YEAR 1985 was $1.05 million. Note that an abstract can also be considered as a report [PS91]. It aggregates data in a form that can be useful to an executive. In fact, much of the consolidation of data involved in management reporting systems involves the generation of abstract-like tables which provide useful summaries of the data. Note that our definition of an abstract is similar but

[6]Even though we are focusing only on yuppies, the reason why other customer types appear in the abstract will become apparent shortly.

SPENDING (in millions)			
CUST_TYPE	*REST_TYPE*	*YEAR*	*TOTAL_AMOUNT*
senior_citizen	expensive	1985	1.05
senior_citizen	moderate	1985	1.82
senior_citizen	inexpensive	1985	0.92
student	expensive	1985	0.04
student	moderate	1985	0.18
student	inexpensive	1985	0.42
senior_citizen	moderate	1987	2.44
student	moderate	1989	0.56
yuppie	expensive	1985	7.41
yuppie	moderate	1985	7.39
yuppie	inexpensive	1985	4.32
yuppie	expensive	1986	8.54
yuppie	expensive	1987	8.93
yuppie	expensive	1988	9.61

Figure 5: An Abstract of Relations CUSTOMER and TRANSACTION.

more general than the attribute-oriented generalization of Han et. al [HCC92], a point we discuss more fully in Section 5.

The term "spend more" in the pattern from Example 4 requires clarification. If the total spending of yuppies in expensive restaurants in 1985 was only slightly higher than in moderate restaurants, e.g. 7.41 vs. 7.39, we might consider them for practical purposes to be equal (or not significantly different), whereas 7.41 would be *significantly* greater than, say, 4.3. Accordingly, we could restate the pattern from Example 4 to say that

> `yuppies spend` *significantly* `more money in expensive restaurants than in any other types of restaurants`

where we can interpret significance in the standard statistical sense using, say, the t-test [Kac86]. To do this, we assume that the spending of a yuppie for a meal in a certain type of a restaurant is a normally distributed random variable (its mean and variance can be computed in a standard way by examining all the transactions of yuppies in this type of a restaurant). Then the *total* spending of all the yuppies over a year in this type of a restaurant is also a normally distributed random variable because it is the sum of individual random variables. In order to test whether or not the total spending of yuppies in moderate restaurants x_{mod} is significantly less than the total spendings of yuppies in expensive restaurants x_{exp} in a year (e.g., if 7.39 is significantly less than 7.41 in 1985 in Fig. 5), we have to apply the t-test to these two random variables. This means that we have to test the null hypothesis $H_0 : \mu(x_{mod}) < \mu(x_{exp})$ (where μ is the mean of a random variable), and the answer to this test constitutes the meaning of "significantly more" in the pattern stated above.

As defined in Example 4, the *strength* of a pattern is a ratio of the number of tuples satisfying the head and the body of a rule to the number of tuples satisfying the body of the rule. To generalize this approach, the strength of a pattern $p \rightarrow q$ is defined as a *conditional probability* that the head of a rule is true given that the body of the rule is true, i.e. as

$$\begin{aligned} P(q \text{ is true} \mid p \text{ is true}) &= \frac{P(q \text{ is true and } p \text{ is true})}{P(p \text{ is true})} \\ &= \frac{\text{number of tuples satisfying conditions } p \text{ and } q}{\text{number of tuples satisfying condition } p} \end{aligned} \qquad (2)$$

For instance, if it turns out in Example 4 that in 17 years out of 20 yuppies spent significantly more in expensive restaurants than in moderate and inexpensive ones, then the strength of the pattern is 17 / 20 * 100% = 85%.

To summarize, we compare values such as means for statistically significant differences. We then compute the strength of a rule based on the number of values that turn out to be (significantly) different using conditional probabilities.

4.1 Abstracts

As mentioned above, one way to limit the search for interesting patterns is to let the user specify in broad terms the things to be considered (objects, aggregation functions, etc.) in deriving the patterns of interest. In particular, the user has to specify three types of information:

- the list of relational attributes and/or user-defined predicates the pattern should contain
- the list of abstraction functions the pattern should contain
- *aggregation principle* (or aggregation function).

User-defined predicates and abstraction functions were defined in Section 2. An *aggregation principle* specifies how observations in patterns of interest should be aggregated. For example, the user may specify `yuppies` and `expensive restaurants` as user-defined predicates, `year` as abstraction functions, and *summation* as an aggregation principle. This specification means that the user is interested in the cumulative spending patterns of yuppies in expensive restaurants over the years. In

	LOCATION		
CUSTOMER_TYPE	*CITY*	*BOROUGH*	*COUNT*
yuppie	New York	Manhattan	95,000
yuppie	New York	Queens	1,000
yuppie	New York	Brooklyn	3,500
yuppie	New York	Bronx	500
senior_citizen	New York	Manhattan	450,000
yuppie	Boston	Brookline	9,000
student	Los Angeles	Hollywood	2,000

Figure 6: Example of an Abstract.

this example, we considered summation as an aggregation principle. Examples of other aggregation principles are *counting*, *averaging*, *maximizing*, and *minimizing*.

