Foundations of a Logic-Based Framework for Intelligent Data Analysis

Giuseppe Manco

March 2001
Thesis supervisors:
Dr. Fosca Giannotti
Prof. Franco Turini
Abstract

The objective of this thesis is to define a logic-based Knowledge Discovery Support Environment, i.e., a flexible knowledge discovery system with capabilities to obtain, maintain, represent, and utilize both induced and deduced knowledge. We present a logic database language with elementary data mining mechanisms to model the relevant aspects of knowledge discovery, and to provide a support for both the iterative and interactive features of the knowledge discovery process.

We study some distinguishing features of query languages that are conducive of modeling complex knowledge discovery applications, and identify such features in the Datalog++ logic database language and its practical implementation $\mathcal{LDL}++$. As a result, we propose the notion of iterative user-defined aggregate as a mean for (i) modeling typical data mining tasks as operations unveiling unseen knowledge, while (ii) dealing with specification and implementation at different abstraction levels. We illustrate the use of iterative aggregates to model specific data mining tasks, such as frequent pattern discovery, classification, data discretization and clustering, and show how the resulting data mining query language supports the modeling of typical steps of the knowledge discovery process, that range from data preparation to knowledge extraction and evaluation. Finally, we study the implementation of the resulting query language in the $\mathcal{LDL}++$ deductive database system, and provide a technique for copying with efficiency issues that may arise.

As a result, we obtain an integrated logical framework providing:

- a declarative unified view of data representation, extraction and reasoning techniques;
- a query system for knowledge extraction and reasoning that guarantees efficiency.
Acknowledgments

This thesis originates from an intense research activity within the Pisa KDD Lab group in the Department of Computer Science and the CNUCE institute of Pisa, and it is proper to tribute an acknowledgement to all its members. I wouldn’t have been able to develop any thesis without the collaboration with Fosca Giannotti, my terrific advisor. I must thank her for the stimulating (and sometimes vehement!) discussions, the continuous stimuluses she has been able to communicate to me, and for the opportunities she agreed to share with me during these years. Franco Turini, with his guidance and encouragement, has also been a valuable advisor, providing a continuous support during my Ph.D. years. Dino Pedreschi, too, played a fundamental role in the development of this thesis, with his availability and his collaboration. Working with Fosca and Franco and Dino has seriously contributed in growing up my research capabilities.

I would like to thank all my friends and colleagues at both the Department of Computer Science and the CNUCE institute, including Francesco Bonchi, Mirco Nanni and Salvatore Ruggieri. Mirco, in particular, has co-authored most of the papers (and software) from which this thesis originated, and its contribution has been very important for the development of the ideas that resulted in this thesis.

Carlo Zaniolo provided the $\mathcal{LDL}++$ system used in this thesis; his research has been an important seminal activity from which this thesis took the start-off, and I would like to thank him for his availability. I would also like to thank Haixun Wang for his support with the $\mathcal{LDL}++$ system.

Last but not least, I would like thank my family for the continuous support during these years, and in particular Irene, who accepted to share with me anguishes and excitements of my working activity for this thesis.
# Contents

1 **Introduction** \hspace{1cm} 1
   1.1 Motivations ................................. 1
   1.2 Main Contributions of the Thesis ........ 3
   1.3 Thesis Organization ....................... 5

2 **Knowledge Discovery in Databases** \hspace{1cm} 7
   2.1 The KDD Process .............................. 7
   2.2 Data Preprocessing .......................... 9
       2.2.1 Data Discretization .................... 11
       2.2.2 On-Line Analytical Processing ........ 15
   2.3 Data Mining .................................. 19
   2.4 Predictive Modeling ......................... 20
       2.4.1 Decision-Tree Classification .......... 21
       2.4.2 Bayesian Classification ............... 23
   2.5 Undirected Knowledge Discovery .......... 24
       2.5.1 Data Summarization .................... 24
       2.5.2 Data Segmentation ..................... 35
   2.6 Results Interpretation and Refinement .. 38
   2.7 Open Problems ................................ 39
       2.7.1 Technology ............................ 39
       2.7.2 Knowledge Extraction and Evaluation .... 40

3 **Deductive Databases** \hspace{1cm} 41
   3.1 Expressiveness and Semantics of Datalog . 42
   3.2 From Datalog to Datalog++ .................. 43
       3.2.1 Nondeterministic Choice ............... 43
       3.2.2 XY-programs ............................ 45
       3.2.3 A Semantics for Datalog++ ............ 46
   3.3 Query Answering with Datalog++ .......... 52
       3.3.1 User-Defined Aggregates ............. 54
   3.4 Iterative User-Defined Aggregates ....... 57
4 Towards a Logic-Based KDSE ........................................... 61
   4.1 The Role of Domain Knowledge .................................. 62
   4.2 Query Languages for Data Mining ................................. 63
      4.2.1 Relational Extensions ....................................... 64
      4.2.2 Deductive Databases ......................................... 66
      4.2.3 Inductive Logic Programming ................................. 69
   4.3 The Datalog++ Approach ........................................... 70
      4.3.1 Aggregates as Pervasive Concepts ............................ 73
      4.3.2 Logic-Based Inductive Databases ............................ 74
   4.4 Mining Aggregates ................................................ 76

5 Frequent Patterns Discovery ........................................... 79
   5.1 Task Formulation .................................................. 80
      5.1.1 The patterns Iterative Aggregate ............................ 82
   5.2 Using patterns in KDD ............................................. 86
   5.3 Special Cases ...................................................... 90
      5.3.1 Multidimensional patterns .................................... 90
      5.3.2 Item Hierarchies .............................................. 91
      5.3.3 Sequential Patterns ........................................... 92
   5.4 Clustering and Frequent Patterns ................................. 94
      5.4.1 Transactional Clustering ...................................... 97

6 Classification in Deductive Databases .............................. 99
   6.1 Task Formulation .................................................. 99
      6.1.1 The nbayes Iterative Aggregate .............................. 103
   6.2 Using nbayes in KDD ............................................. 106
   6.3 Generalizations ................................................... 109
      6.3.1 Simplification ................................................. 109
      6.3.2 Perceptron Learning .......................................... 110
      6.3.3 Decision tree classification .................................. 111
   6.4 Classification and Discretization ................................. 112
      6.4.1 Unsupervised Discretization ................................ 113
      6.4.2 Supervised Discretization ................................... 115

7 The Physical Level ..................................................... 119
   7.1 Problem Formulation .............................................. 120
   7.2 The $\mathcal{LDC}++$ System ......................................... 120
      7.2.1 Aggregate Implementation ..................................... 121
      7.2.2 User-Defined Predicates ..................................... 122
   7.3 Performance Analysis ............................................. 127
      7.3.1 Synthetic Data Generation ................................... 129
      7.3.2 Performance Evaluation ...................................... 130
   7.4 Optimizations .................................................... 134
8 Concluding Remarks \hspace{1cm} 139
  8.1 Summary \hspace{1cm} 139
  8.2 Open Problems and Further Research \hspace{1cm} 140

Bibliography \hspace{1cm} 143
List of Figures

2.1 The KDD Process ................................................................. 9
2.2 The beer_distributions table and example discretizations ............ 13
2.3 The ChiMerge discretization algorithm .................................... 14
2.4 ChiMerge Execution over the beer_distributions relation ............. 15
2.5 An example multidimensional view of a sales database ............... 16
2.6 Example application of OLAP Operators .................................. 17
2.7 General Decision Tree training schema .................................... 21
2.8 Apriori Algorithm for computing frequent itemsets .................... 27
2.9 Multidimensional and Transaction view of example Data .............. 29
2.10 GSP Algorithm for computing sequential patterns .................... 31
2.11 Profile analysis using association rules ................................... 36
2.12 The k-Means algorithm for computing data segments ................ 37

3.1 Iterated Stable Model Procedure ......................................... 50

4.1 Components of an Inductive database instance ......................... 71

5.1 Association rules on a sample transaction table ....................... 81
5.2 Multi-level rules over the transaction table ............................ 88
5.3 Frequent pattern discovery ................................................. 93
5.4 Clustering and frequent patterns ......................................... 95
5.5 Transactional Clustering example ......................................... 97

6.1 Sample playTennis table ..................................................... 101
6.2 classifier tuples for the playTennis table ............................... 102
6.3 Hash Tree segment for the attribute Outlook ........................... 106
6.4 Graphical schema of a Perceptron ....................................... 111
6.5 Discretization via iterative aggregates ................................... 113
6.6 Bayesian tables using ChiMerge ......................................... 118

7.1 Extended Rule Rewriting for Iterative Aggregates ..................... 121
7.2 Hash tree with items from transaction table ........................... 124
7.3 Hash tree management ..................................................... 125
7.4 Frequent itemsets and rules over synthesized data .................... 129
7.5 Performance Graph ......................................................... 130
7.6 $\mathcal{D} = 500, N = 500, |\mathcal{I}| = 100$. ........................................... 131
7.7 $\mathcal{D} = 1,000, N = 1,000, |\mathcal{I}| = 100$. ........................................... 131
7.8 $\mathcal{D} = 5,000, N = 3,000, |\mathcal{I}| = 100$. ........................................... 132
7.9 $\mathcal{D} = 10,000, N = 3,000, |\mathcal{I}| = 100$. ........................................... 132
7.10 $\mathcal{D} = 50,000, N = 5,000, |\mathcal{I}| = 100$. ........................................... 132
7.11 $\mathcal{D} = 100,000, N = 30,000, |\mathcal{I}| = 1,000, |\mathcal{T}| = 5$. ................. 133
7.12 Context switching overhead. ................................................................. 133
7.13 Performance of the hash approach (w.r.t. the data size). ....................... 134
7.14 Performance of the nested-loop approach (w.r.t. the data size). ............. 134
7.15 Curve fitting of the performances of the LDCL++ approaches. ............... 135
7.16 $\mathcal{D} = 100, N = 100, |\mathcal{I}| = 100, |\mathcal{T}| = 5$. ........................................... 135
Chapter 1

Introduction

1.1 Motivations

A Database is a reliable store of information. One of the prime purposes of such a store is the efficient retrieval of information, that in addition to the data stored in the database includes information that can be inferred from this, using two inference techniques:

- Deduction, a technique to infer information that is a logical consequence of the information in the database. Most database management systems, such as relational DBMSs, offer simple operators for the deduction of information. Extending the deductive expressiveness of query languages while remaining computationally tractable is pursued by Deductive Databases [146].

- Induction, a technique to infer information that is generalized from the information in the database. This is higher-level information, or knowledge general statements about properties of objects. We search in the database for regularities - combinations of values for certain attributes shared by facts in the database. In a sense, this regularity is a high-level summary of information in the database.

The most important difference between deduction and induction is that the former results in provably correct statements about the real world provided that the database is correct, while the latter results in conjectures that are supported by the database, but not necessarily true in the real world. The search for descriptions is called Data Mining when the amount of data under consideration is huge. Data mining is at the heart of Knowledge Discovery in Databases, an evolving research field that finds its roots in fields like statistics, machine learning and databases.

Research in data mining and knowledge discovery in databases has mostly concentrated on algorithmic issues, assuming a naive model of interaction in which data is first extracted from a database and transformed in a suitable format, and next
processed by a specialized inductive engine. However, such a decoupled approach has several drawbacks:

- It requires the data analyst to perform a preliminary preparation of data consisting in first extracting relevant data from the database, and next encoding data into a given format, suitable for the particular algorithm that will perform the analysis.

- It provides a fixed paradigm, that does not easily accommodate requirement changes, or implementation refinements.

- Extracted information is “embedded” in the data mining tool in an ad-hoc format.

On the other hand, current applications of data mining techniques highlight the need for flexible knowledge discovery systems, capable of supporting the user in specifying and refining the application-centric high-level objectives, combining multiple inductive strategies, and defining the quality of the extracted knowledge. Using knowledge on relevant context, enables the expert to respond rather precisely to application needs. A key issue is to obtain, maintain, represent, and utilize high-level knowledge in a unified framework. This comprises representation of domain knowledge, extraction knowledge and metaknowledge, its organization in ontologies, its creation through specialized algorithms, and its utilization for context recognition, disambiguation, needs identification, etc. Providing a sufficiently flexible model for such interactions is hence crucial in order to achieve a high degree of accuracy in application deployment.

The current literature has proposed various solutions to this problem. In our opinion, the approach combining deduction and induction reveals as one of the most promising approaches. A suitable integration, capable of amalgamating deduced and induced knowledge and providing powerful knowledge representation and reasoning formalism, permits a high degree of expressiveness. On the one hand, the deductive component can infer knowledge resulting from the underlying data, that can be generalized by the the inductive component. On the other hand, the induced knowledge can be further exploited to infer new knowledge.

In this respect, deductive databases are simple, powerful and versatile formalisms which originated from relational calculus and first order logic to increase the expressive power of the underlying logical model of data. Research in the design and implementation of deductive databases has led to expressive, practical and efficient query language and systems, which have supported the development of important applications, including data manipulations [158], data dredging [145], integrated heterogeneous information [62, 9].

The goal of this thesis is hence twofold:

1. to provide a logical model for integrating data mining techniques with deductive databases, well-suited both from a conceptual point of view and a physical point of view;
1.2. Main Contributions of the Thesis

The original contribution of this thesis is the systematic development of a logic database language with elementary data mining mechanisms to model the relevant aspects of knowledge discovery, and to provide a process view of data mining, i.e., to recognize the iterative and interactive features of knowledge discovery.

Our starting point is the definition of a **logic-based knowledge discovery support environment**, as an integrated formalism capable of

- rigorous definition of user interaction during the search process,
- separation of concerns between the specification and the mapping to the underlying databases and data mining tools, and
- understandable representations for the knowledge.

We show how the above features can be modeled by a logic-based language capable of expressing simple data mining primitives, as well as suitable representations of both domain knowledge and extracted knowledge.

The basic tool of our framework is the logic database language *Datalog++*. The name *Datalog++* is used in this thesis to refer to *Datalog* extended with mechanisms supporting:

- **temporal reasoning**, by means of temporal, or *stage*, arguments of relations, ranging over a discrete temporal domain, in the style of *Datalog$_{sl}$* [35];
- **nonmonotonic reasoning**, by means of a form of stratified negation w.r.t. the stage arguments, called *XY-stratification* [157];
- **nondeterministic reasoning**, by means of the nondeterministic choice construct [66].

We study the adequacy of the *Datalog++* logic language to flexible query answering, by introducing the distinguishing features of the language that are amenable to model complex applications.

One such feature is the notion of **iterative user-defined aggregate**, a refinement of the notion of user-defined aggregate provided in [157, 159]. We illustrate its semantics and expressiveness, and show how such feature allows to implement the theoretical notion of **inductive theory**, introduced in [105] to formalize the notion of **inductive database**. By using iterative aggregates to specify data mining tasks, one can:
• benefit from viewing the typical data mining tasks not as dynamic operations constructing new nuggets of information but as operations unveiling hitherto unseen by pre-existing pieces of knowledge;

• deal with the implementation issues of the data mining tasks at the desired abstraction level, thus obtaining both a powerful way of exploiting background knowledge in the search process and a viable middleware to efficient implementation.

We investigate how some important data mining tasks can be easily modeled in the Datalog++ framework provided with iterative user-defined aggregates. In particular:

• We face the problem of modeling frequent patterns discovery, that is one of the more widely studied data mining tasks in the database community, both for its database-centric nature [45] and for its challenging difficulty from an implementation point of view. We show how iterative aggregates allow a suitable representation formalism, as well as an efficient implementation by means of specialized algorithms.

• We provide a logic-based predictive modeling framework, based on the notion of iterative aggregate, and show how the expressiveness of the logic-based language allow to define iterative and interactive data classification features.

• We study how complex techniques can be easily combined in a uniform formalism, thus providing an amalgamation between deduced and extracted knowledge. Moreover, we show how multiple mining techniques (such as clustering and frequent patterns, or discretization and classification) can be easily combined in the resulting logic-based framework, thus providing a high level of expressiveness and flexibility.

Finally, we study the effectiveness of the approach both from an extensibility and an efficiency point of view. We show how the the \(\mathcal{LDL}++\) deductive database system, that implements most of the features of Datalog++, can be adapted to efficiently support specialized algorithms for some of the introduced data mining tasks in a transparent way. The resulting knowledge discovery system is comparable in performance, when restricted to “purely” relational fragments, to standard knowledge discovery systems.

As a result, we obtain an integrated logical framework providing:

• a declarative unified view of data extraction, analysis and reasoning techniques;

• a system for knowledge extraction that guarantees efficiency of data mining algorithms.
1.3 Thesis Organization

We proceed as follows. In chapter 2, a brief introduction to some relevant aspects of knowledge discovery is presented. We provide a database-oriented review of the most relevant techniques of the three main steps of knowledge discovery, namely data preprocessing, undirected knowledge discovery and data mining, and result interpretation and refinement.

Chapter 3 is devoted to the study of the semantics and its expressiveness of Datalog++. We review the declarative and procedural semantics of the language, and analyze the most relevant features that provide a suitable support to flexible query answering. In particular, we study the notion of iterative aggregate, by illustrating its semantics and expressiveness.

In chapter 4, we formalize the notion of logic-based knowledge discovery support environment. We review some proposals in the current literature, and show how the notion of inductive database allows a flexible theoretical basis for integrating induction and deduction from a database point of view. As a result, we show how an iterative user-defined aggregate provides a suitable interface for modeling inductive theories, allowing the formalization of specific mining tasks at different abstraction levels.

In chapter 5, we formalize the notion of frequent patterns discovery as an iterative aggregate, provide a viable specification of such an aggregate and show how the aggregate can model iterative and interactive aspects of the knowledge discovery process. Moreover, we show how the formalization can be extended to knowledge discovery tasks related to frequent pattern discovery. In particular, a formalization of clustering techniques is defined, and its relationship with the frequent patterns discovery task is highlighted.

Chapter 6 deals with Bayesian classification. We provide an iterative aggregate formalization of such a data mining task, and show how predictive modeling can be accomplished within the resulting language. We show how the technique can be generalized to further classification techniques and how to combine discretization and classification, so that the resulting classification framework has a general applicability.

Chapter 7 describes the practical implementation of the language in the $\text{LDCL++}$ system. We discuss some architectural alternatives, and measure the performance of the system with standard benchmarking tools. Finally, we envisage some substantial modifications to the logic abstract machine, in order to provide efficient support to application-specific optimizations.

Finally, in chapter 8 we summarize the features the approach and highlight some still open problems that are worth further investigations.
CHAPTER 1. INTRODUCTION
Chapter 2

Knowledge Discovery in Databases

Abstract

Providing easy navigation, exploration, summarization, or modeling of large databases is one of the goals of data mining. Current research results in algorithms that make it easy to analyze data for prediction and reporting purposes. Areas of emphasis include: classification, clustering, sequential data modeling, detecting frequent sequences and episodes in sequential event data.

Knowledge discovery denotes an “interdisciplinary” nature: it draws many of its basic principles from mature concepts in databases, machine learning, statistics, economics and sociology. By knowledge discovery, interesting knowledge regularities, or high-level information can be extracted from the relevant sets of data and be investigated from different angles. The discovered knowledge can be applied to information management, query processing, decision making, process control and many other applications.

In the following sections we shall give an overview to some of the characterizing aspects of knowledge discovery. The overview is obviously functional to the purposes of this thesis, and does not have the claim to be complete. We refer the interested reader to more complete overviews, such as [79, 86, 32].

2.1 The Knowledge Discovery Process

We can provide a concise definition of Knowledge Discovery in Databases, according to [46], as “the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data”. The definition includes the following concepts:

- knowledge discovery is multi-step Process, which involves data preparation, search for patterns, knowledge evaluation and refinement.