Given the user-specified inputs containing user-defined predicates, aggregation functions and an aggregation principle, an abstract can be built from these inputs and the data as follows.

1. For each user-defined predicate and each abstraction function, create a column in the abstract. Furthermore, create an aggregation column based on the user-specified aggregation principle.

   For example, if the user specified `yuppie` as a user-defined predicate, `city` and `borough` as abstraction functions, and *count* as an aggregation principle, then the abstract has four columns, one for each of the inputs.

2. For each user-defined predicate selected by the user, consider all of its siblings in the classification hierarchy described in Section 2.2. For each abstraction function, determine its range. Form the Cartesian product of the sets of siblings for the user-defined predicates and the ranges of all the abstraction functions.

   For example, as is shown in Figure 3, the siblings of the predicate `yuppie` are `senior_citizen` and `student`, and their parent is `customer_type`. The abstraction function `city` defines the set of all the cities in the USA, and `borough` the set of all boroughs in these cities. Take the Cartesian product of all the combinations of all the customer types in all the boroughs in all the cities.

3. For each tuple *t* from the Cartesian product obtained in Step 2, formulate a conjunctive query against the database as follows. Each user-defined predicate in tuple *t* is a relational view, and the query that defines this view gives rise to a conjunct in the conjunctive query that

is being formed. Each component c in tuple t corresponding to the abstraction function f defined on attribute A gives rise to the conjunct $f(A) = c$.

For example, consider the tuple (**yuppie**, `New York`, `Manhattan`) from the Cartesian product obtained in Step 2. The query is formed based on this tuple as follows. `Yuppie(name)`, being a view, gives rise to the expression `age < 35 and income > 80000 or card_type = 'Gold'`. **New York**, belonging to the range of the abstraction function **city**, gives rise to the conjunct `city(address) = 'New York'`. The final query is

`(age < 35 and income > 80000 or card_type = 'Gold') and`
`city(address) = 'New York' and borough(address) = 'Manhattan'`.

4. Aggregate the values of the tuples in the answers to the queries formulated in Step 3 based on the aggregation principle specified by the user. If the aggregated value of a tuple is 0, then the corresponding tuple is removed from the abstract.

For example, if the aggregation principle is counting, then the aggregation field contains the number of customers belonging to a certain customer type who live in a certain borough in a certain US city. For example, if there are 95,000 yuppies living on Manhattan then we get a tuple (**yuppies**, `New York`, **Manhattan**, `95000`). Furthermore, if no yuppies live in Bismarck, North Dakota then the tuple (**yuppie**, `Bismarck`, x, `0`) will not appear in the abstract for any borough x in Bismarck.

An abstract for the user-defined predicate **yuppie**, abstraction functions **city** and **borough**, and counting as the aggregation principle is shown in Figure 6.

This definition of an abstract does not specify some procedural details (e.g. how to evaluate the queries described in Step 3). However, these details are straightforward, and we therefore do not describe all the details of the abstract generating algorithm based on this definition. Note that in this algorithm we have to formulate as many queries as there are elements in the Cartesian product formulated in Step 3 of the definition. It is quite possible that the size of this Cartesian product can be very large. Therefore, this naive algorithm is impractical.

4.1.1 A More Efficient Algorithm

We assume initially that the user-defined predicates and all of their siblings do not contain aggregate functions in their definitions. We will explain subsequently how to handle the case when aggregation functions are allowed in these predicates.

Since an abstract is built from the database relations, as a first step, we have to determine what parts of which relations should participate in building the abstract. This can be achieved by examining all the siblings of all the user-defined predicates specified by the user and determining the attributes of the relations involved in their definitions. Similarly, we have to include the attributes of all the abstraction functions specified by the user. For instance, in the previous example we have to determine relations and relational attributes that are used in the definitions of the user-defined predicates `yuppie`, `senior_citizen`, and `student`, and the attributes used in the abstraction functions specifying `New York` and `Manhattan`. `Yuppie` is defined in terms of `age`, `income`, and `card_type` attributes of relation CUSTOMER, `senior_citizen` in terms of the attribute `age`, and `student` in terms of the attribute `profession`. Finally, `city` and `borough` abstraction functions use the attribute `address`. This means that the abstract should be built from the relation obtained from the CUSTOMER relation by projecting it on the attributes `name`, `address`, `age`, `income`, `card_type`, and `profession`. Note that attributes can belong to more than one relation in general. This means that the relations containing any of these attributes should be joined and then projected on the attributes involved. We will call the resulting relation an *underlying relation*.

The algorithm that builds the abstract works as follows. Initially, assume that the abstract has no tuples in it (is an empty set). For each tuple t in the underlying relation do the following. For each sibling of each user-defined predicate specified by the user check if it is true for the values in t. For example, if the first tuple in the underlying relation is `(Jack, 125 3 Av New York 10017, 85K, financial analyst, 33, Gold AmEx, single)` then it should be checked whether `yuppie(Jack)`, `senior_citizen(Jack)`, and `student(Jack)` are true. Note that *each* sibling has to be checked because, as was pointed out in Section 2.2, their definitions may not be mutually exclusive (e.g., a person can be a senior citizen and a student). Also note that these checks can be done in *constant* time because we assumed that definitions of the user-defined predicates have no aggregate functions (e.g., to check if a person is a yuppie, one has to look only at this person's record). It should also be determined to which category abstraction functions belong. For example, it should be determined in which city and borough Jack lives (New York, Manhattan).

If a tuple[7] of user-defined predicates and abstraction functions satisfies *all* the checks, then see if it was already added to the set of tuples in the abstract. If the tuple has not been added to the abstract already, then add it. Furthermore, add an additional attribute to this new tuple based on the aggregation principle specified for the abstract. For example, if the aggregation principle is counting then associate a counter with this tuple and set it initially to one; if the aggregation

[7] Note that this tuple differs from the tuples in the underlying set.

principle is summation then associate a total sum with this tuple and set its initial value to the value of the summation attribute from the database. If the tuple was already added to the abstract before, then update the aggregation attribute based on the aggregation principle specified by the user for the abstract. For example, if the aggregation principle is counting then the counter associated with the tuple in the abstract is incremented by one.

For example, since `yuppie(Jack)` is true, `city(125 3 Av New York 10017) = ''New York''` and `borough(125 3 Av New York 10017) = ''Manhattan''`, we add the tuple (`yuppie, New York, Manhattan`) to the abstract and initialize the counter column to one since the aggregation principle is counting in this case.

This process is repeated for each tuple in the underlying relation. The process of building an abstract is terminated when all the tuples from the underlying relation are processed.

The time complexity of this algorithm is determined as follows. For each tuple in the underlying relation, we have to consider all the siblings in all the user-defined predicates. As was explained above, the check for whether a user-defined predicate is true for the current tuple (e.g. that `yuppie(Jack)` or `student(Jack)` is true) is done in constant time. Therefore, the overall time complexity of the algorithm is proportional to the size of the underlying relation times the sum of the numbers of all the siblings in all the user-defined predicates specified in the abstract. Typically, the latter component (the sum) is much smaller than the size of the underlying relation. In this case the algorithm is liner in the size of the underlying relation.

The algorithm above is linear in the size of the underlying relation because the user-defined predicates do not contain any aggregation functions. Suppose we change the definition of a yuppie to be a person who is younger than 35 and spends more than 30K annually on purchases with a Gold American Express card. In this case, we cannot determine that Jack is a yuppie, i.e. that `yuppie(Jack)` is true, in constant time because it is not sufficient to examine only Jack's record in the underlying relation but to sum all his purchases in order to answer this question. To solve this problem, we have to establish an index on person's name, retrieve all the transactions Jack did over a year and sum the values of his purchases. The time it takes to determine if Jack is a yuppie in this case is proportional to the number of transactions Jack did over the years and this can slow the performance of the algorithm. However, aggregation functions seldom occur in user-defined predicates in practice.

Finally, we want to point out a direct relationship between abstracts and the **GROUP_BY** construct discussed in Section 3: **GROUP_BY**$([\phi], [X_1, \ldots, X_n], [Z = aggr(Y)])$ defines an abstract having attributes $X_1, \ldots, X_n, Z$ and an aggregation principle defined by the aggregation

Major	Birth_place	GPA	*vote*
art	Canada	excellent	35
science	Canada	excellent	40
science	foreign	good	25

Figure 7: Generalization of the List of Graduate Students.

function *aggr*. We will use this observation later on in this paper when we discuss searches for patterns on abstracts.

4.2 Deriving Patterns from Abstracts

Patterns on an abstract can be discovered using various techniques. In this paper, we will consider the techniques based on the attribute-oriented induction, on statistical methods and exhaustive searches on the abstract. We will describe each of these methods in turn now.