- The Data is a collection $F$ of facts (e.g., tuples in a database).
• A Pattern is an expression $E$ in a language $L$ describing facts in a subset $F_E$ of $F$.

• The discovered patterns should be Valid on new data $F' \neq F$ with a certain degree of certainty. Furthermore, they should be Novel and lead to some Useful actions.

• As the main goal is to understand the underlying data, the discovered patterns should be understandable.

As we concentrate on the “process-oriented” definition of knowledge discovery, we can highlight some essential characteristics (shown in fig. 2.1):

• KDD is a multi-step process, composed by four main phases: data consolidation, data preparation, data mining and result interpretation.

• KDD is an iterative process, where each step can inspire rectifications to the preceding steps. For example, the evaluation of the results of mining some kind of data can influence the data preprocessing techniques, suggesting, e.g., the application of suitable data transformation techniques.

• KDD is an interactive process. Interpretation and refinement, quality definitions, and even data processing techniques (e.g., visual data mining) require a strong user interaction.

In the following section we shall concentrate on the following aspects:

• Data preprocessing. In section 2.2.1 we analyze an example of data transformation technique, namely data discretization. Section 2.2.2 is devoted to show the OLAP framework, widely recognized as the most attractive source of information for many kinds of mining activities [76].

• Data mining. Section 2.3 provides an overview of two classes of data mining tasks: predictive modeling (studied in section 2.4) and undirected knowledge discovery (in section 2.5).

• Interpretation and Evaluation. In section 2.6 we briefly overview some problems that may arise in trying to understand and refine the results of mining activities.

As a final result, section 2.7 highlights some of the currently open problems in the KDD literature.
2.2 Data Preprocessing

A famous motto in knowledge discovery cites “Garbage in ⇒ Garbage out”, with the significance that noisy, inconsistent, insignificant data can hardly produce relevant results. This simple statement highlights the importance of the preprocessing phase in data mining, that can be defined as a combination of:

- **data integration**, where multiple heterogeneous data sources may be integrated into one.

- **data selection**, where data relevant to the analysis task are retrieved from the database.

- **data cleaning**, which handles with noisy, missing, or irrelevant data.

- **data transformation**, where data are transformed or consolidated into forms appropriate for mining by performing summary or aggregation operations.

Each of these aspects have been extensively studied in literature [10, 44, 79] and we shall not go into the details of them. However, it should be noticed that, no matter which data preprocessing technique we apply to data sources, in order to formalize the above steps in a unified framework we need a logical model of the data to be processed. We take as the starting point of our framework the relational model of data. We assume the existence of an infinite set of attributes, and, for each attribute $A$, an infinite set of values, called the domain of $A$ (dom($A$)). A
relation scheme is a finite set of attributes $A_1, \ldots, A_n$. The Cartesian product of domains $\text{dom}(A_1) \times \ldots \times \text{dom}(A_n)$ is the set of tuples $\mu = (X_1, \ldots, X_n)$ such that for $i = 1, \ldots, n$, $X_i \in \text{dom}(A_i)$ (and $X_i$ is denoted as $\mu[A_i]$). A relation is any finite subset of the cartesian product of one or more domains. A database instance is a finite set of relations. Given a relation schema $R = A_1 \ldots A_n$, a relation $r$ over $\text{dom}(A_1) \times \ldots \times \text{dom}(A_n)$ is said to be an extension of $R$, and $\text{cod}(A_i)$ represents the set of values of $\text{dom}(A_i)$ that appear in $r$. Data can be manipulated in a relational database using the “procedural” relational algebra, or its equivalent declarative relational calculus.

We can hence define the information sources of the knowledge discovery process as a set of relational extensions, and preprocessing tasks can be thought as applications of operators to such extensions.

- Information integration consists in the definition of the basic relations and the population of the instances of such relations. Difficulties may arise from many perspectives:
  - different databases may model the same concept, but use different terms for the same concepts
  - information sources, such as the web, may have no fixed schemas.
- Data selection, cleaning and transformation correspond to the application of relational operators (projection, selection and join), as well as the computation of statistics, to the given extensions, in order to obtain views relevant to the analysis tasks.

In practice, relational query languages such as SQL can be used to specify the above operations.

**Example 2.1.** Pareto diagrams [26] represent data frequencies. Such diagrams can be easily built by means of simple SQL queries. Given a relation $R$, the query

```
SELECT $A_i$, COUNT(*)
FROM $R$
GROUP BY $A_i$
```

computes the frequencies of the distinct values of the attribute $A_i$.  

**Example 2.2.** A typical data cleaning step can consist in the removal of outliers, e.g., elements that differ more than two times the standard deviation from the mean. Using SQL, we can identify such outliers in a straightforward way: given a relation $R$, the above query

```
SELECT $A_1, \ldots, A_n$
FROM $R$
WHERE $A_i > (\text{SELECT AVG}(A_i) \text{ FROM } R) + 2 \times (\text{SELECT STDEV}(A_i) \text{ FROM } R))$
```

computes the tuples corresponding to outliers related to attribute $A_i$.  

2.2.1 Data Discretization

A particularly interesting data preprocessing technique is data discretization. The case of data discretization can be considered as a typical example of the problems that raw data presents, and that can be overcome by suitable data preparation:

- originary data can be very sparse, whereas discretized data can be grouped into small sets.
- originary data can be more difficult to interpret than discretized data.
- data mining algorithms can be less efficient (or even unappplicable) with continuous data [41].

Roughly, data discretization can be used to reduce the number of values for a given continuous attribute, by dividing the range of the attribute into intervals. Interval labels can be used to replace actual data values. Discretization techniques can be distinguished into [88, 41]:

- supervised, or unsupervised, depending on whether instances have labels associated, and discretization aims at preserve label homogeneity,

- top-down or bottom-up, depending on whether intervals are formed by progressive cutting or merging,

- dynamic or static, depending on whether discretization is applied independently or embedded in a data mining task (e.g., classification).

A typical univariate discretization process (i.e., considering one attribute) consists of two steps: interval bins computation and view population. The intervals computation further consists in four steps:

1. sorting the continuous values of the attribute to be discretized,
2. choosing a cut/merge-point,
3. splitting/merging the interval at some point,
4. evaluating a stopping criterion.

In the view population, we build a view of the originary data source, in which the values of the attribute to be discretized are substituted with the identifiers of the corresponding intervals. For example, if the table Intervals contains the intervals obtained from the attribute \( A_i \) of a table \( R \), the following query populates the view corresponding to the discretized version of \( R \):

```sql
SELECT R.A_{1}, \ldots, R.A_{i-1}, \text{Intervals.Bin}, R.A_{i-1}, \ldots, A_{n}
FROM R, Intervals
WHERE Intervals.Low \leq A_i < Intervals.Sup
```
The way in which the intervals table is populated strongly influences the significance of the view to the purposes of the analysis objectives. Let us consider in deeper details the how discretization techniques may differ according to discretization objectives.

**Unsupervised Methods**

Unsupervised techniques are the simplest methods to discretize a continuous-valued attribute; given a dataset, we discretize the target attribute in \( k \) homogeneous intervals, where \( k \) is provided as input. Homogeneity here refers to data distribution: we aim either at homogeneous values of the intervals (like in *natural binning*), or at homogeneous frequencies in the intervals (like in *equal-frequency binning*).

The significance of such discretization techniques is highly influenced by the number of intervals: too little intervals may cause loose of information concerning the distribution of data; on the other side, if we define too many classes we have data dispersion. The problem of optimal values for the number of intervals has been extensively studied: for example, \([84]\) suggest to exploit the formula \( C = 1 + \frac{10}{3} \log_10(N) \) for the number of classes, or, as an alternative, the formula \( \delta = 3.5 \times \frac{s}{\sqrt{N}} \) (where \( s \) is the sample variance) for the optimal width of an interval. Other approaches are based on clustering or cross-validation \([150]\).

**Natural Binning.** Natural (or equal width) interval binning is perhaps the simplest method to discretize data. It consists in computing the extrema values of a continuous attribute and dividing the range of observed values of the attribute into \( k \) equally width bins, where \( k \) is a parameter supplied by the user. If an attribute \( A \) is observed to have values bounded by \( x_{\min} \) and \( x_{\max} \), natural binning computes the interval width \( \delta \)

\[
\delta = \frac{x_{\max} - x_{\min}}{k}
\]

and assigns each value \( x \in cod(A) \) to the interval \( j \) such that

\[
x \in [x_{\min} + j\delta, x_{\min} + (j + 1)\delta)
\]

**Example 2.3.** Let us consider the *beer_distributions* relation with schema and extension in fig. 2.2 a). A natural binning discretization into four intervals can be obtained by setting the interval width to 15. We obtain the four intervals \( 1 = [100, 115) \), \( 2 = [115, 130) \), \( 3 = [130, 145) \), \( 4 = [145, 160] \). The resulting attribute is transformed as shown in fig. 2.2 b).

The method can be thought as an example of top-down method: we start with a unique interval \( [x_{\min}, x_{\max}] \), and split it into equally sized intervals. Since the number of intervals is known, there is no need to specify the stopping criterion. As pointed out in \([41, 27]\), the main drawback of such an approach is high sensitivity to outliers\(^1\).

\(^1\)Generally, to cases where the distribution is not uniform.
2.2. DATA PREPROCESSING

<table>
<thead>
<tr>
<th>bar</th>
<th>beer</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Bud</td>
<td>100</td>
</tr>
<tr>
<td>A</td>
<td>Becks</td>
<td>120</td>
</tr>
<tr>
<td>C</td>
<td>Bud</td>
<td>117</td>
</tr>
<tr>
<td>D</td>
<td>Bud</td>
<td>130</td>
</tr>
<tr>
<td>D</td>
<td>Bud</td>
<td>150</td>
</tr>
<tr>
<td>E</td>
<td>Becks</td>
<td>140</td>
</tr>
<tr>
<td>E</td>
<td>Becks</td>
<td>122</td>
</tr>
<tr>
<td>F</td>
<td>Bud</td>
<td>121</td>
</tr>
<tr>
<td>G</td>
<td>Bud</td>
<td>133</td>
</tr>
<tr>
<td>H</td>
<td>Becks</td>
<td>125</td>
</tr>
<tr>
<td>H</td>
<td>Bud</td>
<td>160</td>
</tr>
<tr>
<td>I</td>
<td>Bud</td>
<td>135</td>
</tr>
</tbody>
</table>

(a) (b) (c) (d)

Figure 2.2: a) The beer distributions table. b) Natural Binning. c) Equal Frequency binning. d) Chimerge Binning.

**Example 2.4.** Notice that the above distribution is highly unbalanced: the first interval contains only one element, interval 4 contains 2 elements, interval 2 has 3 elements and interval 4 has 4 elements.

**Equal Frequency Binning.** This method produces balanced intervals: each of the intervals has the same frequency, i.e., includes the same number of continuous values. The method consists in sorting the values of a continuous attribute, and in assigning the sorted values into \( k \) equally sized intervals, where \( k \) is a parameter supplied by the user. The size \( f \) of each interval is \( f = \lfloor N/k \rfloor \). The \( j \)-th element \( x_j \) in the sort order belongs to interval \( i \) if \( i f \leq j < (i + 1)f \).

**Example 2.5.** Let us consider again the beer distributions relation of fig. 2.2 a). An equal frequency binning discretization into four intervals can be obtained by setting the interval size to 3. We then obtain the intervals \( 1 = [100, 121], 2 = [121, 130], 3 = [130, 1400], 4 = [140, 160] \), as shown in fig. 2.2 b). Notice that, differently from the previous approach, such an approach produces a more balanced distribution.

**Supervised Discretization**

In cases where it is useful to consider each tuple of a relation as “labeled” (e.g., in classification methods: see section 2.4), it is desirable that discretization techniques preserve the labeling of the original values. In this respect, the previous discretization models proposed so far are unsupervised, in that their homogeneity criterion is defined \textit{a priori} and independently from the labels that may be associated to each value. For example, if we consider the beer brand as a label, interval 2 in both natural and equal frequency binning contains both Becks and Bud labels. Supervised
Algorithm ChiMerge($D, A, C, \alpha$);  
\textbf{Input:} A dataset $D$, a continuous attribute $A$, a label attribute $C$, a significance level $\alpha$;  
\textbf{Output:} a set of discretization intervals the partition cod($A$);  
\textbf{Method:} Let $h$ be the number of intervals, $k = |\text{cod}(C)|$, $N = |D|$. Perform the following steps:
1. order the values of cod($A$): each value $a_i \in \text{cod}(A)$ represents an interval $i = [a_i, a_i]$;  
2. repeat
3. \textbf{foreach} interval $i$
4. \quad let $A_{ij}, 1 \leq j \leq k$, the number of cases of the $j$-th label in the $i$-th interval
5. \quad Let $R_j = \sum_{i=1}^k A_{ij}$
6. \quad Let $C_i = A_{ij} + A_{i+1}$
7. \quad Let $E_{ij} = (R_j \times C_i)/N$
8. \quad $\chi^2[i] = \sum_{i=1}^k \sum_{j=1}^k \frac{(A_{ij} - E_{ij})^2}{E_{ij}}$;  
9. \quad $t = \text{argmax}_i \{\chi^2[i] \text{ significant with significance } \alpha \}$;  
10. merge interval $t = [a_i, a_i]$ with interval $t+1 = [a_k, a_k]$, obtaining $[a_i, a_k]$;  
11. until no further merge is possible;

Figure 2.3: The ChiMerge discretization algorithm.

methods aim at produce a posteriori homogeneous discretization intervals. For example, they can produce a discretization in which each interval is associated to a unique label. Notice that the number of homogeneous intervals in a dataset cannot be fixed like in unsupervised methods: rather, it depends from the distribution of the labels, and hence it is an output of the discretization method.

ChiMerge [98] is a supervised, bottom-up discretization procedure, that uses the $\chi^2$ test as a statistical measure of the similarity of two adjacent intervals. The algorithm is shown in fig. 2.3. Initially, each attribute value represents an interval. At each iteration, $\chi^2$ tests are computed for every pair of adjacent intervals, and intervals with the least $\chi^2$ value (at a given level of significance) are merged together. The algorithm terminates when the $\chi^2$ test (with $k-1$ degrees of freedom, where $k$ is the number of distinct labels) fails for each pair of intervals.

Example 2.6. Let us consider again the beer_distributions relation of fig. 2.2 a). We apply the algorithm of fig. 2.3, with 0.9 significance level, in order to obtain a suitable discretization of price that produces homogeneous intervals w.r.t. the beer attribute. Initially, The values are distributed as shown in fig. 2.4 a). In the first iteration, intervals $[150, 150]$ and $[160, 160]$ are significantly similar, and hence merged. In the same way, intervals $[133, 133]$ and $[135, 135]$, $[130, 130]$ and $[133, 135]$, $[122, 122]$ and $[125, 125]$, and $[100, 100]$ and $[117, 117]$ are merged. As a result, we obtain the intervals of fig. 2.4 b). Further discretization steps produce the final intervals shown in fig. 2.4 c) and, as a result, a transformation of the attribute
2.2. DATA PREPROCESSING

<table>
<thead>
<tr>
<th>interval</th>
<th>Bud</th>
<th>Becks</th>
<th>$\chi^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>[100,100]</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[117,117]</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>[120,120]</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>[121,121]</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>[122,122]</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>[125,125]</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>[130,130]</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[133,133]</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[136,136]</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>[140,140]</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>[150,150]</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[160,160]</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(a) initial status

<table>
<thead>
<tr>
<th>interval</th>
<th>Bud</th>
<th>Becks</th>
<th>$\chi^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>[100,120]</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>[120,120]</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>[121,121]</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>[122,122]</td>
<td>0</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>[130,140]</td>
<td>3</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>[140,140]</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>[150,160]</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

(b) iteration 5

<table>
<thead>
<tr>
<th>interval</th>
<th>Bud</th>
<th>Becks</th>
<th>$\chi^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>[100,120]</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>[120,120]</td>
<td>1</td>
<td>3</td>
<td>3.9</td>
</tr>
<tr>
<td>[130,140]</td>
<td>3</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>[140,140]</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>[150,160]</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

(c) final status

Figure 2.4: ChiMerge Execution over the beer_distributions relation.

price as shown in fig. 2.2 d).

The significance threshold allows the tuning of the discretization procedure: a high value may cause over-discretization (i.e., less intervals), while a low value may cause under-discretization (i.e., more intervals). In the example above, the 0.9 significance threshold produces 5 intervals, while a significance threshold of 0.95 produces a single interval.

2.2.2 On-Line Analytical Processing

The above examples show that the relational data model may have the necessary flexibility to allow complex query specifications that formalize typical business analysis instances (i.e., business rules). However, one of the main problems that the relational database research and development community has been involved on is the capability of “getting data in” [100]. This caused the current DBMS to be inappropriate for business analysis, and actually created a dichotomy between On line transaction Processing and Decision Support. The main problem of decision support systems that current DBMS are unable to deal with, in fact, is to provide a technology capable of

1. providing integrated access to multiple, distributed and heterogeneous data, and

2. performing multidimensional analysis of the integrated data.

The first item is referred in literature as Data Warehousing, and has the primary objective to provide a unified and cleaned view of distributed and heterogeneous data [149].
The second item is referred as On-Line Analytical Processing (OLAP, as opposed to On-Line Transaction Processing). Roughly, OLAP allows users to view aggregate data along a set of dimensions and hierarchies, and enables calculations and transformations applied across such dimensions and hierarchies, viewing of subsets of the data, navigating among levels of data. OLAP provides a suitable conceptual model for decision support systems, and a warehouse environment provides the physical layer for such a conceptual model [28].

Conceptual Model

There is a wide research effort to develop a simple conceptual model for online analytical processing, capable of suitable representation of the entities involved, as well as supporting ad-hoc querying capabilities [68, 73, 21, 3, 99, 126, 36]. The popular conceptual model that influences the front-end tools, database design, and the query engines for OLAP is the multidimensional view of data. In a multidimensional data model, there are a number of numeric measures that are the objects of the analysis. Each of the numeric measures depends on a set of dimensions, which provide the context for the measure. For example, the dimensions associated with a sale amount can be the geographic area, product name, and the date when the sale was made (see, e.g., fig. 2.5). The dimensions together are assumed to uniquely determine the measure.

The attributes of a dimension may be related via a hierarchy of relationships. For example, the product name can be related to its category, the date to month, quarter and year, and the geographic area to the country and city that it contains (see, e.g., fig. 2.6 a)).

Another distinctive feature of the conceptual model for OLAP is aggregation of measures over one or more dimensions: e.g., computing the total sales for each country (or for each year). Other popular operations include comparing two measures...
2.2. Data Preprocessing

\[
\begin{array}{c|c|c|c}
\text{Month} & \text{Jan} & \text{Feb} & \\
\text{Year} & 1996 & 1997 & \\
\hline
\text{Sales} & \text{Miami} & \text{Orlando} & \\
\text{City} & \text{Dallas} & \text{Houston} & \\
\hline
\text{Sales} & 12 & 9 & 8 \\
\end{array}
\]

Figure 2.6: a) Hierarchical spreadsheet-based representation. b) Pivoting on Date and City.

(e.g., sales and budget) aggregated on the same dimensions.

We shall briefly discuss some of the popular operations that need to be supported by the multidimensional view of the database. One such operation is Pivoting. Consider the multidimensional schema of fig. 2.5. A pivot of a multidimensional database selects a set of dimensions that are used to aggregate a measure, e.g., sales in the above example. For example, the pivot on Date and City operator eliminates the dimension Product, by aggregating the values according to different values of the other dimensions. An example is represented in fig. 2.6.