4.2.1 Attribute-Oriented Induction

This approach was proposed in the language DBLEARN [HCC92] and works as follows. The original data is inductively generalized for various attributes in the relation using the classification hierarchies for the attributes. For example, student's majors can be initially generalized to hard sciences (physics, mathematics, etc.), natural sciences (biology, chemistry, etc.), humanities (literature, history, etc.) and so on. At the next level, they can be further generalized to arts and sciences. This data abstraction process continues until the original data is generalized so that only a few tuples remain. Then each tuple defines a rule. For example, the list of graduate students enrolled at a university can be generalized to three tuples shown in Figure 7 (from [HCC92]). The generalized data in Figure 7 produces the following three patterns:

$(\forall x) graduate(x) \rightarrow Major(x) \in art \wedge Birth\_place(x) \in Canada \wedge GPA(x) \in excellent$ [35%]

$(\forall x) graduate(x) \rightarrow Major(x) \in science \wedge Birth\_place(x) \in Canada \wedge GPA(x) \in excellent$ [40%]

$(\forall x) graduate(x) \rightarrow Major(x) \in science \wedge Birth\_place(x) \in foreign \wedge GPA(x) \in good$ [25%]

This data compression technique can be extended from the generalized relations of Han et. al to abstracts[8] described in this paper by successively building abstracts on abstracts until the abstract becomes small (we can use techniques from [HCC92] to determine when an abstract becomes small).

[8] We will describe the difference between our abstracts and the generalized relations from [HCC92] in Section 5.

Then. following [HCC92], each tuple in the final abstract becomes a pattern.

As [HCC92] shows, this approach can generate many interesting patterns. However, it sometimes tends to discover "overgeneralized" patterns, e.g. patterns about graduate students studying science from Canada as opposed to graduate computer science students from British Columbia, unless the user is willing to limit the search to graduate computer science students from British Columbia by "manually" specifying this in the DBLEARN statement

learn characteristic rule for Status = "graduate" and Major = "CS" and Birth_place = "BC"

Furthermore, DBLEARN cannot derive the patterns that have comparisons in their bodies. For example, it cannot derive the pattern "`more yuppies lived in Manhattan than in Queens over the past 15 years.`" This pattern can be expressed in our language as

```
(∀ X) (1977 < X < 1992 and NEW_YORK_YUPPIES(Manhattan, X, NUMB1) and
NEW_YORK_YUPPIES(Queens, X, NUMB2) → NUMB1 > NUMB2)      (with strength 100%)
```

where NEW_YORK_YUPPIES(borough,year,number) can be defined as

NEW_YORK_YUPPIES(X,Y,Z) ≡
GROUP_BY([CUSTOMER(name,addr,income,profession,age,card_type,mar_stat) and
TRANSACTION(name,merchant,type,amount,data) and yuppie(name) and
city(addr) = "New York" and borough(addr) = X and year(date) = Y], [X,Y], [Z = COUNT()])

and can be derived using the techniques to be described in Sections 4.2.2 and 4.2.3.

4.2.2 Statistical Methods

A fundamental statistical representation for studying associations between variables is the *contingency table* [Kac86]. In a 2-dimensional table for example, we might have values of customer types on one axis and residential boroughs on the other. The number of cells in the table depends on the number of partitions of each variable. A table showing the numbers of yuppies and senior citizens distributed over Manhattan, Queens, and Brooklyn would have 6 cells.

It is possible to test the association between customer type and borough by testing the *null hypothesis* [Kac86] that there is no association between the variables, in which case we would expect the customer types to be distributed equally across the boroughs in a random sample. The *chi squared test* [Kac86] tests for independence on the basis of deviations of actual values from expected values, where the expected values are based on the null hypothesis.

An abstract with k attributes is essentially a k-dimensional contingency table. It is therefore possible to perform *all* standard statistical tests on it that can be done on contingency tables. In particular, if there is a relationship among the variables (i.e. they are *not* independent), it is possible to specify a dependent variable and the independent variables, and extract a linear relationship showing the impact of each of the independent variables on the dependent variable. The form of the linear relationship is

$$Y = B_0 + B_1X_1 + \ldots + B_nX_n$$

where the magnitudes and signs of the coefficients $B_0, B_1, \ldots, B_n$ indicate the impacts of the corresponding variables on the dependent variable.

When the dependent variable is categorical (discrete), we are often interested in the *probability* that it has a certain (categorical) value, given the values of the independent variables. For example, we might be interested in the conditional probability that someone is a Manhattan resident given his customer type (i.e. yuppie) and spending level. In such cases, we must make sure that the value of the equation falls between 0 and 1. A popular transformation, called the *Logit* transformation [The71], first takes the *odds* for a certain value of the dependent variable occurring (odds and probabilities have a simple relationship, i.e. a probability of 0.75 means odds of 3:1) and applies the logarithmic transformation to it. Specifically, the above equation is transformed into the following:

$$log\frac{P}{1-P} = B_0 + B_1X_1 + \ldots + B_nX_n$$

where P is the computed probability value for the observation denoted by the variables $X_1, \ldots, X_n$. P is guaranteed to fall in the interval $\{0,1\}$.