Other related operators are rollup and drill-down. Rollup corresponds to taking the current view of the multidimensional database and doing a further aggregation on one of the dimensions. For example, by considering the example of fig. 2.6 a), we can apply a rollup over the Time dimension, at the Month level, producing a new view in which different values of Month with the same value of Year are aggregated. The drill-down operation is the opposite of rollup. slice and dice correspond to reducing the dimensionality of the data, i.e., taking a projection of the data on a subset of dimensions for selected values of the other dimensions.

### Warehouse Server

The data model previously described can be implemented in two main ways. The multidimensional OLAP servers (MOLAP) store directly the multidimensional views by means of a multidimensional storage engine. This make it possible to implement front-end multidimensional queries on the storage layer through direct mapping. Such an approach has the advantage of excellent indexing properties, but provides poor storage utilization, especially when the data set is sparse.

A different approach can consist of relying on relational DBMSs, (Relational OLAP). Usually, data is stored as a star schema, (or snowflake schema, to explicitly support attribute hierarchies) to represent the multidimensional data model. The
database consists of a single fact table and a single table for each dimension. Each
tuple in the fact table consists of a pointer to each of the dimensions that provide
multidimensional coordinates, and stores numeric measures for that coordinates. As
an example, the fact table sales(Store, Date, Product, Sales) and the dimension
tables store(Store, City, Region), product(Product, Category) physically repre-
sent the multidimensional view in fig. 2.6 a).

The spreadsheets related to the multidimensional representation can be popu-
lated by means of a series of SQL queries on the tables of a schema and the main aim
of the OLAP engine is to support efficient extraction from the relational database.
Many extensions to SQL have been proposed. For example, the CUBE operator de-
scribed, e.g., in [72]. One could write a query such as:

```sql
SELECT Store, Date, Product, SUM(Sales)
FROM sales
CUBE BY Store, Date, Product
```

which has as meaning the union of SQL queries, obtained as follows: for each subset
S of the CUBE BY attributes, the CUBE BY clause is replaced by group by S, and
each attribute in the SELECT clause that is not in S is replaced by the ALL special
constant. For example, for the above query, the translation is:

```sql
SELECT Store, Date, Product, SUM(Sales)
FROM Sales
GROUP BY Store, Date, Product
UNION
SELECT Store, Date, 'ALL', SUM(Sales)
FROM Sales
GROUP BY Store, Date
UNION
SELECT Store, 'All', 'ALL', SUM(Sales)
FROM Sales
GROUP BY Store
UNION
SELECT Store, 'ALL', Product, SUM(Sales)
FROM Sales
GROUP BY Store, Product
UNION
SELECT 'ALL', Date, Product, SUM(Sales)
FROM Sales
GROUP BY Date, Product
UNION
SELECT 'ALL', Date, 'ALL', SUM(Sales)
FROM Sales
GROUP BY Date
```
2.3. Data Mining

UNION
SELECT 'ALL', 'ALL', Product, SUM(Sales)
FROM Sales
GROUP BY Product

The slice and pivot operations can then be implemented by means of simple relational operations on the resulting table.

In order to implement the rollup and drill-down operations, we can use the group-by clauses as well, and extend the relational language. For example, the following statement represents a combination of the rollup and cube operations:

SELECT Region, City, Store, Date, Product, SUM(Sales)
FROM sales, store
CUBE BY Date, Product
ROLLUP BY Region, City, Store

In order to answer queries efficiently, data warehouses use redundant structures such as indices and materialized views [94, 142]. Choosing which indices to build and which views to materialize is an important physical design problem [39].

The above shown transformations show that, in principle, the techniques used for computing aggregates and group-by operations can apply to compute the core of the cube as well. In practice, further optimizations are possible by exploiting ad-hoc properties of the underlying conceptual model. For example, since a cube is the union of many group-by's, the naive algorithm that computes their union can be improved when data is aggregated with a \textit{distributive aggregate} [72], by exploiting suitable data organization combined with ad-hoc algorithms [2, 141, 45, 81].

2.3 Knowledge Data Discovery and Data Mining

The term data mining is used in this context to describe a set of algorithms and methods capable of extracting patterns from a structured data source. We search in the data source for \textit{regularities} - combinations of values for certain attributes shared by facts in the database. In a sense, this regularity is a high-level summary of information in the data under consideration.

Data Mining draws many of its basic principles from mature concepts in databases, machine learning and statistics, none of which is capable by itself to offer a solution to the challenging problems that arise, such as mining different types of data, efficiency and scalability, mining at multiple abstraction levels. Hence, different classification schemes can be used to categorize data mining methods, based on the kinds of databases to be studied, the kinds of knowledge to be discovered, and the kinds of techniques to be used [32].

According to [22], we classify data mining techniques in two general categories, that:
• predictive modeling (or directed knowledge discovery), when samples of past experience with known answers are examined and generalized to future cases.

• undirected knowledge discovery, when the task is to identify patterns in the data that may be significant.

Prediction problems are described in terms of specific goals, which are related to past records with known answers. These are used to project new cases. Undirected knowledge discovery problems usually describe a stage prior to prediction, where information is insufficient for predictive analysis and has to be integrated with further knowledge.

2.4 Predictive Modeling

The main goal of predictive modeling is to predict some fields in a database based on other fields. If the field being predicted is a numeric (continuous) variable, then the prediction problem is a regression problem. If the field is categorical then it is a classification problem. There is a wide variety of techniques for classification and regression. The problem in general is cast as determining the most likely value of the variable being predicted given the other fields, the training data (in which the target variable is given for each observation), and a set of assumptions representing one’s prior knowledge of the problem.

A classification task can be described as a probability estimation problem. The aim is to estimate the probability that the class $C$ takes some value $c$, provided that the other fields $X$ take value $x$ for some feature vector $x$. One could derive this probability from the joint density on $C$ and $X$ resulting from the training set. However, this joint density is rarely known and very difficult to evaluate. Hence, there’s the need to resort to various techniques for estimating. The techniques include:

1. Density estimation, such as Bayesian methods [42, 111].

2. Metric space based methods in which we define a distance measure on data points and guess the class value based on proximity to data points in the training set (for example, the k-nearest-neighbor method [111, 13]).

3. Projection into decision regions: divide the attribute space into decision regions and associate a prediction with each. Examples are decision trees (that make a piecewise constant approximation of the decision surface [111, 13, 148, 54]), and neural networks (that find non-linear decision surfaces).

In the following we shall consider two methods that received a particular attention in the database community: decision-trees and Bayesian classification.
2.4. PREDICTIVE MODELING

Algorithm BuildTree(n, D, CL);
Input: an empty node n, a data partition D, a splitting criterion CL;
Output: a decision tree rooted at n and trained according to the data partition D;
Method: Perform the following steps:
1. A = CLfind_best_classifier_attribute(p)
2. k = CLsplit_dataset(D, A)
3. if k > 0
4. for each partition Di, 1 ≤ i ≤ k
5. create a new children ni of n
6. BuildTree(ni, Di, CL)
7. endfor
8. endif

Figure 2.7: General Decision Tree training schema.

2.4.1 Decision-Tree Classification

Decision trees are widely studied among the classification methods for their simplicity, understandability and capability to deal with high-dimensional data.

A decision tree is a tree data structure consisting of decision nodes and leaves, where each decision node denotes a test on an attribute (and each branch represents the outcome of the test), and leaf nodes represent classes or class distributions. Unknown samples are classified by testing their attribute instances against the decision tree, and assigning the instance to the class of the leaf node that is reached as a result of testing.

The construction of a decision tree consists of two main phases.

- In the learning phase, the tree is built using training data. A general schema for decision-tree learning algorithms is given in fig. 2.7. A tree is built starting from a root node, and recursively growing a leaf node in subtrees, according to some splitting (i.e., data partitioning) technique. Many decision-tree learning algorithms differ simply in the definition of the splitting criterion CL (i.e., in the methods find_best_classifier_attribute and split_dataset as shown in fig. 2.7) [48]. Let freq(ci, D) denote the frequency of class ci (1 ≤ i ≤ n) in a set D of cases. Let D1, ..., Dk be the subsets of D consisting of cases with distinct known value for a given attribute A.

  - The C4.5 induction algorithm [122] chooses the attribute with the highest information gain, i.e., the attribute that requires the minimum amount of information to classify a given sample. The information gain of an
attribute $A$ for a set of cases $D$ is given by the formula

$$ gain(A) = info(D) - \sum_{i=1}^{k} \frac{|D_i|}{|D|} \times info(D_i) $$

where

$$ info(T) = -\sum_{j=1}^{n} \frac{freq(c_j, T)}{|T|} \times \log_2 \frac{freq(c_j, T)}{|T|} $$

is the entropy function, i.e., a function computing the expected information needed to classify a given sample.

- The CART induction algorithm [23] chooses the attribute with the minimum impurity. The homogeneity of an attribute $A$ for a set of cases $D$ is computed as the impurity reduction of a splitting, given by

$$ gini(D, A) = -\sum_{i=1}^{k} \frac{|D_i|}{|D|} gini(D_i) $$

where

$$ gini(T) = 1 - \sum_{j=1}^{n} \left( \frac{freq(c_j, T)}{|T|} \right)^2 $$

is the index of homogeneity of the set $T$. Attributes with low gini index are homogeneous, in the sense that they show little impurity.

- The CHAID induction algorithm [47] chooses the attribute with the best statistical significance, given by the $\chi^2$ statistics:

$$ \chi^2 = \sum_{i=1}^{k} \sum_{j=1}^{n} \frac{(freq(c_j, D_i) - expected(D_i, c_j, D))^2}{expected(D_i, c_j, D)} $$

where

$$ expected(D_i, c_j, D) = \frac{freq(c_j, D) \times |D_i|}{|D|} $$

is the expected frequency of class $c_j$ in partition $D_i$. Attributes with low $\chi^2$ probability are highly correlated with the target attribute.

- In the cleaning phase, the tree structure is revised to improve prediction accuracy. The most widely used methods are tree-pruning methods, that allow to address the problem of overfitting the data. Such methods use statistical measures to remove the less reliable branches, so that the resulting trees have improved classification ability both in speed and in accuracy.

There are two common approaches to tree pruning:
2.4. PREDICTIVE MODELING

1. early stopping methods: a tree is pruned by stopping its construction. For example, the split of a node can be prevented if the predictor attribute under consideration is below a level of statistical significance (e.g. if the $\chi^2$ test is under a user-defined level of confidence in the CHAID algorithm), and the current node becomes a leaf.

2. postpruning methods: after the full tree has been grown, branches are removed by replacing a split (a subtree) with a leaf if some conditions hold. For example, a subtree can be replaced by a leaf if the validation error of the pruned tree is not worse than the validation error of the unpruned tree.

2.4.2 Bayesian Classification

Bayesian classification is among the most practical approaches to many types of learning problems [110, 111, 40]. Bayesian classifiers are statistical classifiers, used to predict class membership probabilities of instances. The classification algorithm is based on Bayes theorem: for a given data sample $X = \{A_1, \ldots, A_m\}$, the posterior probability that $X$ belongs to class $C$ is given by $\Pr(C|X)$, and reflects our confidence about $C$ given $X$. Bayes theorem relates such a probability with the prior probability $\Pr(C)$ of class membership and the posterior probability $\Pr(X|C)$ of $X$ given $C$:

$$\Pr(C|X) = \frac{\Pr(X|C)\Pr(C)}{\Pr(X)}$$

In order to train the above probability on a dataset $D$, we can rewrite the above formulas as

$$\Pr(C|X, D) = \frac{\Pr(X|C, D)\Pr(C|D)}{\Pr(X|D)}$$

for an instance $X$ with unknown class value, a Bayesian classifier classifies the tuple in class $c_i$ if

$$\Pr(c_i|X, D) > \Pr(c_j|X, D) \text{ for } 1 \leq j \leq n, j \neq i$$

The estimation $\Pr(X|D)$ is constant for all classes, so that the above condition is rewritten as

$$\Pr(X|c_i, D)\Pr(c_i, D) > \Pr(X|c_j, D)\Pr(c_j|D) \text{ for } 1 \leq j \leq n, j \neq i$$

In the simplest hypothesis (Naive Bayesian Learning), the attributes of the dataset are pairwise independent, so that the probability $\Pr(X|c_i)$ can be rewritten as

$$\Pr(X|c_i, D) = \prod_{i=1}^{m} \Pr(A_i|c_i, D)$$

Now, we can estimate $\Pr(c_j|D) = freq(c_j, D)/|D|$, and the parameters $\Pr(A_i|c_i, D)$ as follows:
• if $A_i$ is categorical, then $\Pr(A_i|c_j, D)$ is “maximum-likelihood” estimated as

$$\frac{freq(c_j, D_i)}{freq(c_j, D)}$$

where $D_i$ is the fraction of $D$ such that $A_i = a_i$.

• if $A_i$ is continuous, then the attribute is assumed to have a normal distribution with mean $\mu_{c_j}$ and variance $\sigma_{c_j}$ to be estimated on $D$.

Empirical studies have shown that, although the independence hypothesis is too strong in practice, the Naive Bayesian classifier is comparable in classification accuracy to other classification methods, and also exhibits high performance speed when applied to large databases.

2.5 Undirected Knowledge Discovery

Undirected knowledge discovery is different from predictive modeling - there is no target field. The data mining tool is simply let loose on the data, in the hope that it will discover some meaningful structure. In this context, the process of interpretation and refinement of the results plays a crucial role, in that understandability and usefulness of the results have to be defined a posteriori.

The most common techniques studied in literature are data summarization and data segmentation.

2.5.1 Data Summarization

The goal is to extract compact patterns that describe subsets of the data. There are two classes of methods which represent taking horizontal (cases) or vertical (fields) slices of the data. In the former, one would like to produce summaries of subsets: e.g., producing sufficient statistics, or logical conditions, that hold for subsets. In the latter case, one would like to predict relations between fields.

Association rules discovery is the task that has attracted most attention from the database community [143, 131, 5, 4]. Associations are rules that state that certain combinations of values occur with other combinations of values with a certain frequency and certainty. A general definition is the following.

Let $\mathcal{I} = \{a_1, \ldots, a_n\}$ be a set of literals, called items. An itemset $T$ is a set of items such that $T \subseteq \mathcal{I}$. Given a relation $R = A_1 \ldots A_n$, a transaction of an instance $r$ of $R$ associated to attribute $A_i$ (where $\text{dom}(A_i) = \mathcal{I}$) w.r.t. a transaction identifier $A_j$ is a set of items of the tuples of $r$ having the same value of $A_j$. Formally, for each $v \in \text{cod}(A_j)$, a transaction associated to $v$ is an itemset $T_v \in \mathcal{I}$ such that

• for each $a \in T_v$ there exist $\mu \in r$ such that $\mu[A_j] = v$ and $\mu[A_i] = a$;
for each \( \mu \in r \), if \( \mu[A_j] = v \) then \( \mu[A_i] \in T_v \).

The set of all transactions of \( r \) associated to \( A_j \) over the transaction identifier \( A_j \) is represented as \( B_{r,i,j} \).

An association rule is a statement of the form \( X \Rightarrow Y \), where \( X \subseteq \mathcal{I} \) and \( Y \subseteq \mathcal{I} \) are two sets of items. To an association rule we can associate some statistical parameters. The support of a rule is the percentage of transactions that contain the set \( X \cup Y \), and the confidence is the percentage of transactions that contain \( Y \), provided that they contain \( X \). More precisely, we define the support of an itemset \( I \) as \( \text{supp}(I) = \# \{ T \in B_{r,i,j} | I \subseteq T \} \). An association rule is then an expression:

\[
A \Rightarrow B[S, C]
\]

where

- \( \emptyset \neq A, B \subseteq \mathcal{I} \)
- \( A \cap B = \emptyset \)
- \( S = \text{supp}(A \cup B) \), is the support of the rule
- \( C = \text{supp}(A \cup B) / \text{supp}(A) \), is the confidence of the rule

The problem of association rules mining can be finally stated as follows: given an instance \( r \) of a relation \( R \), find all the association rules from the set of transactions \( B_{r,i,j} \), such that for each rule \( A \Rightarrow B[S, C] \), \( S \geq \sigma \) and \( C \geq \gamma \), where \( \sigma \) is the support threshold and \( \gamma \) is the confidence threshold.

Association rules find their most natural application in market basket analysis, where each transaction corresponds to a purchase, and an association rule models items that are usually sold together. This, for the marketing analyst seems to be particularly interesting and useful as a tool for the more general task of verifying how a seller’s collection of products meets its customers’ preferences.

**Custom Algorithms**

The computation of association rules can be decomposed in two phases:

1. generation of frequent itemsets, i.e., of collections of items that usually appear together in transactions.

2. generation of rules from frequent itemsets. Rules can be generated from frequent itemsets by observing that a rule \( A \Rightarrow B \) has the same support of \( A \cup B \). Hence, all the valid rules are of the form \( R \Rightarrow (I \setminus R) \) such that \( I \) is frequent and \( \emptyset \subset R \subset I \), and for each frequent itemset \( I \) we need to select the righthand sets \( R \) such that the above rule has a sufficient confidence. It is easy to see that the confidence of such a rule is a monotonically decreasing function on itemsets \( R \), so that the generation of rules can be performed level-wise.
The first phase is the most computationally intensive phase, and has attracted most attention from the current literature.

The simplest way to compute frequent itemsets of a given size is that of using of iceberg queries [45]. Given a relation \( R \) with attributes \( a_1, a_2, \ldots, a_n \) and \( b \) and an aggregation function \( \text{agg}_f \), an iceberg query is a query of the form

\[
\text{SELECT } R.a_1, R.a_2, \ldots, R.a_n, \text{agg}_f(R_i.b) \\
\text{FROM } R \\
\text{GROUP BY } R.a_1, R.a_2, \ldots, R.a_n \\
\text{HAVING } \text{agg}_f(R_i.b) \geq \text{threshold}
\]

In the case of frequent itemsets, \( \text{agg}_f = \text{COUNT} \), and \( \text{threshold} \) represents the minimum value \( \sigma \) of the support. An example of simple query that computes itemsets of size 2 over the \textit{sales} table shown in section 2.2.2 is the following:

\[
\text{SELECT } R\	ext{.Product}, S\	ext{.Product}, \text{COUNT}(*) \\
\text{FROM } \text{sales} R \text{ INNER JOIN sales S} \\
\text{ON } R\	ext{.Store} = S\	ext{.Store} \\
\text{GROUP BY } R\	ext{.Product}, S\	ext{.Product} \\
\text{HAVING } \text{COUNT}(*) \geq \text{thres}
\]

From an efficiency viewpoint, the approach based on iceberg queries is rather inefficient. If \( |I| = n \), with \( n \) large, then the number of itemsets with exactly \( k < n \) elements is \( O(n^k) \). Things are even worst when trying to compute rules of arbitrary size: the number of possible itemsets is \( 2^n \), a number clearly intractable even for low values of \( n \).

One can think to exploit the methods described in [45] to implement frequent-itemset iceberg queries efficiently. However, [5] suggests a smarter technique that exploits the following property. Given an itemset \( I \),

\[
\text{if } \text{supp}(I) > \sigma \text{ then } \text{supp}(J) > \sigma \text{ for each } J \subseteq I
\] (2.1)

Practically, each frequent itemset contains only frequent subsets and, by the converse, the discovery of an unfrequent itemset automatically excludes all the supersets of such itemset. This suggests a way for computing frequent itemsets by using a breadth-first search over the lattice of itemset with union. The \textit{Apriori} algorithm, proposed in [5], implements such a technique. Figure 2.8 shows the general schema of the algorithm.