Let us illustrate applicability of the Logit transformation to pattern discovery with a simple example showing the conditional probability of someone being a Manhattan resident given spending level and customer type. The relationship would be:

$$log\frac{P}{1-P} = B_0 + B_1X_1 + B_2X_2$$

where X_1 is the spending level and X_2 is the categorical 0/1 variable, where $X_2 = 0$ means that the person is a yuppie and $X_2 = 1$ otherwise. If $X_2 = 0$, the right hand side is $B_0 + B_1X_1$. If $X_2 = 1$, it is $B_0 + B_1X_1 + B_2$. The coefficient B_2 therefore represents the impact on the probability that a person lives in Manhattan as we change the value of the customer type from yuppie to some other value. For example, suppose that the parameters, based on a data set, have been estimated to have the following values: $B_0 = 0$, $B_1 = 0.002$, and $B_2 = -138.0$, where coefficients B_0, B_1, B_2 are obtained using the standard regression analysis techniques [The71]. For a yuppie with a spending level of \$70,000, the probability of being a Manhattan resident is 0.99, whereas for some other customer type, it drops to 0.33. From this, it is apparent that for an individual at the above level of spending, the customer type has a high impact on the borough of residence. In fact, such an analysis can be interpreted as a rule such as ``high spending New York yuppies live in Manhattan,'' which is similar to the pattern in Example 2. Similarly, it is possible to perform other types of analyses on the outputs of the Logit model by choosing different dependent and independent variables.

It should be noted that statistical techniques, such as the ones presented above, can be applied not only to the original data (as is typically done in statistics), but more importantly, to abstracts. The latter allows us to extract patterns in terms of the vocabulary of the user.

4.2.3 Exhaustive Search on an Abstract

In general, there is an infinite number of patterns that can be discovered on an abstract if attribute domains are infinite. Therefore, we must concentrate on some finite "interesting" subset of these patterns.

Our interest is in two types of patterns that capture a rich subset of all possible patterns. The first class of patterns does not contain any aggregates and can be generated from the abstracts with the *counting* aggregation principle. The second class of patterns consists of patterns on the abstract of the form

$$\text{ABSTRACT}(X_1, \ldots X_n, N_1) \text{ and } \text{ABSTRACT}(Y_1, \ldots Y_n, N_2) \rightarrow N_1 \theta N_2,$$

where $X_1, \ldots X_n$, $Y_1, \ldots Y_n$ are either constants or variables associated with regular fields of an abstract, and N_1 and N_2 are constants or variables associated with the aggregated field of the abstract, and θ is a relational operator $<$, $=$, etc. We consider these two classes of patterns in turn now.

Patterns without Aggregates. Assume that an abstract has *counting* as an aggregation principle. One general way to discover patterns without aggregates on the abstract is to "fix" some of its attributes and analyze the relationships among the remaining attributes by varying one or more of them and seeing the impact on the values of the others. We will illustrate this procedure on the abstract from Figure 6. Patterns on this type of an abstract can be searched as follows. Fix all the attributes in the abstract except one. For example, we can fix attributes CUSTOMER_TYPE to be "yuppie" and CITY to be "New York." We then compute the *conditional probabilities* of yuppies living in different boroughs. If NY_yuppie is defined as

$CUSTOMER\_TYPE =$ yuppie and $CITY =$ New York then the conditional probability that a yuppie lives in borough x of New York is

$$P(x) = P\{\texttt{NY\_yuppie} \text{ and } \texttt{BOROUGH} = x \mid \texttt{NY\_yuppie}\}$$

Substituting the numbers from Figure 6, we obtain the following conditional probabilities:

P(Manhattan) = 95%

P(Queens) = 1%

P(Brooklyn) = 3.5%

P(Bronx) = 0.5%

In general, if r is a condition describing the values of fixed attributes and q is the condition describing a free attribute and if p is the conditional probability, i.e.,

$$P\{q \text{ is true} \mid r \text{ is true}\} = p$$

then we can obtain the rule

$$r \rightarrow q \quad (\text{with strength } p)$$

For example, if r is the condition `NY_yuppie` and q is `BOROUGH` $= x$ then, since P(Manhattan) = 95%, we obtain the pattern

```
NY_yuppie → BOROUGH = Manhattan   (with strength 95%)
```

or in words: `New York yuppies most likely live in Manhattan.`

If we expand `NY_yuppie` in the previous rule then we obtain

```
CUSTOMER(name,addr,income,profession,age,card_type,mar_stat) and
yuppie(name) and city(addr) = ''New York''
→ borough(addr) = ''Manhattan''    (with strength 95%)
```

An example of another pattern obtained in a similar way is

```
CUSTOMER(name,addr,income,profession,age,card_type,mar_stat) and
yuppie(name) and city(addr) = ''New York''
→ borough(addr) = ''Bronx''    (with strength 0.5%)
```

or in words: `New York yuppies most unlikely live in the Bronx` [9].