Initially, the algorithm computes the candidate itemsets of size 1. The core of the algorithm is then a cycle, where the \( k \)-th iteration examines the set \( C_k \) of candidate itemsets of size \( k \). During such an iteration the occurrences of each candidate itemset are computed by scanning the data. Unfrequent itemset are then dropped, and frequent itemset are maintained in \( L_k \). By exploiting property 2.1,
Algorithm Apriori($\mathcal{B}$, $\sigma$);

**Input**: a set of transactions $\mathcal{B}$, a support threshold $\sigma$;

**Output**: a set $\text{Result}$ of frequent itemsets

**Method**: let initially $\text{Result} = \emptyset$, $k = 1$, $C_1 = \{ \{a\, | \, a \in \mathcal{I}\} \}$. Perform the following cycle:

1. while $C_k \neq \emptyset$ do
2.  \hspace{1em} \textbf{foreach} itemset $c \in C_k$ do
3.  \hspace{2em} \text{supp}(c) = 0;
4.  \hspace{1em} \textbf{foreach} $b \in \mathcal{B} \text{ such that } c \subseteq b$ do $\text{supp}(c) \leftarrow \text{supp}(c) + 1$;
5.  \hspace{1em} \textbf{foreach} $c \in C_k \text{ such that } c \subseteq b$ do $\text{supp}(c) \leftarrow \text{supp}(c) + 1$;
6.  \hspace{1em} $L_k := \{ c \in C_k \text{ such that } \text{supp}(c) > \sigma \}$;
7.  \hspace{1em} $\text{Result} := \text{Result} \cup L_k$;
8.  \hspace{1em} $C_{k+1} := \{ e_i \cup e_j \mid e_i, e_j \in L_k \land |e_i \cup e_j| = k + 1 \land \forall e \subseteq e_i \cup e_j \text{ such that } |e| = k \land e \in L_k \}$;
9.  \hspace{1em} $k := k + 1$;
10. end while

Figure 2.8: Apriori Algorithm for computing frequent itemsets.

candidate itemsets of size $k + 1$ can be built from pairs of frequent itemset of size $k$ differing only in one position (as specified by step 9 of the algorithm). Finally, $\text{Result}$ shall contain $\bigcup_k L_k$.

The algorithm minimizes the number of candidate itemsets to be generated. Each unfrequent candidate itemset automatically drops all candidate itemsets that are its supersets. The number of scans over $\mathcal{B}$ equals the maximum size of frequent itemsets considered.

Many variations to the Apriori algorithm have been proposed:

- A simple variation of the algorithm considers the generation of candidate itemsets of order $(i + 1), \ldots, (i + k)$ (i.e. $C_{i+1}, \ldots, C_{i+k}$ from $L_i$) in a single step, in order to reduce of a factor $k$ the number of scans over the database. However, the real bottleneck in Apriori is the generation of frequent $k$-itemsets when $k$ is small. Usually, the candidate generation phase produces too many small-size candidates, so that counting and pruning becomes quite expensive in the first iterations of the algorithm. The DHP algorithm [120] introduces some improvements to the Apriori schema, to solve such a problem. When $k$ is small, $k$-itemsets can be easily generated from the set of transactions. Hence, by exploiting hash tables, their support can be efficiently estimated during the counting phase of $k - 1$ itemsets. A further pruning can occur during the $k$-candidate generation, looking only at the $k$-itemsets in the hash table that were predicted with high confidence.

- Apriori neatly divides each step of the computation in two phases: counting
of the candidate $i$-itemsets (which requires a sequential scan of the database), and then generation of the candidate $(i + 1)$-itemsets. However, generation of candidate itemsets can be interleaved to counting as soon as a candidate itemset is recognized as frequent. The DIC (Dynamic Itemsets Counting) algorithm [24] extends the Apriori algorithm by generating new candidates as soon as they are recognized as frequent, and starting the counting phase for such new candidates from the position in the database reached in the last step. This strategy reduces the number of scans of the database.

- A further extension of the apriori algorithm can fix the number of scans of the database to 2. The Partition algorithm [131] proposes two phases. In the first phase the algorithm partitions the database in small segments that fit into memory. Frequent itemsets are computed for each partition. Since an itemset that is frequent in the whole database is frequent in at least one partition of the database, the only candidates that need to be checked are given by the union of the frequent itemsets discovered in each partition. Hence, a second phase rescans the database once in order to compute the global support of all such itemsets.

- A straightforward way to reduce the complexity of the algorithm is to exploit sampling techniques [143] in order to mine only a subset of the given data. The basic idea is to pick a random sample $S$ of the given data $D$, and then search frequent itemsets in $S$ instead of $D$. The method is a trade-off between some degree of accuracy and efficiency. The sample size $S$ is chosen to fit into memory, so that only one scan over $D$ is required. The rest of $D$ is used to compute the actual support of each frequent itemset found.

**Extensions**

The basic framework for association rules has been extended towards several directions, aimed at

- improving the expressive power of the mined rules;
- improving the quality of the mined rules;
- exploiting the results of associations mining to mine different kinds of knowledge (e.g., classification).

**Multidimensional Rules.** The most straightforward extension to association rules is the problem of discovering multidimensional rules. A unidimensional association rule $a \Rightarrow b$ can be represented as a logical expressions of the form

$$P(X, Y) \land Q(Y, a) \Rightarrow P(X, Z) \land Q(Z, b)$$
with the meaning that, in table $P$, there exist a sufficient number of transactions (identified by the transaction identifier $X$) in which $a$ and $b$, of the same type, appear together. However, by relaxing the requirement that $a$ and $b$ must have the same type, we can obtain multidimensional rules, of the form $(A_i = a) \rightarrow (A_j = b)$ (i.e., rules involving itemsets with items of different type or, equivalently, items from different columns), logically represented as:

$$P(X) \land (Q(X, a) \Rightarrow R(X, b))$$

**Example 2.7.** Let us consider the relation $R = ABC$, with instance $r$, shown in fig. 2.9 (a). By mining unidimensional rules with support 60\% and confidence 75\% over the attribute $B$ (using $A$ as a transaction identifier) we find the rule $(B = b_1) \Rightarrow (B = b_2)$; by mining multidimensional rules with support 60\% and confidence 75\% over the attributes $B$ and $C$ we find the rule $(C = c_1) \Rightarrow (B = b_1)$. <

Even if efficient methods to compute multidimensional rules can be defined [95], it is easy to see that the two specifications are interchangeable, so that the computation of either unidimensional or multidimensional rules is simply a matter of preprocessing before applying the algorithm of fig. 2.8. Given an instance $r$ of a transactional relation $R = AB$, (where $A$ is the transaction identifier), we can define a new relation $S = AB_1 \ldots B_n$, where $B_i$ is a column corresponding to the $i$-th distinct value $b_i \in \text{cod}(B)$. For each distinct value $a \in \pi_A(r)$, there exists a unique
tuple \( \mu \) in the instance \( s \) of \( S \); we associate \( \mu[B_i] = 1 \) if \( \langle a, b_i \rangle \in r \), \( \mu[B_i] = 0 \) otherwise. Practically speaking, such a transformation corresponds to build a bitmap index for each transaction in the table, and to replace the original table with the new collection of the bitmap indexes. Of course, such a transformation is quite expensive when \( \text{cod}(B) \) is huge and the original table is not dense (i.e., it has little values in each itemset).

**Example 2.8.** Let us consider the restriction to the attributes \( AB \) of the relation and instance defined in example 2.7, we can define \( S = AB_1B_2 \), and build the instance \( s \) as shown in fig. 2.9 (b).

The opposite transformation is straightforward as well. Let us consider a table \( R = A_1\ldots A_n \). For each \( a_i \in \text{cod}(A_i) \), we can define a unique identifier \( \langle A_i, a_i \rangle \), so that each row \( \mu = \langle a_1, \ldots a_n \rangle \) can be represented by means of an itemset \( \{\langle A_1, a_1 \rangle, \ldots \langle A_n, a_n \rangle\} \). We can then define a view \( S = TT \) of \( R \), and an instance \( s \) of \( S \) corresponding to \( r \). For each itemset \( \{\langle A_1, a_1 \rangle, \ldots \langle A_n, a_n \rangle\} \), we define \( n \) tuples \( \mu_i (1 \leq i \leq n) \) such that \( \mu_i[T] \) contains a unique identifier for the itemset, and \( \mu_i[I] \) contains the code \( \langle A_i, a_i \rangle \).

**Example 2.9.** Again, let us consider the restriction to the attributes \( BC \) of the relation and instance defined in example 2.7. We can define the coding \( \langle B, b_i \rangle = a_i \), \( \langle B, b_2 \rangle = b_i \), \( \langle C, c_1 \rangle = c \) and \( \langle C, c_3 \rangle = d \), so that \( s \) is defined as shown in fig. 2.9 (c).

Notice that the above approach produces a new dataset with all itemsets of the same length. A major drawback of this is that high-dimensional datasets (i.e., tables with many attributes) may result in dense datasets. The apriori algorithm exhibits poor performance when executed against such datasets. In fact, in order to check the frequency of an itemset \( I \) of size \( l \) the apriori algorithm needs to check all its subsets (i.e., it check \( 2^l \) itemsets). When \( l \) is large (as it happens to be in dense datasets) such a computation is too expensive. To solve this problem, in [11, 12] a new algorithm for mining multidimensional association rules is presented. The working principle of the algorithm is conceptually different from that of the apriori algorithm: here, the algorithm checks maximal-frequent itemsets (i.e., itemset with no frequent supersets), and candidate itemsets are generated only from unfrequent supersets.

**Sequential patterns.** When it is possible to define a sequential order among data (e.g., when transactions can be organized along a time dimension), an interesting variation of the problem of frequent itemsets discovery can be defined: frequent sequences discovery, where a sequence is a collection of ordered itemsets.

The approach we describe here is based on the work by Agrawal and Srikant [137]. A sequence \( (s_1, s_2, \ldots, s_n) \) is an ordered list of itemsets, where each itemset \( s_i \) represents events occurring simultaneously. The whole dataset can hence be organized as a set
2.5. UNDIRECTED KNOWLEDGE DISCOVERY

Algorithm GSP(\(B, \sigma\));

**Input:** a set of sequences \(B\), a support threshold \(\sigma\);

**Output:** a set Result of frequent sequences

**Method:** let initially Result = \(\emptyset\), \(k = 1\), \(C_1 = \{(i)|i \in I\}\). Perform the following cycle:

1. while \(C_k \neq \emptyset\) do
2.   \hspace{1em} \textbf{foreach} itemset \(e \in C_k\) do
3.     \hspace{2em} \text{supp}(e) = 0;
4.   \hspace{1em} \textbf{foreach} \(b \in B\) do
5.     \hspace{2em} \textbf{foreach} \(e \in C_k\) such that \(e \subseteq b\) do \(\text{supp}(e) = +\);
6.   \hspace{1em} \text{L}_k := \{e \in C_k | \text{supp}(e) > \sigma\};
7.   \hspace{1em} \text{Result} := \text{Result} \cup \text{L}_k;
8.   \hspace{1em} \text{C}_{k+1} := \{e \cup e_j | e_i \in \text{L}_k \land |e_i \cup e_j| = k + 1 \land \forall e \subseteq e_i \cup e_j\ such that |e| = k - 1: e \in \text{L}_{k-1}\};
9.   \hspace{1em} k := k + 1;
10. end while

Figure 2.10: Algorithm for computing frequent patterns. The candidate generation phase generates only contiguous subsequences [137].

of sequences. A classical example is market basket analysis; here, events are items purchases, and a sequence represents the set of itemsets bought by a customer.

A sequence \(s = \langle a_1 \ldots a_n \rangle\) is supported by a sequence \(t = \langle b_1 \ldots b_m \rangle\) (\(s \subseteq t\)) if for each \(a_i \in s\) there exists \(b_j \in t\) such that \(a_i \subseteq b_j\). The support of a sequence in a dataset as the fraction of sequences that support it.

Again, notice that if a sequence \(s_1\) is supported by a sequence \(s_2\), then each of its sub-sequences is supported as well: small sequences are subsumed by larger ones which contain them. An algorithm for computing sequential patterns can hence be defined, which has many similarities with the Apriori algorithm. The algorithm proceeds level-wise (see fig. 2.10: at step \(i\), the \(i\)-sequences are sequences with exactly \(i\) itemsets. The candidate \((i + 1)\)-sequences are computed from the frequent \(i\)-sequences found in the last step (this extension step is performed in the same way of the Apriori algorithm), and then such candidates are checked upon the (translated) against the database, deleting the non frequent ones.

**Hierarchies.** Hierarchies provide a simple and powerful way of dealing with domain knowledge in data mining. A concept hierarchy defines a sequence of mappings from a set of low level concepts to higher level, more general concepts. Concerning association rules mining, we can substantially devise two kinds of hierarchies:
• Transaction hierarchies, that define multi-level abstractions of the transaction identifiers, and consequently partition the transactions of a relation in categories and sub-categories. Given a set $\mathcal{B}$ of transactions, a transaction identifier abstraction $\tau$ defines a subset $\mathcal{B}_\tau$ of $\mathcal{B}$. For a given abstraction level specified by $\tau_1, \ldots, \tau_n$, $\mathcal{B}_{\tau_1} \cup \ldots \mathcal{B}_{\tau_n} = \mathcal{B}$. For example, a temporal abstraction splits a relation instance in smaller (possibly disjoint) instances, corresponding to different time periods.

• Items hierarchies, that provide multi-level abstractions of the items involved, and provide different views of the same transactions in the originary relation. Given a set $\mathcal{B}$ of transactions, an item abstraction $\tau$ defines a set $\mathcal{B}_\tau$ such that each transaction in $\mathcal{B}$ has a correspondent transaction in $\mathcal{B}_\tau$. For example, a transaction composed by the items pasta, wine and shoes corresponds, at a higher abstraction level, to the transaction composed by the items food and wear.

The most straightforward transaction hierarchy is the time hierarchy. Time hierarchies can be specified, at different abstraction levels, by means of complex time algebras [125], and association rules mining can be related to such hierarchies in many different ways.

• Given a time abstraction level, we can compute the behavior of the quality measures of associations within such an abstraction. For example, we can monitor the weekly support of a given set of association rule, to discover that rules involving sport items are more frequent during week-ends. Sample evolution tasks are

  – cyclic rules [119], i.e., rules holding within cycles of time. The main objective is to find all association rules $A \Rightarrow B$ which hold only in subsets $\mathcal{B}_o, \mathcal{B}_{o+l}, \mathcal{B}_{o+2l}, \ldots$ of $\mathcal{B}$, for a given cycle length $l$ ($0 < l < n$) and some offset $o$ ($0 \leq o < l$).

  – calendric rules [125], i.e., rules holding according to a given calendar algebra. The main objective is to find all association rules $A \Rightarrow B$ which hold in all segments $\mathcal{B}_{t_i}$ such that the time chronon $t_i$ belongs to some calendar algebra.

• Within different abstraction levels, we can relate the validity of a given association rule at an abstraction level with its validity at a different abstraction level. For example, we can compare rules holding weekly with rules holding daily, and identify interesting exceptions.

The adoption of item hierarchies in association rules mining has been extensively studied. A typical example in which item hierarchies are very useful is given by application domains where the number of items is very large. In such cases, if
transactions are sparse, it is difficult to find strong associations among flat data items. However, generalizing the items to more general abstraction can help in finding stronger association rules. For example, a simple hierarchy among web references is given by web pages and web sites: a site can contain many web pages, and a web page is uniquely identified by a url. We may not be able to find any relevant association rule at the url abstraction level but we may find associations, e.g., among sites concerning stock trades and sites concerning financial news. There are two main ways in which association rules can interact with item hierarchies:

- **Roll-up/Drill-down** [74, 77]. Rules are computed over datasets that contain items at the same abstraction level. For example, we can compute rules relating web sites, or rules relating web pages (which describe associations at a lower abstraction level).

- **Drill-Through** [138, 137]. Rules are computed over datasets that contain items at different abstraction levels. For example, we may find a rule stating that, if a user visits a given page A in a given site B, then he is likely to visit any page in a given site C.

**Quality Measures.** The most popular quality measures for evaluating the strength and relevance of association rules, support and confidence, can be of little help in evaluating the discovered rules from different perspectives. For example, support alone may not serve as a reliable interestingness measure, and confidence can be misleading in many practical situations where items are negatively correlated [135].

There exist various proposals to extend the quality measures of an association rule. We mention the following.

- A first straightforward extension is aimed at investigating the predictive power of itemsets $A$ and $B$ in a rule $A \Rightarrow B|S,C$. More precisely, the purpose is to correlate $S$ to $\Pr(A, B)$ and $C$ to $\Pr(B|A)$. Let us consider a sample $N$ of transactions, where each transaction is represented as a random boolean vector $X = \langle X_1, \ldots, X_n \rangle$. Each random variable $X_i$ is a boolean variable, such that $X_i = 1$ if and only if the item $a_i \in I$ belongs to transaction $X$. For a given itemset $S$, suppose that the (real) probability that $S$ is included in $X$ is $\text{Prob}(S \subseteq X) = \pi^S$. A standard estimation for $\pi^S$ is $\pi = s_N(S)/N$, where $s_N(S)$ is the number of transactions of the sample $N$ that contain $S$.

The (real) number $N^S$ of transactions that contains $S$ is a binomial random variable. We are interested in giving a good estimate of the value $N^S$, or, better, in determining the degree of certainty under which $\pi$ can be considered as a good estimate of $\pi^S$. Formally, we are interested in determining a such that

$$\text{Prob}(|N^S - N\pi^S| \geq a) = \epsilon$$
for a given $\epsilon$. Now, notice that $\text{Prob}([N^S - N\pi^S] \geq a) = 1 - \text{Prob}(\alpha \leq N^S \leq \beta)$ for some $\alpha$ and $\beta$. Since $N^S$ is binomial, the problem can be formulated as finding $\alpha$ and $\beta$ such that

$$
\sum_{s=\alpha}^{s=\beta} \binom{N}{s} \cdot \pi^s \cdot (1 - \pi)^{N-s} = \epsilon
$$

When $n$ is large, we can approximate the binomial distribution $B(N, \pi)$ with the normal distribution with mean $N\pi$ and variance $N\pi(1 - \pi)$, and hence compute $\alpha$ and $\beta$ that define confidence intervals. The interval $[\alpha, \beta]$ gives us the degree of accuracy of the (computed) support $\pi$ w.r.t. the (real) support ($\pi^S$).

- Rule relevance can be computed by exploiting confidence intervals. Let us consider a rule $R = A \Rightarrow B$, such that $S = A \cup B$. Let $[\alpha_R, \beta_R]$ be the confidence interval associated to the support of the rule. Since $A$ and $B$ are itemsets, we can compute a confidence interval $[\alpha_C, \beta_C]$ that measures the degree of accuracy of the (computed) probability $\pi^C$ that $B$ appears in a transaction, supposed that $A$ has already appeared.

Consider now a rule $R' = A \cup C \Rightarrow B$. $R'$ is relevant if and only if it has greater (real) confidence than the rule $R$. In order to guarantee such a condition, we check that $\pi^C_{R'} > \pi^C_R$ and that the confidence intervals are disjoint, i.e. $\alpha^R_C > \beta^R_R$. Such a test allows the removal of redundant rules where $C$ represent an itemset with very strong support (and hence is very likely to appear in a rule: take, e.g., the case of plastic bags in supermarket sales data).

- The $\chi^2$ test can be used to test the hypothesis that the itemset $A$ is actually correlated to item $B$ in a rule $A \Rightarrow B$. Starting from the contingency table of $A$ and $B$,

<table>
<thead>
<tr>
<th></th>
<th>$B$</th>
<th>$\neg B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$f_{11}$</td>
<td>$f_{10}$</td>
</tr>
<tr>
<td>$\neg A$</td>
<td>$f_{01}$</td>
<td>$f_{00}$</td>
</tr>
<tr>
<td>$f_{-1}$</td>
<td>$f_{-0}$</td>
<td>$N$</td>
</tr>
</tbody>
</table>

we compute the $\chi^2$ value given by

$$
\chi^2 = \frac{N(f_{11}f_{00} - f_{01}f_{10})^2}{f_{1-}f_{0-} - f_{-1}f_{-0}}
$$

The larger the $\chi^2$ value, the more evidence we have to reject the independence hypothesis.