Notice that these patterns do not contain aggregation (i.e. GROUP_BY constructs) because the abstract has counting as the aggregation principle which is used to compute the strength of a pattern. Furthermore, in this pattern the head of the rule contains the equality predicate because

[9] Notice that the strength of this pattern is 99.5% because it is a *negation* of the previous pattern. Also note that there is an interesting relationship between pattern strengths and fuzzy logic; however, we do not explore this relationship in the paper.

the free attribute is based on the aggregation function BOROUGH. If it were based on a user-defined predicate, then the derived pattern would have a predicate in its head.

By exhaustively fixing different values of different attributes and varying other attributes, we can obtain many different patterns. However, we will retain only the patterns with high levels of strength that is *a priori* specified by the user.

The method described above finds patterns without aggregates because the abstract has counting as the aggregation principle. If an abstract has any other aggregation principle, such as summation or averaging, then only patterns with aggregates (that have **GROUP_BY** construct) can be discovered on that abstract.

Patterns with Aggregates. Since there is a direct relationship between **GROUP_BY** constructs and abstracts that was discussed in Section 4.1, we will search for patterns on abstracts of the form

$$ABSTRACT(X_1, \ldots X_n, N_1) \text{ and } ABSTRACT(Y_1, \ldots Y_n, N_2) \rightarrow N_1 \theta N_2 \qquad (3)$$

where $ABSTRACT$ is the abstract being considered and θ is one of the comparison operators $>, =, \geq$, etc.. These types of patterns can be converted into patterns with **GROUP_BY** construct: **GROUP_BY**$([\phi], [X_1, \ldots X_n], [N_1 = aggr(Z)])$ and **GROUP_BY**$([\phi], [Y_1, \ldots Y_n], [N_2 = aggr(Z)])$ $\rightarrow N_1 \theta N_2$

where ϕ is the expression that is used to build the abstract, and *aggr* is the aggregation principle being used. We describe a conceptual method of finding patterns on abstracts now (efficiency issues are less important in this case than when we described how to build abstracts because abstracts are significantly smaller than the underlying database in most of the cases).

Patterns of the type (3) can be discovered as follows. First, form a Cartesian product of the abstract with itself. For example, in case of the abstract SPENDING from Figure 5 , the Cartesian product will have fields CUST_TYPE, REST_TYPE, YEAR, TOTAL_AMOUNT, CUST_TYPE', REST_TYPE', YEAR', and TOTAL_AMOUNT'. Then different patterns can be searched on this Cartesian product by fixing some of the attributes, varying others and comparing the corresponding aggregation columns. For example, we may fix the attributes CUST_TYPE and CUST_TYPE' to be yuppie and REST_TYPE to be expensive, vary attributes REST_TYPE' and YEAR, assume that YEAR = YEAR', and compare aggregated attributes TOTAL_AMOUNT and TOTAL_AMOUNT'. During the comparison process, we compute the percentage of cases in which the value of the field

TOTAL_AMOUNT is significantly greater than TOTAL_AMOUNT' while values of REST_TYPE' and YEAR range over all restaurant types and years respectively. Assume the percentage turns out to be 85%. This specifies the strength of the pattern. If the strength is higher than the one specified by the user then we retain the pattern. In this example we get the pattern

```
(∀ X) (∀ Y) SPENDING(yuppie,expensive,Y,N1) and SPENDING(yuppie,X,Y,N2) → N1>N2
(with strength 85%)
```

where variables X and Y range over restaurant types and years respectively, or in words

```
''yuppies spend more money in expensive restaurants than in any other type of
restaurant''.
```

4.3 Interestingness of Patterns

Since the data can contain billions of patterns, it is important to provide methods that limit the search for patterns. One way to solve this problem would be to provide some measure of "interestingness" of patterns and then search only for patterns interesting according to this measure.

As was pointed out before, one such possible measure could be the strength of a pattern. For example, the pattern saying that 95% of New York yuppies live in Manhattan is intuitively more interesting than the pattern saying that only 65% of New York yuppies live in Manhattan. However, strength by itself is not a sufficient measure of interestingness. The pattern saying that all the Manhattan yuppies live in New York has trivially the strength of 100%. However, this pattern is not interesting because it reflects the fact that Manhattan is a part of New York. In general, this type of problem arises because of a functional dependency (in the example above, between borough and city).

In our approach, part of the interestingness heuristic for focusing the system comes from the classification hierarchy. For example, if the user asks the system to find cumulative spending patterns of yuppies in expensive restaurants over the years, the system automatically includes in the search related nodes such as senior citizens, inexpensive restaurants, and so on. Intuitively, the search procedure really tries to answer the question "what is it about the data, raw or abstracted, that according to the user-specified aggregation functions, is interesting given a specific classification hierarchy?"

The discovered patterns are presented to the user (sorted in the order of their strength). If the system discovers too many strong patterns or the user is not satisfied with the results, the user can adjust the inputs in order to find more interesting patterns. For example, the user may want to search for cumulative spending patterns of yuppies in expensive restaurants during recessions. This interactive process of user specifying the types of patterns to search for, the system returning the patterns found, and the user "manually" adjusting the inputs based on the system feedback can continue until the system does not return patterns of interest to the user.

5 Related Work

A recent book by Piatetsky-Shapiro and Frawley [PSF91] contains a collection of articles on pattern discovery. It presents various approaches to this problem ranging from purely statistical approaches to the knowledge-based methods. We compare some of these methods and other related work to our approach to pattern discovery.

There has been much work done in the area of pattern discovery in the scientific arena. However, there are some fundamental differences between commercial and scientific data, in the types of patterns that one is trying to discover, and in the methods of discovery. First, much of the business data is qualitative or categorical, not numeric. It is not collected in a controlled manner, but is a by product of decisions about what data is necessary for business functions. Secondly, the patterns in a large business database tend to be inherently fuzzy, not precise mathematical relationships as in the natural sciences. It therefore makes sense to also include statistical techniques in addition to explicit enumeration techniques to extract patterns. Finally, the criteria for deciding what is "interesting" in scientific domains, which is the generator part of the generate-and-test, tend to be theory-based as in AM [Len77] and BACON [LBS81]. In the business arena, as illustrated in the examples, executives are usually interested in trends dealing with changes in aggregate-based functions, such as totals and averages, for the terms in their vocabulary that they are interested in investigating. This information provides some of the "interestingness" heuristics for focusing the pattern discovery system.

There are also other techniques in the machine learning literature referred to as "learning from examples" techniques such as Winston's learning program [Win75], Mitchell's LEX system that works with "version spaces" [MKKC86], and Quinlan's ID3 algorithm [Qui86]. Of these, the ID3 is most closely related to our work. It tries to generate a decision tree that explains the data. For example, given a table with information on yuppies, expensive restaurants, and spending amounts, it would generate a decision tree where each node would involve a test on a particular attribute, in effect partitioning the data. The leaves of the tree contain the classification, such a low spenders and high spenders.

One of the limitations of ID3 is that it does not deal well with noisy data. Specifically, the tree becomes overly complicated in order to account for the noisy instances. A related problem is that it cannot deal with inconclusive data, that is, when there are no rules that classify all possible examples correctly using only the available attributes. Uthurusamy et.al. [UFS91] propose that the solution to this problem is to use probabilistic rather than categorical rules. This essentially makes it a statistical approach, in the same spirit as our method which makes use of strengths.

More recently, there has been work done by Pearl and Verma [PV91] on finding causal relationships in the data. Pearl and Verma define causation among a set of variables as some type of a minimal model that is consistent with the joint probability distribution for these variables. Since there can be many such models that are consistent with a given distribution, they consider the intersection of these minimal models in order to postulate "strong" and "possible" causal relationships. The authors do point out as a caveat that "fitness to data ... is an insufficient criterion for validating causal theories." In other words, they claim that there is a difference between discovering patterns in the data and establishing causality between variables based on the data because causality assumes a certain degree of data independence, whereas pattern discovery is entirely data-driven. Nevertheless, we think the work of Pearl and Verma will prove to be very promising in *explaining* the types of patterns which we discover. In fact, from a user's point of view, the explanation is often more interesting than the pattern itself.

As we mentioned at the outset, our use of abstracts builds on the work of Walker [Wal80]. The concept of an abstract, as described by Walker, was independently used by Cai et.al [CCH91] and Han et.al [HCC92] who term this method "attribute oriented generalization" and extend it to deal with uncertainty. Our approach generalizes on the above by providing a more general concept of a pattern and considering other types of discovery procedures such as statistical pattern discovery and exhaustive searches, in addition to the attribute-oriented induction method proposed in [HCC92]. All the patterns introduced in [HCC92] can be expressed with our rules[10]. Furthermore, we extend the rules in [HCC92] as follows. First, their rules can only contain predicates from the base relations, e.g. CUSTOMER, and predicates from the classification hierarchy (concept tree in their terminology), e.g. city(x) or major(x). In contrast, we allow arbitrary predicates (relational views) from the data dictionary in our rules, e.g. yuppie(name), recession(date). Second, we extend patterns with aggregates (GROUP_BY construct) because many interesting patterns are expressed in terms of aggregates. Third, we extend the expressive power of patterns by supporting relational operators in the rules (an example of a rule expressible in our rule language that cannot be expressed with the rules of [HCC92] was presented in Section 4.2.1). Fourth, we support the

[10]Although Han et al. allow disjunctions in the head of a rule, and we don't, their rules can be converted into several of our rules, each rule containing only one conjunctive clause per head.

notion of statistical significance in patterns as, for example, in ``significantly more New York yuppies live in Manhattan than in any other borough.''