- The interest of a rule defines the ratio between the joint probability of two variables with respect to their expected probabilities under the independence
assumption:
\[
I(A, B) = \frac{\Pr(A, B)}{\Pr(A) \Pr(B)}
\]

An interest value close to 1 corresponds to a statistical independence.

**Predictive Rules.** Association rules can be used to solve the problem of Partial Classification [6, 108], i.e., to discover the features of some given class. The approach is based on the computation of *predictive rules*, and is effective with respect to the others classification approaches when:

- there is a very large number of attributes;
- most values of attributes are missing;
- the class distribution is very skewed, and the classes of interest are low-frequency classes;
- the number of training examples is very large.

Given a relation \( R = A_1 \ldots A_n C \), predictive rules are of rules in which the consequent is an itemset of size 1, associated to attribute \( C \) (that is a target attribute):

\[
(A_{i_1} = a_1) \land (A_{i_2} = a_2) \ldots \land (A_{i_k} = a_k) \Rightarrow C = c
\]

A simple algorithm for computing predictive rules is shown in fig. 2.5.1. The algorithm computes set of frequent itemsets \( P = \{(A_{i_1} = a_1) \land (A_{i_2} = a_2) \ldots \land (A_{i_k} = a_k)\} \) in the segment of the dataset such that \( C = c \). For each pattern found, a rule \( P \Rightarrow (C = c) \) is formed, and the prediction accuracy of the rule is computed by adopting some significant interest measures, such as, e.g., the \( \chi^2 \) test of independence.

### 2.5.2 Data Segmentation

Also known as clustering, segmentation aims at separating the tuples of a given relations into subsets such that

- similarity of tuples of the same subset is maximized, and
- similarity of tuples of different subsets is minimized.

Clustering is also referred as *unsupervised learning*, since, differently from supervised learning (such as classification) the number of subsets that can be produced is not known. Clustering algorithms typically employ a two stage search: an outer loop over possible cluster numbers and an inner loop to fit the best possible clustering for a given number of clusters. Given the number \( k \) of clusters, clustering methods can be divided into five classes:
Algorithm Partial Classification,$(\sigma, \gamma, \epsilon)$

**Input:** An instance $\xi$ of a relation $R = A_1 \ldots A_n$; support $\sigma$ and confidence $\gamma$; significance level $\epsilon$ for $\chi^2$ test.

**Output:** A sequence of rules $(A_{i_1} = a_1) \land (A_{i_2} = a_2) \ldots \land (A_{i_k} = a_k) \Rightarrow C = c$

**Method:** Perform the following steps:

1. for each $c \in C$
2. Compute the sets of frequent itemsets $L$ on $\sigma_{C=c}(\xi)$;
3. For each itemset $P \in L$, compute the $\chi^2$ independence test for the rule $P \Rightarrow (C = c)$:
   - if the test fails, output the rule $P \Rightarrow (C = c)$

---

**Figure 2.11:** Profile analysis using association rules.

1. **Partition-based methods.** Such methods basically enumerate various partitions and then score them by some criterion. For example, the metric-distance based methods, when a distance measure is defined and the objective becomes finding the best $k$-way partition such that cases in each block of the partition are closer to each other to the cases in other clusters.

2. **Hierarchical methods.** Such methods group objects in cluster trees. Depending on how the cluster trees are built, they can be further partitioned in two classes. Agglomerative methods use a bottom-up approach, in which initially each element is a cluster, and clusters are merged at further steps. Divisive methods use a top-down approach: initially the whole dataset is a unique cluster, and subclusters are detected at subsequent steps. Birch and CURE [160] are examples of hierarchical methods.

3. **Density-based methods.** Differently from the previous approaches, that use a notion of metric distance, such methods use a notion of density: regions with higher densities form clusters. An example of density-based algorithm is DBSCAN [43].

4. **Model-based methods.** A model is hypothesized for each of the clusters and the idea is to find the best fit of that model to each cluster. Examples of model-based methods are Bayesian clustering [31], that use a Bayesian model of the available data to build clusters.

5. **Grid-based methods.** Such methods code the space of objects to analyze into a grid, where each cell represents an object.
2.5. **Undirected Knowledge Discovery**

---

**Algorithm $k$-Means($D$)**

**Input:** A dataset $D$.

**Output:** A sequence of cluster assignments for each tuple in $D$.

**Method:** Let $C$ be a set of $k$ tuples, randomly chosen from $D$. Perform the following steps:

1. while ($C$ is stable)
2. for each $t \in D$
3. for each $c_l \in C$
4. $k = \min_{i} d(t, c_l)$
5. $c_k = t$
6. recompute the centroids of ($C$);
7. return $C$.

---

Figure 2.12: The $k$-Means algorithm for computing data segments.

**K-Means Clustering**

A widely studied clustering algorithm is the $K$-Means method. Given a distance measure $d(x, y)$ between two objects $x$ and $y$, the algorithm initially defines $k$ initial points, representing cluster *means* (the centroids of $k$ partitions). Next, the algorithm iterates until some criterion is met (for example, until centers do not change, or until the error $E = \sum_{i=1}^{k} \sum_{o \in C_i} d(o, m_i)$ reaches a given threshold). At each iteration, objects are assigned to the nearest center, hence forming $k$ different agglomerations. Cluster means are recomputed and a new iteration is then started. The basic schema of the algorithm is shown in fig. 2.5.2.

Many variants of the above algorithm have been defined, which differ in the selection of the initial points, in the definition of the distance, and in the definition of the means. A significant variant is the $K$-medoid method [87], which uses medians instead of means.

Despite of its simplicity, there are many drawbacks of the above algorithm:

- it works only in spaces where the mean can be easily defined.

- it is a main memory algorithm, and hence poorly supports large datasets. CLARANS [117] is an extension proposed to cope with such a problem, mainly based on the idea of sampling.

- it is very inefficient with high dimensional datasets.

An example of application domain in which such an algorithm has poor performances is market-basket analysis. If we are interested in transactional clustering, i.e., in clustering the transactions of a relation with respect to a given transaction identifier in order to find significant groupings of the transactions, we have to deal
with huge dimensionality. Indeed, in order to apply the traditional $K$-means algorithm, we have to provide a binary representation of the given transactions. In sparse datasets, such a representation can be very expensive. In [69] a variant of the $K$-Means algorithm is proposed, that works with records of variant size, and uses a suitable notion of distance. Given two itemsets $I$ and $J$, we define a distance

$$d(I, J) = \frac{|I \cup J| - |I \cap J|}{|I \cup J|}$$

and the mean of a set of itemsets is defined as the threshold-union of the set:

$$\text{mean}_\sigma(I_1, \ldots, I_n) = \{x | x \in \bigcup_i I_i, \text{freq}(x, [I_1, \ldots, I_n]) \geq \sigma\}$$

where $\text{freq}(x, D)$ represent the number of itemsets $I \in D$ such that $x \in D$. Notice that the definition (and hence the significance) of the mean depends on the value of the threshold $\sigma \in [0, 1]$: when $\sigma = 0$, the mean is the union of the sessions, and when $\sigma = 1$, the union is the intersection.

### 2.6 Results Interpretation and Refinement

The tasks and techniques that we have presented in the previous sections are implemented by means of ad hoc algorithms, whose results in most cases are not directly useful for analysis purposes, but often need a tuning phase, in which they are interpreted and refined. Unsupervised learning techniques, such as clustering, are a typical example on how difficult it is to find a meaning for the results. The classes discovered by the given algorithm can be meaningless to the objectives of the user analyst, who may need to reiterate the discovery process, e.g. trying to transform/manipulate some attributes.

Even when the results are clear and easy to understand, like in the case of discovery of associations, the interpretation and usefulness of such results is not immediate. Some rules can be simply inexplicable, thus providing no useful information; other rules, instead, can be expected. For example, a rule “most purchaser of a supermarket buy plastic bags”, though clear, is rather trivial, and hence useless. The discovery of such rules requires the iteration of the process on a more refined data source, in which the occurrences of the “plastic bags” from the transactions have been eliminated.

Most of the data mining tools and methods require deep technical and statistical knowledge by the Data Analyst, in addition to a clear comprehension of the data. In addition, even if many techniques are semi-automatic, the overall process of data selection, preparation and result interpretation is slow and expensive, and require a strong user interaction.

In most cases, it is a qualitative kind of knowledge that is needed. Understanding data distributions, identifying trends, analyzing temporal evolutions all need
2.7 Open Problems

The diversity of data mining tasks and approaches poses many challenging research issues in Knowledge discovery. The design of data mining languages, the development of efficient and effective systems, the construction of integrated data mining environments, and the application of knowledge discovery techniques to solve large application problems are important research issues. This section describes some of these challenges.

2.7.1 Technology

One of the main obstacles in applying learning techniques to databases is the size of the database, and many research efforts have concentrated on this aspect. Some of the interesting research areas related to the problem include [22]:

- Develop mining algorithms that scale to large databases. There is a tradeoff between performance and accuracy as one surrenders to the fact that data resides primarily on disk, while many of the data mining techniques have been historically defined to work over memory-resident data and not much attention has been given to integrating them with database systems.

- Enhance database management systems to support new primitives for the efficient extraction of necessary sufficient statistics that are tightly coupled with mining algorithms and database access methods. Sufficient statistics are properties of the data that, from the perspective of data mining algorithms, eliminate the need for the data.

- Scale methods to parallel databases with hundreds of tables, thousands of fields, and terabytes of data. Issues of query optimization in these settings are fundamental.

A challenging research area is the extension of the Data Mining concept to different database areas. The current techniques have been defined and applied on simple relational data sets, in which the only data types under consideration were numeric or categorical. However, it could be very interesting and useful to develop schemes capable of mining over nonhomogeneous data sets (including mixtures of...
multimedia, video and text modalities) [152, 151], or to apply mining techniques to different application domains [17, 140].

2.7.2 Knowledge Extraction and Evaluation

The process of making decisions requires the combination of two kind of activities: knowledge acquisition and reasoning on the acquired knowledge according to the expert rules that characterize the business. Data mining techniques are an answer to the first issue in that they extract from raw data knowledge that is implicit and, more important, that is at a higher abstraction level.

However, while from the technological viewpoint knowledge discovery techniques are mature enough, we are still far away from an integrated methodology and support environment, which makes knowledge extraction feasible. It is now clear that the problem is that each step in the KDD process requires specific tools and expertise, as witnessed by the distance between data mining tools and query languages. There is a need to move from mining in rough data to mining in databases. More ambitiously, there is a need for combining the inductive capabilities of the data mining tools with reasoning, to the purpose of addressing the difficult analysis tasks posed by key applications.

One such example is market basket analysis, which is rapidly becoming a key factor of success in the highly competing scene of big supermarket retailers. Association rules are often too low-level to be directly used as a support of marketing decisions. Market analysts expect answers to more general questions, such as “Is supermarket assortment adequate for the company’s target class of customers?” “Is a promotional campaign effective in establishing a desired purchasing habit in the target class of customers?” These are business rules, and association rules are necessary, albeit insufficient, basic mechanisms for their construction. Business rules require also the ability of combining association rule mining with deduction, or reasoning: reasoning on the temporal dimension, reasoning at different levels of granularity of products, reasoning on the spatial dimension, reasoning on association rules themselves.

A coherent formalism, capable of dealing uniformly with induced knowledge and background, or domain, knowledge, would represent a breakthrough in the design and development of decision support systems, in diverse challenging application domains. The advantages of such an integrated formalism are, in principle:

- a high degree of expressiveness in specifying expert rules, or business rules;

- the ability to formalize the overall KDD process, thus tailoring a methodology to a specific class of applications;

- the separation of concerns between the specification level and the mapping to the underlying databases and data mining tools.
Chapter 3

Deductive Databases

Abstract

Relational query languages are often powerless to express complete applications, and are thus embedded in traditional programming languages, resulting in *impedance mismatch*. Deductive Database systems, on the other hand, can be used as general-purpose programming languages. They can be used to express facts, deductive information, recursion, queries, updates, and integrity constraints in a uniform way.

In this thesis we consider the *Datalog++* deductive database language, which extends *Datalog* with mechanisms supporting temporal, nonmonotonic and nondeterministic reasoning. *Datalog++*, which is essentially a fragment of *LDC++* [8], and is advocated in [158, Chap. 10], revealed a highly expressive language, with applications in diverse areas such as AI planning [25], active databases [154], object databases [55], semistructured information management and Web restructuring [55, 63]. Although semantic issues are not a central point of this thesis, it is important to highlight that a language has a formal foundation which accommodates and integrates the temporal, nonmonotonic and nondeterministic mechanisms. A study of its semantics is provided in [67, 8, 25] and thoroughly refined in [114, 56, 57] thus providing a basis to sound and efficient implementations and optimization techniques.

This chapter has the objective to introduce the distinguishing features of the *Datalog++* language that are amenable to model complex applications. In particular, one such feature is the notion of *iterative user-defined aggregate*, a refinement of the notion of user-defined aggregate introduced in [157, 159]. We shall provide a semantics for such a construct, and illustrate its expressiveness in modeling complex applications. The next chapters will show in particular how such a feature makes *Datalog++* suitable enough to be successfully applied to knowledge discovery in databases.
3.1 Expressiveness and Semantics of Datalog

The notion of *relational completeness* for database query languages essentially means that they should be at least as expressive as relational algebra expressions (and relational calculus formulas). Nonrecursive safe Datalog, the basis of deductive databases, is essentially a friendly syntax to express relational queries, and to extend the query facilities of the relational calculus with *recursion* and *complex records* [158, 146, 1]. Datalog’s simplicity in expressing complex queries impacted on the database technology and nowadays recursive queries/views have become part of the SQL3 standard. Recursive queries find natural applications in all areas of information systems where computing transitive closures or traversals is an issue, such as in bill-of-materials queries, route or plan formation, graph traversals, and so on.

The great increase in expressive power brought by recursion is not without a price, since the fixpoint semantics and other “natural” semantics no longer hold when negation or other nonmonotonic constructs are used in Datalog programs [107]. Negation in databases and in knowledge bases is nonmonotonic because negative information is not stored explicitly but, rather, it is derived by default. Moreover, the notion of stable models [49] that provides a sound declarative semantics for programs with negated goals cannot be used directly as operational semantics because of its exponential complexity.

Therefore, the identification of classes of Datalog programs whose stable models can be computed in polynomial time has been the focus of much research in deductive databases [156, 158]. The notion of stratification (where negation can be used on predicates defined in lower strata, but not on the predicate defined by the same rule) represents a crucial notion for the introduction of nonmonotonic reasoning in deductive databases. From the original idea in [7] of a static stratification based on predicate dependencies, stratified negation has been refined to deal with dynamic notions, as in the case of locally stratified programs [121] and modularly stratified programs [127]. Dynamic, or local, stratification has a close connection with temporal reasoning, as the progression of time points yields an obvious stratification of programs—consider for instance Datalog [35]. It is therefore natural that non monotonic and temporal reasoning are combined in several deductive database languages, such as those in [104], [97], [55], [158, Chap. 10].

Nondeterminism is introduced in deductive databases by means of the choice construct. The original proposal in [101] was later revised in [130], and refined in [66]. These studies exposed the close relationship connecting nonmonotonic reasoning with nondeterministic constructs, leading to the definition of a stable model semantics for choice. While the declarative semantics of choice is based on stable model semantics which is untractable in general, choice is amenable to efficient implementations, and it is actually supported in the logic database language *LDC* [115] and its evolution *LDC++* [8].

A striking mismatch is apparent between the above two lines of research: nondeterminism leads to a multiplicity of (stable) models, whereas stratification leads to
a unique (perfect) model. A comprehensive study should address the combination of the two lines, which occur in Datalog++, and should require the development of a non deterministic iterated fixpoint procedure. An approach to this problem is sketched in [25], with reference to locally stratified programs augmented with choice. In the following section, we present the treatment of Datalog++ programs shown in [57], which resumes the approach in [25] and repairs an inconvenience concerning the incompleteness of the iterated fixpoint procedure.

3.2 From Datalog to Datalog++

It is widely recognized that the expressiveness of Datalog's (recursive) rules is limited, and several extensions, along various directions, have been proposed. In this section, we address in particular two such directions, namely nondeterministic and nonmonotonic reasoning, supported respectively by the choice construct and the notion of XY-stratification. We introduce these mechanisms by means of a few examples, which are meant to point out the enhanced query capabilities.

We proceed by assigning a natural, non effective, semantics for Datalog++ is assigned using the notion of a stable model. An effective semantics is then assigned using an iterative procedure which exploits the stratification induced by the progression of the temporal argument. Finally, we show that the two semantics are equivalent, provided that a natural syntactic restriction is fulfilled, which imposes a disciplined use of the temporal argument within the choice construct. It is worth noting that, on the basis of this result, it is possible to define a more concrete operational semantics, and a repertoire of optimization techniques, especially tailored for Datalog++ [57].

3.2.1 Nondeterministic Choice

The choice construct is used to nondeterministically select subsets of answers to queries, which obey a specified FD constraint\(^1\). For instance, the rule

\[
st_{\text{ad}}(St, Ad) \leftarrow \text{major}(St, Area), \text{faculty}(Ad, Area), \text{choice}((St), (Ad)).
\]

assigns to each student a unique, arbitrary advisor from the same area, since the choice goal constrains the st_{\text{ad}} relation to obey the FD (St → Ad). Therefore, if the base relation major is formed by the tuples \{(smith, db), (gray, se)\} and the base relation faculty is formed by \{(brown, db), (scott, db), (miller, se)\}, then there are two possible outcomes for the query st_{\text{ad}}(St, Ad): either the tuples \{(smith, brown), (gray, miller)\} or the tuples \{(smith, scott), (gray, miller)\}. In

\(1\) However, choice can be interestingly employed to compute new deterministic queries, which are inexpressible in Datalog, as well as in pure relational calculus. A thorough account on programming with nondeterminism in deductive databases can be found in [50, 67].
practical systems, such as CDLC++, one of these two solutions is computed and presented as a result.

Thus, a first use of choice is in computing nondeterministic, nonrecursive queries. However, choice can be combined with recursion, as in the following rules which compute an arbitrary ordering of a given relation $r$:

$$ord_r(\text{root}, \text{root}).$$

$$ord_r(X, Y) \leftarrow ord_r(X, Y), \text{choice}(X, Y), \text{choice}(Y, X).$$

Here root is a fresh constant, conveniently used to simplify the program. If the base relation $r$ is formed by $k$ tuples, then there are $k!$ possible outcomes for the query $ord_r(X, Y)$, namely a set:

$$\{ord_r(\text{root}, \text{root}), ord_r(\text{root}, t_1), ord_r(t_1, t_2), \ldots, ord_r(t_{k-1}, t_k)\}$$

for each permutation $\{t_1, \ldots, t_k\}$ of the tuples of $r$. Therefore, in each possible outcome of the mentioned query, the relation $ord_r$ is a total ordering of the tuples of $r$. The double choice constraint in the recursive rule specifies that the successor and predecessor of each tuple of $r$ is unique.

The semantics of choice is assigned using the so-called stable model semantics of Datalog− programs, a concept originating from autoepistemic logic, which was applied to the study of negation in Horn clause languages by Gelfond and Lifschitz [49]. To define the notion of a stable model we need to introduce a transformation $H$ which, given an interpretation $I$, maps a Datalog− program $P$ into a positive Datalog program $H(P, I)$:

$$H(P, I) = \{A \leftarrow B_1, \ldots, B_n \mid A \leftarrow B_1, \ldots, B_m, \neg C_1, \ldots, \neg C_m \in \text{ground}(P) \wedge \{C_1, \ldots, C_m\} \cap I = \emptyset\}$$

Next, we define:

$$S_P(I) = T_{H(P, I)} \uparrow \omega$$

Then, $M$ is said to be a stable model of $P$ if $S_P(M) = M$. In general, Datalog− programs may have zero, one or many stable models. The multiplicity of stable models can be exploited to give a declarative account of nondeterminism.

We can in fact define the stable version of a program $P$, $SV(P)$, to be the program transformation where all the references to the choice atom in a rule $r : H \leftarrow B, \text{choice}(X, Y)$ are replaced by the atom $\text{chosen}_r(X, Y)$, and define the $\text{chosen}_r$ predicate with the following rules:

$$\text{chosen}_r(X, Y) \leftarrow B, \neg \text{diffchoice}_r(X, Y).$$

$$\text{diffchoice}_r(X, Y) \leftarrow \text{chosen}_r(X, Y_1), Y \neq Y_1.$$
Notice that, by construction, each occurrence of a choice atom has its own pair of chosen and diffchoice atoms, thus bounding the scope of the atom to the rule it appears in. The various stable models of the transformed program $SV(P)$ thus correspond to the choice models of the original program.

### 3.2.2 XY-programs

Another notion used in this section is that of XY-programs originally introduced in [157]. The language of such programs is Datalog+$\uparrow$, which admits negation on body atoms and a unary constructor symbol, used to represent a temporal argument usually called the stage argument. A general definition of XY-programs is the following. A set $P$ of rules defining mutually recursive predicates, is an XY-program if it satisfies the following conditions:

1. each recursive predicate has a distinguished stage argument;

2. every recursive rule $r$ is either an X-rule or a Y-rule, where:
   - $r$ is an X-rule when the stage argument in every recursive predicates in $r$ is the same variable,
   - $r$ is a Y-rule when (i) the head of $r$ has a stage argument $s(J)$, where $J$ is a variable, (ii) some goal of $r$ has $J$ as its stage argument, and (iii) the remaining recursive goals have either $J$ or $s(J)$ as their stage argument.

Intuitively, in the rules of XY-programs, an atom $p(J, \perp)$ denotes the extension of relation $p$ at the current stage (present time) $J$, whereas an atom $p(s(J), \perp)$ denotes the extension of relation $p$ at the next stage (future time) $s(J)$. By using a different primed predicate symbol $p'$ in the $p(s(J), \perp)$ atoms, we obtain the so-called primed version of an XY-program. We say that an XY-program is XY-stratified if its primed version is a stratified program. Intuitively, if the dependency graph of the primed version has no cycles through negated edges, then it is possible to obtain an ordering on the original rules modulo the stage arguments. As a consequence, an XY-stratified program is also locally stratified, and has therefore a unique stable model that coincides with its perfect model [121].

Let $P$ be an XY-stratified program. Then, for each $i > 0$, define $P_i$ as

$$P_i = \{ r[s^i(nil)/I] \mid r \in P, I \text{ is the stage argument of the head of } r \}$$

(here $r[x/I]$ stands for $r$ where $I$ is replaced by $x$) i.e., $P_1$ is the set of rule instances of $P$ that define the predicates with stage argument $s^i(nil) = i$. Then the iterated fixpoint procedure for computing the (unique) minimal model of $P$ can be defined as follows:

1. compute $M_0$ as the minimal model of $P_0$;


2. for each \( j > 0 \) compute \( M_j \) as the minimal model of \( P_j \cup M_{j-1} \).

Notice that for each \( j \geq 0 \), \( P_j \) is stratified by the definition, and hence its perfect model \( M_j \) is computable via an iterated fixpoint procedure.

In this thesis, we use the name Datalog++ to refer to the language of XY-programs augmented with choice goals.

### 3.2.3 A Semantics for Datalog++

When choice constructs are allowed in XY-programs, a multiplicity of stable models exists for any given program, and therefore it is needed to clarify how this phenomenon combines with the iterated fixpoint semantics of choice-free XY-programs. This task is accomplished in three steps.

1. A general result states that, whenever a Datalog\( \neg \) program \( P \) is stratifiable into a hierarchy of recursive cliques \( Q_1, Q_2, \ldots \), then any stable model of the entire program \( P \) can be reconstructed by iterating the construction of approximating stable models, each associated to a clique.

2. Under a syntactic restriction on the use of the choice construct that does not compromise expressiveness, Datalog++ programs can be naturally stratified into a hierarchy of recursive cliques \( Q_1, Q_2, \ldots \), by using the temporal arguments of recursive predicates.

3. By the observation in 2., we can apply the general result in 1. to Datalog++ programs, thus obtaining that the stable models of the entire program can be computed by an iterative fixpoint procedure which follows the stratification induced by the temporal arguments.

Given a (possibly infinite) program \( P \), consider a (possibly infinite) topological sort of its distinct recursive cliques \( Q_1 \prec Q_2 \prec \ldots \prec Q_i \prec \ldots \), induced by the dependency relation over the predicates of \( P \). Given an interpretation \( I \), we use the notation \( I_i \) to denote the subset of atoms of \( I \) whose predicate symbols are predicates defined in clique \( Q_i \).

The following observations are straightforward:

- \( \bigcup_{i>0} I_i = I \), and analogously \( \bigcup_{i>0} Q_i = P \);
- the predicates defined in \( Q_{i+1} \) depend only on the definitions in \( Q_1 \cup \ldots \cup Q_i \); as a consequence, the interpretation of \( Q_{i+1} \) is \( I_1 \cup \ldots \cup I_i \cup I_{i+1} \) (i.e., we can ignore \( \bigcup_{j>i+1} I_j \)).

The next definition shows how to transform each clique, within the given topological ordering, in a self-contained program which takes into account the information deduced by the previous cliques. Such transformation resembles the Gelfond-Lifschitz transformation [49].
Definition 3.1 ([57]). Consider a program $P$, a topological sort of its cliques $Q_1 \prec Q_2 \prec \ldots \prec Q_i \ldots$, and an interpretation $I = \bigcup_{i \geq 0} I_i$. Now define

$$Q_i^{ed(I)} = \{ H \leftarrow B_1, \ldots, B_n \mid H \leftarrow B_1, \ldots, B_n, C_1, \ldots, C_m \in \text{ground}(Q_i) \\land B_1, \ldots, B_n \text{ are defined in } Q_i \\land C_1, \ldots, C_m \text{ are defined in } (Q_1 \cup \ldots \cup Q_{i-1}) \\land I_1 \cup \ldots \cup I_{i-1} \models C_1, \ldots, C_m \}$$

The idea underlying the transformation is to remove from each clique $Q_i$ all the dependencies induced by the predicates which are defined in lower cliques. We abbreviate $Q_i^{ed(I)}$ by $Q_i^{ed}$, when the interpretation $I$ is clear by the context.

Example 3.1. Consider the program $P = \{ p \leftarrow q, r. \ q \leftarrow t, r \leftarrow q, s \}$ and the cliques $Q_1 = \{ q \leftarrow r, t. \ s \leftarrow q, s \}$ and $Q_2 = \{ p \leftarrow q, r \}$. Now, consider the interpretation $I = \{ s, q, r \}$. Then $Q_i^{ed} = \{ q \leftarrow r, t. \ r \leftarrow q, s \}$ and $Q_2^{ed} = \{ p \leftarrow \}$. <

The following Lemma 1 states the relation between the models of the transformed cliques and the models of the program. We abbreviate $I_1 \cup \ldots \cup I_i$ with $I^{(i)}$, and analogously for $Q^{(i)}$.

Lemma 1 ([57]). Given a (possibly infinite) Datalog program $P$ and an interpretation $I$, let $Q_1 \prec Q_2 \prec \ldots \prec Q_i \ldots$ and $I_1 \prec I_2 \prec \ldots \prec I_i \ldots$ be the topological sorts on $P$ and $I$ induced by the dependency relation of $P$. Then the following statements are equivalent:

1. $S_P(I) = I$
2. $\forall i > 0. \ S_{Q^{(i)}}(I_i) = I_i$
3. $\forall i > 0. \ S_{Q^{(i)}}(I^{(i)}) = I^{(i)}$

Proof (sketch). The proof is structured as follows: (1) $\iff$ (3) and (2) $\iff$ (3).

(3) $\implies$ (1) We next show that (a) $S_P(I) \subseteq I$, and (b) $I \subseteq S_P(I)$.

(a) Each rule in $H(P, I)$ comes from a rule $r$ of $P$, which in turn appears in $Q^{(i)}$ for some $i$, and then $I^{(i)}$ is a model of $r$, by the hypothesis. No atom in $I \setminus I^{(i)}$ appears in $r$, so also $I$ is model of $r$. $I$ is then a model of $H(P, I)$, and hence $S_P(I) \subseteq I$.

(b) If $A \in I$, then $A \in I^{(i)}$ for some $i$, so (by the hypothesis and definition of $S_P$) for each $I^*$ such that $I^* = T_{H(Q^{(i)}, I^{(i)})}(I^*)$, $A \in I^*$. Moreover, for each $I^*$ such that $I^* = T_{H(P, I)}(I^*)$, it is readily checked that for each $i$ $I^{(i)} = T_{H(Q^{(i)}, I^{(i)})}(I^{(i)})$, and then $I \subseteq S_P(I)$.
(1) $\implies$ (3) We observe that $I = \min\{I^k : I^k = T_{H(P,I)}(I^*)\}$, which implies:
$I^{(i)} = \min\{I^{k(i)} : I^{k(i)} = T_{H(Q^{(i)}),I^{(i)}}(I^{(i)})\}$.

(2) $\implies$ (3) We proceed by induction on $i$. The base case is trivial. In the inductive case, we next show that (a) $S_{Q^{(i)}}(I^{(i)}) \subseteq I^{(i)}$, and (b) vice versa.

(a) Notice that from the induction hypothesis, $I^{(i)} \models Q^{(i-1)}$, and then it suffices to show that $I^{(i)} \models Q_i$ (by a simple case analysis).

(b) Exploiting the induction hypothesis, we see that $I^{(i-1)} \subseteq S_{Q^{(i-1)}}(I^{(i-1)}) = S_{Q^{(i)}}(I^{(i)}) \subseteq S_{Q^{(i)}}(I^{(i)})$ (by definition of $H(P,I)$). We now show by induction on $n$ that $\forall n \geq 0 T^n_{H(Q^{red},I_i)} \subseteq T^n_{H(Q^{(i)},I^{(i)})}$. The base case $n = 0$ is trivial. In the induction case $n > 0$, if $A \in T^n_{H(Q^{red},I_i)}$, then there exists a rule $A \leftarrow b_1, \ldots, b_h$ in $H(Q^{red},I_i)$ such that $\{b_1, \ldots, b_h\} \subseteq T_{H(Q^{red},I_i)}^n$. Now, by definition of $H$ and $Q^{red}$, there exists a rule:

$A \leftarrow b_1, \ldots, b_h, \neg e_1, \ldots, \neg e_j, d_1, \ldots, d_k, \neg e_1, \ldots, \neg e_l$
in $Q_i$ such that $\{e_1, \ldots, e_j\} \cap I_i = \emptyset$ and $I^{(i-1)} \models d_1 \land \ldots \land d_k \land \neg e_1 \land \ldots \land \neg e_l$. Observe now that by definition of $H$, $A \leftarrow b_1, \ldots, b_h, d_1, \ldots, d_k \in H(Q^{(i)}, I^{(i)})$. Furthermore, by the induction hypothesis and $I^{(i-1)} \subseteq S_{Q^{(i)}}(I^{(i)})$, we have the following: $\{b_1, \ldots, b_h, d_1, \ldots, d_k\} \subseteq T_{H(Q^{(i)},I^{(i)})}$. Hence, by definition of $T^\omega$, $A \in T_{H(Q^{(i)},I^{(i)})}^\omega$, that is $A \in S_{Q^{(i)}}(I^{(i)})$. This completes the innermost induction, and we obtain that $I_i = S_{Q^{red}}(I_i) \subseteq S_{Q^{(i)}}(I^{(i)})$.

(3) $\implies$ (2) We proceed is a way similar to the preceding case. To see that $\forall i I_i \subseteq S_{Q^{red}}(I_i)$, it suffices to verify that for each rule instance $r$ with head $A$, the following property holds: $\forall n A \in T^n_{H(Q^{(i)},I^{(i)})} \Rightarrow A \in T^n_{H(Q^{red},I_i)}$. For the converse, we simply observe that $I_i$ is a model of $Q^{red}$.

This result states that an arbitrary Datalog program has a stable model if and only if each its approximating clique, according to the given topological sort, has a local stable model. This result gives us an intuitive idea for computing the stable models of an approximable program by means of the computation of the stable models of its approximating cliques.

Notice that Lemma 1 holds for arbitrary programs, provided that a stratification into a hierarchy of cliques is given. In this sense, this result is more widely applicable than the various notions of stratified programs, such as that of modularly stratified programs [127], in which it is required that each clique $Q^{red}_i$ is locally stratified. On the contrary, we do not require here that each clique is, in any sense, stratified. This is motivated by the objective of dealing with non determinism, and justifies why we adopt the (nondeterministic) stable model semantics, rather than other
deterministic semantics for (stratified) Datalog\~\ programs, such as, for instance, perfect model semantics [1, 21].

We turn now our attention to XY-programs. The result of instantiating the clauses of an XY-program $P$ with all possible values (natural numbers) of the stage argument, yields a new program $SG(P)$ (for stage ground). More precisely, $SG(P) = \bigcup_{i \geq 0} P_i$, where

$$P_i = \{ r[i/I] \mid r \text{ is a rule of } P, I \text{ is the stage argument of } r \}.$$

The stable models of $P$ and $SG(P)$ are closely related:

**Lemma 2 ([57]).** Let $P$ be an XY-program. Then, for each interpretation $I$:

$$Sp(I) = I \iff S_{SG(P)}(I) = I$$

**Proof (sketch).** We show by induction that $\forall n. T^n_{H(SG(P), I)}(\emptyset) = T^n_{H(P, I)}(\emptyset)$, which implies the thesis. The base case is trivial. For the inductive case, observe that since $P$ is XY-stratified, if $A \in T^{n+1}_{H(P, I)}(\emptyset)$ then for each rule $A \leftarrow B_1, \ldots, B_n \in H(P, I)$ such that $\{B_1, \ldots, B_n\} \in T^n_{H(P, I)}(\emptyset) = T^n_{H(SG(P), I)}(\emptyset)$, we have $A \leftarrow B_1, \ldots, B_n \in H(SG(P), I)$.

Vice versa, if $A \in T^{n+1}_{H(SG(P), I)}(\emptyset)$ then for each rule $A \leftarrow B_1, \ldots, B_n \in H(SG(P), I)$ such that $\{B_1, \ldots, B_n\} \in T^n_{H(SG(P), I)}(\emptyset) = T^n_{H(P, I)}(\emptyset)$, we have $A \leftarrow B_1, \ldots, B_n \in H(P, I)$.

However, the dependency graph of $SG(P)$ (which is obviously the same as $P$) does not induce necessarily a topological sort, because in general XY-programs are not stratified, and therefore Lemma 1 is not directly applicable. To tackle this problem, we distinguish the predicate symbol $p$ in the program fragment $P_i$ with the same predicate symbol in all other fragments $P_j$ with $j \neq i$, by differentiating the predicate symbols using the temporal argument. Therefore, if $p(i, x)$ is an atom involved in some rule of $P_i$, its modified version is $p_i(x)$. More precisely, we introduce, for any XY-program $P$, its modified version $SO(P)$ (for stage-out), defined by $SO(P) = \bigcup_i SO(P_i)$, where $SO(P_i)$ is obtained from the program fragment $P_i$ of $SG(P)$ by extracting the stage arguments from any atom, and adding it to the predicate symbol of the atom. Similarly, the modified version $SO(I)$ of an interpretation $I$ is defined. Therefore, the atom $p(i, x)$ is in $I$ iff the atom $p_i(x)$ is in $SO(I)$, where $i$ is the value in the stage argument position of relation $p$.

Unsurprisingly, the stable models of $SG(P)$ and $SO(P)$ are closely related:

**Lemma 3 ([57]).** Let $P$ be an XY-program. Then, for each interpretation $I$:

$$S_{SG(P)}(I) = I \iff S_{SO(P)}(SO(I)) = SO(I).$$

**Proof (sketch).** It is easy to see that $SO(SG(P)) = SO(P)$. Hence, the least Herbrand models of $SO(H(SG(P), I))$ and $H(SO(P), SO(I))$ coincide.
Iterated stable model procedure

**Input:** A Datalog++ program $P$;

**Output:** a stable model $M$ of $P$;

**Method:** Iterate the following inductive steps, until a fixpoint is found:

- **Base case.** $M_0$ is a stable model of the bottom clique $SO(P)_0$.
- **Induction case.** For $i > 0$, $M_i$ is a stable model of $SO(P)_i^\text{red}(M_i)$, i.e. the clique $SO(P)_i$ reduced with respect to $M_0 \cup \cdots \cup M_{i-1}$.

The interpretation $M = \bigcup_{i \geq 0} M_i$ is the iterated stable model of $P$.

---

**Figure 3.1: Iterated Stable Model Procedure.**

Our aim is now to conclude that, for a given Datalog++ program $P$:

(a) $SO(P)_0 \prec SP(P)_1 \prec \cdots$ is the topological sort over $SO(P)$ in the hypothesis of Lemma 1\(^2\); recall that, for $i \geq 0$, the clique $SO(P)_i$ consists of the rules from $SO(P)$ with stage argument $i$ in their heads;

(b) by Lemmas 1, 2 and 3, an interpretation $I$ is a stable model of $P$ iff $I$ can be constructed as $\bigcup_{i \geq 0} I_i$, where, for $i \geq 0$, $I_i$ is a stable model of $SO(P)^\text{red}(I_i)$, i.e. the clique $SO(P)_i$ reduced by substituting the atoms deduced at stages earlier than $i$.

On the basis of (b) above, it is possible to define an iterative procedure to construct an arbitrary stable model $M$ of $P$ as the union of the interpretations $M_0, M_1, \ldots$, defined in 3.1. It should be observed that this construction is close to the procedure called *iterated choice fixpoint* in [25]. Also, following the approach of [67], each local stable model $M_i$ can in turn be efficiently constructed by a nondeterministic fixpoint computation, in polynomial time.

Unfortunately [114], the desired result that the notions of stable model and iterated stable model coincide does not hold in full generality, in the sense that the iterative procedure is not complete for arbitrary Datalog++ programs. In fact, as demonstrated by the example below, an undisciplined use of *choice* in Datalog++ programs may cause the presence of stable models that cannot be computed incrementally over the hierarchy of cliques.

**Example 3.2.** Consider the following simple Datalog++ program $P$:

$q(0, a)$.
$q(s(I), b) \leftarrow q(I, a)$.
$p(I, X) \leftarrow q(I, X), \text{choice}((), X)$.

\(^2\)In general, $SO(P)_i$ can be composed by more than one clique, so that in the above expression it should be replaced by $SO(P)^\text{red}(I_i)_1 \prec \cdots \prec SO(P)^\text{red}(I_i)_n$. However, for ease of presentation we ignore it, since such general case is trivially deducible from what follows.
In the stable version \( SV(P) \) of \( P \), the rule defining predicate \( p \) is replaced by:

\[
p(I, X) \leftarrow q(I, X), \text{chosen}(X).
\]

\[
\text{chosen}(X) \leftarrow q(I, X), \neg\text{diff choice}(X).
\]

\[
\text{diff choice}(X) \leftarrow \text{chosen}(Y), Y \neq X.
\]

It is easy to see that \( SV(P) \) admits two stable models: \( \{q(0, a), q(s(0), b), p(0, a)\} \)
and \( \{q(0, a), q(s(0), b), p(s(0), b)\} \). However, only the first model is an iterated stable models, and therefore the second model cannot be computed using the \textit{iterated choice fixpoint} of \([25]\).