As we mentioned at the outset of the paper, it is important to use statistical techniques as part of a pattern discovery system, particularly when the data are noisy. Such techniques are theoretically well developed and pervasive in practice. The natural question, therefore, is why are they not sufficient? The answer is that they require the user to do too much work, thereby making exploratory data analysis cumbersome and hence reducing the likelihood of finding something that might even be "close" to what the user had as an initial hypothesis. Secondly, they require the user to provide data sets corresponding to the variables before the analysis is performed, as opposed to having a system construct them dynamically from a database depending on the variables under investigation. For example, when a statistical technique is being employed by the user to test the hypothesis that yuppies spend more on moderately priced restaurants than expensive ones, it is unlikely to discover that yuppies spend more than any other type of customer on any type of restaurant which might be an unexpected but interesting insight for the user. Also, the data set corresponding to yuppie, the view, must be explicitly provided prior to the analysis. In contrast, our method would generate this view when required by the search procedure.

In summary, our approach is to use *knowledge* about the domain to take user inputs and focus the search while at the same time guiding the discovery procedure to parts of the search space that are in some sense "close" to the inputs specified by the user (i.e. include senior citizens in the picture even though the user specified yuppies, and so on). In this way, the domain knowledge simultaneously focuses and broadens search to potentially interesting parts of the search space.

6 Conclusions

In conclusion, the model of pattern discovery we have described makes use of whatever domain knowledge is specifiable in terms of a classification hierarchy in order to focus search. Specifically, nodes in the hierarchy, which can be used to construct database views on the original data, serve as the basis for generating abstracts from which patterns can be derived in terms of the vocabulary of the user.

We also make use of the fact that what is often of interest to the user is a *comparison* of different subsets of the data, measured on the basis of some type of aggregation. The types of patterns presented in the examples represent a useful subset of patterns that the system can discover on its own. At the same time, the user can specify interactively, additional constraints or restrictions to be included as part of a pattern. This interaction between the user and the system can focus the search for interesting patterns, which is essential for dealing with large quantities of data.

Acknowledgments

We would like to thank Gregory Piatetsky-Shapiro for comments on the earlier version of this paper and Dennis Shasha for discussing the issue of efficient computations of abstracts.

References

[CCH91] Y. Cai, N. Cercone, and J. Han. Attribute-oriented induction in relational databases. In G. Piatetsky-Shapiro and W.J. Frawley, editors, *Knowledge Discovery in Databases.* AAAI / MIT Press, 1991.

[FPSM91] W.J. Frawley, G. Piatetsky-Shapiro, and C.J. Matheus. Knowledge discovery in databases: an overview. In G. Piatetsky-Shapiro and W.J. Frawley, editors, *Knowledge Discovery in Databases.* AAAI / MIT Press, 1991.

[HCC92] J. Han, Y. Cai, and N. Cercone. Knowledge discovery in databases: An attribute-oriented approach. In *Proceedings of the 18th VLDB Conference*, 1992.

[Kac86] S. K. Kachigan. *Statistical Analysis.* Radius Press, New York, 1986.

[LBS81] P. Langeley, G.L. Bradshaw, and H. Simon. BACON.5: The discovery of scientific laws. In *Proceedings of IJCAI Conference*, 1981.

[Len77] D. Lenat. Automated theory formation in mathematics. In *Proceedings of IJCAI Conference*, 1977.

[MKKC86] T. Mitchell, R.M. Keller, and Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1), 1986.

[MPR90] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *Proceedings of the VLDB Conference*, pages 264–277, 1990.

[PV91] J. Pearl and T.S. Verma. A Theory of Inferred Causation. In *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 441 – 452, 1991.

[PS91] G. Piatetsky-Shapiro, November 1991. Personal communication.

[PSF91] G. Piatetsky-Shapiro and W. J. Frawley. *Knowledge Discovery in Databases.* AAAI / MIT Press, 1991.

[Qui86] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[The71] H. Theil. *Principles of Econometrics.* John Wiley & Sons, 1971.

[UFS91] R. Uthurusamy, U.M. Fayyad, and S. Spangler. Learning useful rules from inconclusive data. In G. Piatetsky-Shapiro and W.J. Frawley, editors, *Knowledge Discovery in Databases.* AAAI / MIT Press, 1991.

[Wal80] A. Walker. On retrieval from a small version of a large database. In *Proceedings of the VLDB Conference*, 1980.

[Win75] P.H. Winston. Learning structural descriptions from examples. In P.H. Winston, editor, *The Psychology of Computer Vision.* McGraw-Hill, 1975.

[Win84] P. Winston. *Artificial Intelligence.* Addisson-Wesley, 1984.