The technical reason for this problem is that the free use of the \textit{choice} construct inhibits the possibility of defining a topological sort on \( SO(P) \) based on the value of the stage argument. In the Example 3.2, the predicate dependency relation of \( SO(SV(P)) \) induces a dependency among stage \( i \) and the stages \( j > i \), because of the dependency of the \textit{chosen} predicate from the predicates \( q_i \) for all stages \( i \geq 0 \).

To prevent this problem, it is suffices to require that \textit{choice} goals refer the stage argument \( I \) in the domain of the associated functional dependency. The Datalog++ programs which comply with this constraint are called \textit{choice-safe}. The following is a way to turn the program of Example 3.2 into a choice-safe program (with a different semantics):

\[
p(I, X) \leftarrow q(I, X), \text{choice}(I, X).
\]

This syntactic restriction, moreover, does not greatly compromise the expressiveness of the query language, in that it is possible to simulate within this restriction most of the general use of \textit{choice} (see [114]).

The above considerations are summarized in the following main result, which, under the mentioned restriction of choice-safety, is a direct consequence of Lemmas 1, 2 and 3.

**Theorem 1 (Correctness and completeness of the iterated stable model procedure).**

Let \( P \) be a choice-safe Datalog++ program and \( I \) an interpretation. Then \( I \) is a stable model of \( SV(P) \) iff it is an iterated stable model of \( P \).

The following example shows a computation with the iterated stable model procedure.

**Example 3.3.** Consider the following Datalog++ version of the \textit{seminaive} program, discussed in [157], which non-deterministically computes a maximal path from node \( a \) over a graph \( g \):

\[
delta(0, a).
\]
\[
delta(s(I), Y) \leftarrow \delta(I, X), g(X, Y), \neg\text{all}(I, Y), \text{choice}((I, X), Y).
\]
\[
\text{all}(I, X) \leftarrow \delta(I, X).
\]
\[
\text{all}(s(I), X) \leftarrow \text{all}(I, X), \delta(s(I), X).
\]
Assume that the graph is given by $g = \{ \langle a, b \rangle, \langle b, c \rangle, \langle b, d \rangle, \langle d, e \rangle \}$. The following interpretations are carried out at each stage of the iterated stable model procedure:

1. $I_0 = \{ \text{delta}_0(a), \text{all}_0(a) \}$.
2. $I_1 = \{ \text{all}_1(a), \text{all}_1(b), \text{delta}_1(b) \}$.
3. $I^1_2 = \{ \text{all}_2(a), \text{all}_2(b), \text{delta}_2(c), \text{all}_2(c) \}$, 
   $I^2_2 = \{ \text{all}_2(a), \text{all}_2(b), \text{delta}_2(d), \text{all}_2(d) \}$
4. $I^3_3 = \emptyset, I^3_3 = \{ \text{all}_3(a), \text{all}_3(b), \text{all}_3(d), \text{delta}_3(e), \text{all}_3(e) \}$
5. $I_j = \emptyset$ for $j > 3$.

By Theorem 1, we conclude that there are two stable models for the program: $I^1 = I_0 \cup I_1 \cup I^1_2$ and $I^2 = I_0 \cup I_1 \cup I^2_2 \cup I^3_3$. Clearly, any realistic implementation, such as that provided in $\mathcal{LDC}++$, computes non deterministically only one of the possible stable models. \hfill $\blacktriangleleft$

### 3.3 Query Answering with Datalog++

This section is aimed at showing how the features of Datalog++ can be fruitfully exploited to model complex queries capabilities. As a deductive database, Datalog++ can be used in a variety of application domains including scientific modeling, financial analysis, decision support, language analysis, parsing, and various applications of transitive closure such as bill-of-materials and path problems [124]. However, it is best suited for applications in which a large amount of data must be accessed and complex queries hardly definable must be supported.

A main extension of the relational model supports nested relations. This extension is easily accomplished in a logic-based framework using functors to define complex objects, such as sets. Thus,

\[
\text{dept\textunderscore list}(\text{shoe}, \{ \text{adams}, \text{jones}, \text{zong}, \text{smith} \})
\]

denotes that $\{ \text{adams}, \text{jones}, \text{zong}, \text{smith} \}$ is the set of employees working in the shoe department. Practical languages, such as $\mathcal{LDC}++$, provide many ad-hoc predicates that implement typical operations over sets, such as membership/substring testing, union, difference, etc.

If we are restricted to Datalog proper, where sets are not allowed, we can use the following representation, where the list is kept as a chain connecting the employees:

\[
\begin{align*}
\text{dept\textunderscore chain}(\text{shoe}, \text{adams}). \\
\text{emp\textunderscore chain}(\text{adams}, \text{jones}). \\
\text{emp\textunderscore chain}(\text{jones}, \text{zong}). \\
\text{emp\textunderscore chain}(\text{zong}, \text{smith}).
\end{align*}
\] (3.1)
From either lists or chains, it is simple to derive a set of 1NF facts representing the same information.

Example 3.4. For the chain representation of sets we can use the following (transitive closure) rules, normalizing the relation \texttt{dept\_list}.

\[
\begin{align*}
\text{flat}(\text{Dept}, \text{Emp}) & \leftarrow \text{dept\_chain}(\text{Dept}, \text{Emp}). \\
\text{flat}(\text{Dept}, \text{Emp2}) & \leftarrow \text{flat}(\text{Dept}, \text{Emp1}), \text{emp\_chain}(\text{Emp1}, \text{Emp2}).
\end{align*}
\]

These rules applied to the previous \texttt{dept\_chain} and \texttt{emp\_chain} yield:

\[
\begin{align*}
\text{flat}(\text{shoe}, \text{adams}). \\
\text{flat}(\text{shoe}, \text{jones}). \\
\text{flat}(\text{shoe}, \text{zong}). \\
\text{flat}(\text{shoe}, \text{smith}).
\end{align*}
\]

In summary, there exist simple recursive rules that transform a nested-relation representation to a flat (1NF) representation. The inverse transformation, however, reveals the limitation of stratified datalog. It is not possible to translate flat relations into nested ones, unless we assume that there is an underlying order in the database or we use choice.

In Datalog++ [50], we can write the rules of Example 3.5 using recursive rules with choice and stratified negation. This program is similar to that of section 3.2.1.

Example 3.5. De-normalizing relations.

\[
\begin{align*}
\text{echain}(\text{root}, \text{root}) & \leftarrow \text{flat}(\text{shoe}, \_). \\
\text{echain}(\text{Emp1}, \text{Emp2}) & \leftarrow \text{echain}(\_, \text{Emp1}), \text{flat}(\text{shoe}, \text{Emp2}), \\
& \quad \text{choice}((\text{Emp1}), (\text{Emp2})), \text{choice}((\text{Emp2}), (\text{Emp1})). \\
\text{empchain}(\text{Em1}, \text{Em2}) & \leftarrow \text{echain}(\text{Em1}, \text{Em2}), \text{Em1} \neq \text{root}. \\
\text{depchain}(\text{shoe}, \text{Em}) & \leftarrow \text{flat}(\text{shoe}, \text{Em}), \text{echain}(\text{root}, \text{Em}), \text{Em} \neq \text{root}.
\end{align*}
\]

Observe that we obtain the original nested-relation representation of Equation 3.1, modulo the renaming of \texttt{empchain} into \texttt{emp\_chain}, of \texttt{depchain} into \texttt{dep\_chain}, and, possibly, a different order for the employee-names in the chain.

Various deterministic queries which could not be expressed on flat relations are easily expressed on the chain-based representation of Example 3.5. Aggregates represent a good example of such deterministic queries [159]. For example, the following program computes the summation aggregate over a relation \texttt{r}, using an arbitrary ordering of \texttt{r} computed by \texttt{ord\_r} (as defined in sect. 3.2.1):
\[ \text{sum}_{r}(\text{root}, 0). \]
\[ \text{sum}_{r}(Y, N) \leftarrow \text{sum}_{r}(X, M), \text{ord}_{r}(X, Y), N = M + Y. \]
\[ \text{total}_{\text{sum}}_{r}(N) \leftarrow \text{sum}_{r}(X, N), \neg \text{ord}_{r}(X, _). \]

Here, \( \text{sum}_{r}(X, N) \) is used to accumulate in \( N \) the summation up to \( X \), with respect to the order given by \( \text{ord}_{r} \). Therefore, the total sum is reconstructed from \( \text{sum}_{r}(X, N) \) when \( X \) is the last tuple in the order. Notice the use of (stratified) negation to the purpose of selecting the last tuple.

The query \( \text{total}_{\text{sum}}_{r}(N) \) is deterministic, in the sense that the value of the summation is independent from the particular order adopted, as \( + \) is commutative and associative. Notice that the hypothesis to have such a built-in operator is obviously an extension of Datalog, whose semantics is in this moment not relevant for the purpose of the presentation. Indeed, we might adopt the very unpractical hypothesis to base it on a finite extensional definition of a domain of integers.

### 3.3.1 User-Defined Aggregates

A general framework for dealing with aggregates in Datalog++ is the definition of a “sort” of syntactic sugar for aggregation [157]. For example, the following rule

\[ \text{total}_{\text{sum}}_{r}(\text{sum}(X)) \leftarrow r(X). \]

is used as an abbreviation of the above program. In order to compute the following aggregation predicate

\[ q(Y, \text{aggr}(X)) \leftarrow p(X, Y). \]

we exploit the capability of imposing a nondeterministic order among the tuples of the relation \( p \),

\[ \text{ord}_{p}(Y, \text{nil}, \text{nil}) \leftarrow p(X, Y). \]
\[ \text{ord}_{p}(Z, X, Y) \leftarrow \text{ord}_{p}(Z, _, X), p(Y, Z), \text{choice}(X, Y), \text{choice}(Y, X). \]

As shown in [157], we can then exploit such an ordering to define “recursive” aggregates, i.e., aggregates inductively defined:

\[ f(\{x\}) = g(x) \] (3.2)
\[ f(S \cup \{x\}) = h(f(S), x) \] (3.3)

We can directly specify the base and inductive cases, by means of ad-hoc user-defined predicates single and multi. In particular, single(\( \text{aggr}, X, C \)) associates to the first tuple \( X \) in the nonterministic ordering a value, according to 3.2, and multi(\( \text{aggr}, \text{old}, X, \text{new} \)) computes the value of the aggregate \( \text{aggr} \) associated to the current value \( X \) in the current ordering, by incrementally computing it from the
previous value, according to 3.3. The exploitation of such predicates is specified by the following recursive rules:

\[
\text{aggrP}(\text{Aggr}, Z, X, C) \leftarrow \text{ord}_p(Z, \text{nil}, X), X \neq \text{nil}, \text{single}(\text{Aggr}, X, C).
\]

\[
\text{aggrP}(\text{Aggr}, Z, Y, C) \leftarrow \text{ord}_p(Z, X, Y), \text{aggrP}(\text{Aggr}, X, C), \text{multi}(\text{Aggr}, C_i, Y, C).
\]

Finally, the originary rule can be translated into

\[
q(Y, C) \leftarrow \text{ord}_p(Y, X), \neg \text{ord}_p(Y, X, \text{aggrP}(\text{aggr}, Y, X, C)).
\]

**Example 3.6 ([159]).** The aggregate \text{sum} can be easily defined by means of the following rules:

\[
\text{single}(\text{sum}, X, X).
\]

\[
\text{multi}(\text{sum}, S, X, S N) \leftarrow S N = S + X.
\]

\]

In [159], a further extension to the approach is proposed, in order to deal with more complex aggregation functions. Practically, we can manipulate the results of the aggregation function by means of two predicates \text{freturn} and \text{ereturn}. The rule defining the aggregation predicate is translated into the following:

\[
q(Z, R) \leftarrow \text{ord}_p(Z, X, Y), \text{aggrP}(\text{aggr}, Z, X, C), \text{ereturn}(\text{aggr}, C, Y, R).
\]

\[
q(Z, R) \leftarrow \text{ord}_p(Z, X, Y), \neg \text{ord}_p(Z, Y, \_), \text{aggrP}(\text{aggr}, Z, Y, C), \text{freturn}(\text{aggr}, C, R).
\]

where the first rule defines early returns (i.e., results of intermediate computations), and the second rule defines final returns, i.e., results on overall values.

**Example 3.7.** The sample variance of a set of values is defined as \( s = \frac{1}{n} \left( \sum_i x_i^2 - \frac{1}{n} (\sum_j x_j)^2 \right) \). The aggregate that computes such a measure over a column of a table is the following:

\[
\text{single}(\text{var}, X, (X, X \times X, 1)).
\]

\[
\text{multi}(\text{var}, (S, SS, N), X, (S + X, SS + X \times X, N + 1)).
\]

\[
\text{freturn}(\text{var}, (C, S, N), 1/(N - 1) \times (S - 1/N \times C \times C)).
\]

\]

**Example 3.8 ([159]).** The aggregate \text{argmax} considers tuples \((c_i, n_i)\), where \( n_i \) is a real number, and returns the term \( c_i \) with the greatest value of \( n_i \). The aggregate can be defined by means of \text{single}, \text{multi} and \text{freturn}:

\[
\text{single}((\text{argmax}, (C, P), (C, P)).
\]

\[
\text{multi}((\text{argmax}, (C, PO), (C, P), (C, P)) \leftarrow P \geq PO.
\]

\[
\text{multi}((\text{argmax}, (C, PO), (C, P), (C, PO)) \leftarrow P < PO.
\]

\[
\text{freturn}((\text{argmax}, (C, PO), CO)).
\]
Example 3.9. Given the relation gate(G,X) specifying the output signal of a gate G, the andGate(Y) predicate should compute the intersection of the signals of all the available gates. The computation can be specified by means of an aggregate and:

\[
\text{andGate}(\text{and}(X)) \leftarrow \text{gate}(G, X).
\]

\[
\text{single}(\text{and}, X, X).
\]

\[
\text{multi}(\text{and}, A, X, A) \leftarrow X \neq 0, A \neq 0.
\]

\[
\text{ereturn}(\text{and}, A, 0, 0).
\]

\[
\text{return}(\text{and}, X, X).
\]

Notice that the ereturn predicate allows to stop the computation as soon as a 0 gate is found.

On the basis of these simple examples, more sophisticated forms of aggregation, such as datacube and other OLAP functions, can be built [72, 28]. As an example, consider a relation sales(Date, Department, Sale), and the problem of aggregating sales along the dimensions Date and Department. Three aggregation patterns are then possible, corresponding to the various facets of the datacube: \langle Date, \ast \rangle, \langle \ast, \text{Department} \rangle, \langle \ast, \ast \rangle. The former two patterns correspond to the aggregation of sales along a single dimension (respectively Department and Date), and can be obtained from the original relation by applying the method shown above. The latter pattern, then, can be obtained by recursively applying such method to one of the two patterns previously computed, in order to aggregate along the remaining dimension. In general, aggregations along \( n \) dimensions can be computed in many ways, starting from any aggregation along \( m \) dimensions, with \( m < n \). A careful choice of a plan of aggregation can minimize the number of required operations to compute the whole datacube. The following program, which combines aggregation and recursion, implements the above idea.

\[
\text{cuboid}(0, \text{Date}, \text{Department}, \text{Sale})) \leftarrow \text{sales}(\text{Date}, \text{Department}, \text{Sale}).
\]

\[
\text{cuboid}(I + 1, \text{Dim}'1, \text{Dim}'2, \text{sum}(S)) \leftarrow \text{cuboid}(I, \text{Dim}1, \text{Dim}2, S),
\]

\[
\text{path}(\text{Date}, \text{Department}, \text{Date}, \ast) \leftarrow \text{Department} \neq \ast, \text{Date} \neq \ast.
\]

\[
\text{path}(\text{Date}, \text{Department}, \ast, \text{Department}) \leftarrow \text{Department} \neq \ast, \text{Date} \neq \ast.
\]

\[
\text{path}(\ast, \text{Department}, \ast, \ast) \leftarrow \text{Department} \neq \ast.
\]

\[
\text{cube}(\text{Dim}1, \text{Dim}2, S) \leftarrow \text{cuboid}(-, \text{Dim}1, \text{Dim}2, S).
\]

Here, relation path specifies the execution plan for computing aggregations along \( I \) dimensions starting from a specific aggregation along \( I - 1 \) dimensions. In the example, we decided that the aggregation \langle \ast, \ast \rangle should be computed starting from \langle \ast, \text{Department} \rangle, as presumably there are a few departments and many dates. Using complex objects, such as lists, the datacube example can be generalized to an arbitrary number of dimensions.
3.4 Iterative User-Defined Aggregates

The main deficiency of the traditional user-defined aggregate model, specified in the previous section, is the impossibility of defining more complex forms of aggregates than distributive ones, within Datalog++. In many cases, even simple aggregates may require multiple steps over data in order to be computed.

Example 3.10. The absolute deviation $S_n = \sum_x |\bar{x} - x|$ of a set of $n$ elements is defined as the sum of the absolute difference of each element with the mean value $\bar{x} = 1/n \sum x$ of the set. In order to compute such an aggregate, we need to scan the available data twice: first, to compute the mean value, and second, to compute the sum of the absolute difference. $\triangleright$

Example 3.11. The median of a set of elements is defined as the element in the mid position of the sorted order of the elements. In order to compute the median as an aggregate, we need to sort the elements, and sorting require (in general) multiple scans. $\triangleright$

A simple way [52] of coping with the problem of multiple scans over data can be done by extending the declarative rewriting of the aggregation rule

$$q(\text{aggr}(X)) \leftarrow p(X)$$

(3.4)

in order to impose some user-defined conditions for iterating the scan over data. More specifically, the extension we propose can be specified as an iterative function $\text{aggr}$ over a set $S$:

```plaintext
aggr(S)
{
    init(res);
    do
        res = f(res, S)
    while (iterate(res));
    return res;
}
```

where $f$ is defined, as usual, as a fold operator

$$f(v, \{\}) = g(v)$$
$$f(v, S \cup \{x\}) = h(f(v, S), x)$$

and $\text{iterate}$ is a user-defined condition specifying whether the aggregate computation is completed. By exploiting recursion, it is easy to define such an iteration directly as an aggregate computation:
\[
cagr_p(nil, Aggr, nil, New) \leftarrow \text{empty}(Aggr, New).
\]
\[
cagr_p(nil, Aggr, X, New) \leftarrow \text{ord}_p(nil, X), X \neq \text{nil},
\]
\[
\text{single}(Aggr, X, New),
\]
\[
\neg \text{empty}(Aggr, \_).
\]
\[
cagr_p(I, Aggr, Y2, New) \leftarrow \text{ord}_p(Y1, Y2), Y2 \neq \text{nil},
\]
\[
cagr_p(I, Aggr, Y1, Old),
\]
\[
\text{multi}(Aggr, Old, Y2, New).
\]
\[
cagr_p(s(I), Aggr, nil, New) \leftarrow \text{ord}_p(X, Y), \neg \text{ord}_p(Y, \_),
\]
\[
cagr_p(I, Aggr, Y, Old),
\]
\[
\text{iterate}(Aggr, Old, New).
\]

The main difference with respect to the schema shown in section 3.3 is that the predicate \( cagr_p \) contains a stage argument, counting the iterations over the tuples. The last rule specifies the condition for iterating the aggregate computation: the activation (and evaluation) of such a rule is subject to the successful evaluation of the user-defined predicate iterate, so that any failure in evaluating it results in the termination of the computation. This guarantees that the proposed schema is conservative: any aggregate specified by single, multi and return is computed in the same way as shown in the previous section.

Rule 3.4 is then rewritten in the following rules,
\[
q(S) \leftarrow \text{ord}_p(X, Y), \neg \text{ord}_p(Y, \_), cagr_p(I, aggr, Y, C),
\]
\[
\neg cagr_p(s(I), aggr, nil, C), \text{freturn}(aggr, C, S).
\]
\[
q(S) \leftarrow \text{ord}_p(X, Y), Y \neq \text{nil}, cagr_p(I, aggr, X, C),
\]
\[
\text{iterate}(aggr, C, Y, S).
\]

where the aggregate produces a final result only when iteration stops.

**Example 3.12.** By adopting the above schema, the absolute deviation \( S_n \) can be defined in the following way:
\[
\text{single(abserr, } X, (\text{nil, } X, 1)).
\]
\[
\text{multi(abserr, (nil, } S, C), X, (\text{nil, } S + X, C + 1)).
\]
\[
\text{multi(abserr, (M, } D), X, (M, D + (M - X))) \leftarrow M > X.
\]
\[
\text{multi(abserr, (M, } D), X, (M, D + (X - M))) \leftarrow M \leq X.
\]
\[
\text{iterate(abserr, (nil, } S, C), (S/C, 0)).
\]
\[
\text{freturn(abserr, (M, } D), D).
\]

Notice how the combined use of multi and iterate allows to define two scans over the data: the first scan in order to compute the mean value, and the second one in order to compute the sum of the absolute difference with the mean value.
3.4. ITERATIVE USER-DEFINED AGGREGATES

Example 3.13. A simple naive computation of the median value can be obtained by computing, for each element, both the number of elements that precede and follow it\(^3\).

\[
\text{single(median, } X, (\text{nil, X, nil, 0, 0})).
\]
\[
\text{multi(median, (} \text{L, T, G, N, M}) X, (\text{L, T, X, N, M + 1})). \leftarrow T < X, X \leq G.
\]
\[
\text{multi(median, (} \text{L, T, G, N, M}) X, (\text{L, T, G, N + 1, M})). \leftarrow T > X, X \leq X.
\]
\[
\text{multi(median, (} \text{L, T, G, N, M}) X, (\text{L, T, G, N + 1, M})). \leftarrow T > X, L > X.
\]
\[
\text{multi(median, (} \text{L, T, G, N, M}) X, (\text{X, T, G, N + 1, M})). \leftarrow T > X, L \leq X.
\]
\[
\text{multi(median, (} \text{L, T, G, N, M}) X, (\text{L, X, G, N, M}))
\]

The tuple \((\text{L, T, G, N, M})\) defines for each candidate \(T\) the number of its predecessors \(N\), the number of its successors \(M\), its immediate predecessor \(L\) and its immediate successor \(G\). We then iterate the computation over each element, until we find an element \(x\) such that, given \(G_x = \{y | x < y\}\) and \(L_x = \{y | x > y\}\), either \(|G_x| = |L_x|\) where \(|G_x| = |L_x| + 1\) or \(|G_x| + 1 = |L_x|\).

\[
\text{iterate(median, (T, G, N, M), (} \text{L, G, nil, nil, 0, 0})). \leftarrow N + 1 < M.
\]
\[
\text{iterate(median, (T, G, N, M), (} \text{nil, nil, L, nil, 0, 0})). \leftarrow N > M + 1.
\]
\[
\text{freturn(median, (T, G, N, M), T)}.
\]
\[
\text{freturn(median, (T, G, N, M), (} \text{T, L} + \text{L}/2)). \leftarrow N = M + 1.
\]
\[
\text{freturn(median, (T, G, N, M), (} \text{T, G} + \text{G}/2)). \leftarrow M = N + 1.
\]

The above approach is deterministic, in the sense that each iteration computes the rank of a single element. A simplest approach can be however defined by “non-deterministically” counting the rank of each element in one step. Such an approach requires two iterations: in the first iteration, elements are collected; the second iteration computes the rank of each element. For example, supposing, for simplicity, that there is an odd number of elements, we can provide the following definition:

\[
\text{single(median, X, (X, 1))}.
\]
\[
\text{multi(median, (X, N), (} \text{X, N + 1)).}
\]
\[
\text{multi(median, (} \text{X, N}) X, (\text{X, N + 1}).
\]
\[
\text{iterate(median, (X, N), (0, X, 0, N))}.
\]
\[
\text{multi(median, (L, X, G, N), Y, (} \text{L + 1, X, G, N}). \leftarrow Y < X.
\]
\[
\text{multi(median, (L, X, G, N), Y, (} \text{L, X, G + 1, N}). \leftarrow Y > X.
\]
\[
\text{multi(median, (L, X, G, N), X, (} \text{L, X, G, N})
\]
\[
\text{freturn(median, (N/2, X, N/2, N))}.
\]

The first three rules simply collect all the distinct elements, and count the total number of their occurrences. The second iteration, specified by the \text{iterate} and \text{multi} rules, count the rank of each element. As a final result, the element in the

\(^{3}\text{for simplicity we assume that nil < X for each X.}\)
middle is returned as an answer. Notice that the two approaches are equivalent in efficiency, but the second approach “nondeterministically” computes the rank of each element in one iteration.

Example 3.14. Aggregates involving recursive computations (as defined, for example, in [147]) are easily definable by using the iterative method described above. For example, suppose you want to define the aggregate ancestors that, given the well-known relation parent(Parent, Son), computes all ancestors of a given parent.

\[
\begin{align*}
\text{single}(\text{anc}, (P, C, F), (F, \text{nil})). \\
\text{single}(\text{anc}, (P, C, C), (P, C)). \\
\text{multi}(\text{anc}, (C, _), (P, C, F), (P, C)). \\
\text{multi}(\text{anc}, (P1, C1), (P, C, F), (P1, C1)). \\
\text{iterate}(\text{anc}, (P, C), (P, \text{nil})) \leftarrow C \neq \text{nil}. \\
\text{return}(\text{anc}, (P1, C1), (P, C, F), P1) \leftarrow C1 \neq \text{nil}.
\end{align*}
\]

The above aggregate can hence be used in a rule, as usual. For example, the following rule

\[
\text{ancestors(anc}((P, C, \text{tom}))) \leftarrow \text{parent}(P, C).
\]

computes all the ancestors of \text{tom}.

\[\square\]
Chapter 4

Towards A Logic-Based Knowledge Discovery Support Environment

Abstract

Although it may at first sound appealing to have an autonomous data mining system, it is practically unfeasible to let the data mining algorithm “run loose” into the data in the hope to find some valuable knowledge. Blind search into a database can easily bring to the discovery of an overwhelming large set of patterns, many of which could be irrelevant, difficult to understand, or simply not valid: in one word, uninteresting. This chapter defines the notion of a knowledge discovery support environment: an integrated formalism capable of

• rigorous definition of user interaction during the search process,

• separation of concerns between the specification and the mapping to the underlying databases and data mining tools, and

• understandable representations for the knowledge.

The idea of a knowledge discovery support environment has attracted increasing attention, as a formalism capable supporting the user in defining the evaluation measures of interest, combining multiple learning strategies and supporting the user in the specification of the necessary abstraction mechanism: in one word, to increase programmer productivity of KDD applications. However, such capabilities require higher-order expressive features capable of providing a tight-coupling between knowledge mining and reasoning. Logic database languages seem to be a good starting point for their capability to express complex queries [89]. Consequently, the purpose of this chapter is to justify their use, thus defining the notion of logic-based knowledge discovery support environment.
4.1 The Role of Domain Knowledge

There are two main drawbacks with purely data driven approaches. First, they perform poorly when insufficient data is available. Second, when large training data sets are available they tend to generate many uninteresting patterns from data sets, and usually it is left to the domain expert to distinguish the "useful" pieces of information from the rest. The size of this problem (a data mining issue per se) suggests the need to drive the search of the to relevant sub spaces within the space of all possible hypotheses. The most straightforward form of guidance is the exploitation of domain, or prior knowledge.

The use of domain knowledge is necessary when the features under examination do not capture accurate abstractions, both from a quantitative and a qualitative point of view. Traditional data analysis techniques are primarily oriented toward the extraction of quantitative and statistical data characteristics, and, as such, have inherent limitations. For example, a statistical analysis can determine covariances and correlations between variables in data. It cannot, however, characterize the dependencies at an abstract, conceptual level, and produce a causal explanation of why these dependencies exist. Nor it can develop a justification of these relationships in the form of higher-level logic-style descriptions and laws. Attributes that define the similarity, as well as the similarity measures, must be defined by a data analyst in advance. Also, these techniques cannot by themselves exploit background domain knowledge in order to automatically generate relevant attributes and determine how their relevance changes with respect to different data analysis problems.

Notably, in the evaluation phase we need to associate some domain related quality function with any inferred knowledge. The notion of quality strictly pertains to the business decision process. However, while it is possible to define quantitative measures for uncertainty (e.g., estimated prediction accuracy on new data) or utility (e.g., gain, speed-up, etc.), notions such as novelty and understandability are much more subjective to the task, and hence difficult to define.

To address the above challenges, a data mining system has to be equipped with a substantial amount of background knowledge, and it must be able to perform symbolic reasoning tasks involving that exploits such a knowledge.

Example 4.1. Let us consider the following facts, describing the behavior of some people during a party.

\[
\begin{align*}
\text{drinks(john,wine).} & \quad \text{gets\_drunk(john).} \\
\text{drinks(hugh,whiskey)} & \quad \text{gets\_drunk(hugh).} \\
\text{drinks(bob,tequila).} & \quad \text{gets\_drunk(bob).} \\
\text{drinks(andrew,water).} & \quad \text{not\_enjoying(andrew).} \\
\text{drinks(tom,coke).} & \quad \text{not\_drunk(tom).} \\
\text{drinks(tom,rhum).} & \quad \text{gets\_drunk(tom).} \\
\text{drinks(alfred,tonic).} & \quad \text{gets\_drunk(alfred).} \\
\text{drinks(alfred,gin).} & \quad \text{not\_drunk(alfred).}
\end{align*}
\]
A concept learning system willing to learn the concept $\text{gets} \_\text{drunk}$ against such a database, will be trained to learn the misleading concept (from a common-sense point of view) $\text{drinks}(X,Y) \rightarrow \text{gets} \_\text{drunk}(X)$. However, the simple background knowledge about alcoholic beverages

$$\text{alcoholic}(\text{wine}). \quad \text{alcoholic}(\text{rhum}).$$

$$\text{alcoholic}(\text{whiskey}). \quad \text{alcoholic}(\text{gin}).$$

$$\text{alcoholic}(\text{tequila}).$$

can help to learn the more significant (and actually correct) concept $\text{drinks}(X,Y) \land \text{alcoholic}(Y) \rightarrow \text{gets} \_\text{drunk}(X)$. 

Prior domain knowledge can be used in different ways to constrain the search space and drive the learner to visit only the relevant sub spaces. Mitchell [111] characterizes most learning methods as search algorithms by describing the hypothesis space $H$ they search, the initial hypothesis $h_0$ with which they begin their search, the set of operators $O$ that define individual search steps, and the goal criterion $G$ that specifies the search objective. Domain knowledge can be incorporated into the learning process in three different ways: to derive initial hypothesis from which to begin the search, to alter the objective function of the hypothesis space search, and to alter/constrain the available search steps.

### 4.2 Query Languages for Data Mining

Tailoring data mining techniques to specific classes of applications, requires a rigorous user interaction during the search process, in order to facilitate efficient and fruitful knowledge manipulation and discovery. Such a rigorous interaction can be achieved by means of a set of data mining primitives, that should include the specification of

1. the source data
2. the kind of knowledge to be mined
3. background knowledge
4. interestingness measures for patterns evaluation
5. the representation of the extracted knowledge

Providing a query language capable to incorporate all these features may result, like in the case of relational databases, in a high degree of expressiveness in the specification of data mining tasks, a clear and well-defined separation of concerns between logical specification and physical implementation of data mining tasks, and easy integration with heterogeneous information sources.
There are many proposals for query languages for data mining, and this section is devoted to their analysis with respect to the properties described above. The integrated approaches that we shall consider can be grouped in two main categories:

- relational-based approaches, aimed at extending the current relational technology (query languages and system architectures) in order to provide a relational support for data mining applications.
- logic-based approaches, in which the main effort is devoted in providing intelligent logic-based interfaces to data mining algorithms.

4.2.1 Relational Extensions

The efforts for the development of mining tools tightly integrated with SQL DBMS, aim at the representation of both the source data and the extracted rules as database relations. In this context, the description of mining requests is performed by means of a SQL-like language that allows a flexible specification of mining statements. A major problem with such approaches is the limited expressive power of the underlying relational model. Defining complex relations among entities may require sets of logical expressions, containing advanced forms of negation and recursion. Clearly, such a limitation is particularly important when the aim is to extract high-level knowledge and represent domain knowledge.

The query language proposed in [123, 109] extends SQL with the new operator MINE RULE, which allows the computation and coding of associations in a relational format. Let us consider the relation transaction(Date,CustID,Item,Value) that contains the transactions of a sales representative. The following rule allows the extraction of the rules with support 20% and confidence 50%:

MINE RULE Associations AS
SELECT DISTINCT 1..n Item AS BODY, 1..1 Item AS HEAD,
SUPPORT,CONFIDENCE
WHERE BODY.Value > 100 AND HEAD.Value > 100
FROM transaction
GROUP BY CustID
HAVING COUNT(Item) > 4
CLUSTER BY Date
HAVING BODY.Date < HEAD.Date
EXTRACTING RULES WITH SUPPORT: 0.2, CONFIDENCE: 0.5

The above expression specifies the mining of associations of purchased items such that the right part of the rule (consisting of only 1 item) has been purchased after the left part of the rule (that can consist of more than one item), and related to those customers who bought more than 4 items. Moreover, we consider only items with a value greater than 100.
The above approach reflects the following features, according to the characterization depicted above:

- The source data is specified as a relational entity, and data preparation is accomplished by means of the usual relational operators. For example, the source table can be specified by means of usual join operations, selections and projections.

- The extended query language allows mining of unidimensional association rules. The GROUP BY keyword allows the specification of the transaction identifier, while the item description is specified in the SELECT part of the operator.

- Limited forms of background knowledge can be specified, by imposing some conditions over the admitted values of BODY and HEAD, and by using multiple source tables. Notice, however, that relying directly on SQL does not allow direct specification of more expressive constructs, such as, e.g., concept hierarchies. A limited form of data reorganization is specified by the CLUSTER keyword, that allows the specification of topology constraints (i.e. membership constraints of the components of rules to clusters).

- Concerning interestingness measures, the above operator allows the specification of the usual support and confidence constraints, and of further constraints over the contents of the rules (in particular, the SELECT keyword allows the specification of cardinality constraints).

- extracted knowledge is represented by means of relational tables, containing the specification of four attributes: Body, head, Support, Confidence.

Imielinski and Virmani [90, 91] put in evidence the importance of fully integrating the results of the mining queries in a relational environment. The MSQML proposal defines an extension to the SQL language to achieve closure between extracted knowledge and deductive knowledge.

\[
\text{SELECT *} \\
\text{FROM transaction} \\
\text{WHERE VIOLATES ALL (} \\
\text{GETRULES( transaction ))} \\
\text{WHERE BODY IS \{ ( Item = *)\} } \\
\text{AND CONSEQUENT IS \{ ( Item = *)\} } \\
\text{AND CONFIDENCE > 0.5 )}
\]

The above query allows the specification of the portion of the data that violates the given rules (i.e., the transactions that have the left part, but not the right part of the rule).

On-Line Analytical Mining [34, 76, 75, 134] is an approach that integrates on-line analytical processing with data mining knowledge in multi-dimensional databases.
DMQL [78, 80, 118, 79] is the corresponding querying paradigm. The language shows many similarities with the approaches shown above, denoting however the following differences:

- its source data is a data cube (and hence it is specified by means of a set of dimensions and measures)
- it allows the specification of concept hierarchies
- the result is specified as a "conceptual" cube, with patterns as dimensions and quality measures as measures.

As an example the following query specifies a classification task against a star schema

```sql
MINE DATABASE Credit_Ranks
USE HIERARCHY location_hierarchy FOR C.Address
MINE CLASSIFICATION AS Credit_Risk
ANALYZE Risk
IN RELEVANCE TO C.Address, C.Age, P.type
FROM Client C, account I, account_types P
WHERE C.ID = I.client_ID AND P.ID = I.account_ID
WITH ERROR ≤ 5%
DISPLAY AS TABLE
```

It is out of the scope of this chapter to investigate the implications of the conceptual and logical level over the physical layer. However, it is worth noting how the above approach requires, differently from the others approaches, ad-hoc structures for efficient data cube operations, as well as specialized mining algorithms working on data cubes. Such architectural choices are based on the observation that different memory organizations and indexes can provide a better architectural support for Data Mining. A similar approach has been also defined in [85, 99, 134, 34], where the storage management of the DBMS is defined in way that allows the optimization of the typical OLAP operation and Data Mining access operations.

### 4.2.2 Deductive Databases

The query flock's proposal in [144] shows how the combination of association rules with the deductive capabilities of a logic query language yields a higher degree of expressiveness and flexibility, which is crucial in addressing the above specified problems. The approach proposes the specification of the mining task by means of logical clauses. For example, the problem of mining frequent itemsets of size 2 can be specified by means of the following program:

```
pair(I_1, I_2, count(T)) ← basket(T, I_1), basket(T, I_2), I_1 < I_2.
rules(I_1, I_2) ← pair(I_1, I_2, C), C ≥ 2.
```
4.2. QUERY LANGUAGES FOR DATA MINING

The above specification has the main advantage of allowing a complete integration of source knowledge, background knowledge and extracted knowledge. However, the authors put in evidence how the above "natural" specification must be refined with suitable execution strategies. For example, using the \textit{apriori} property [80],

$$\text{if } \text{count}(S, D) > T \text{ then for each } S' \subseteq S, \text{count}(S', D) > T$$

(where \text{count}(S, D) is the number of occurrences of itemset \(S\) in dataset \(D\)), the above program may be transformed into an equivalent but extremely more efficient one:

\[
\begin{align*}
\text{temp}(I, \text{count}(T)) & \leftarrow \text{basket}(T, I). \\
\text{filter}(I) & \leftarrow \text{temp}(I, C), C \geq 2. \\
\text{pair}(I_1, I_2, \text{count}(T)) & \leftarrow \text{filter}(I_1), \text{filter}(I_1), \\
& \quad \text{basket}(T, I_1), \text{basket}(T, I_2). \\
\text{rules}(I_1, I_2) & \leftarrow \text{pair}(I_1, I_2, C), C \geq 2.
\end{align*}
\]

So, in this approach the logic language is presented as specification formalism while the emphasis is given on the computational support, that has to be re-defined for each mining method on the basis of the properties of the problem to be analyzed.

The first proposal of using a deductive database language for explorative data analysis has been developed in [132, 133]. The main idea is that of defining meta-rules that describe general patterns. Intuitively, a meta-rule

$$P_1(X_1) \land P_2(X_2) \land \cdots \land P_n(X_n) \Rightarrow Q_1(Y_1) \land Q_2(Y_2) \land \cdots \land Q_m(Y_m)$$

where \(P_i, Q_j\) are predicate variables, and \(X_i, Y_j\) are (tuples of) term variables, represents a high-level concept to be investigated. By instantiating such concept with rules built on the relations of the databases, i.e., by instantiating the predicate variables and possibly some term variables, one can then verify which patterns hold in the database. For example, the meta-rule

$$P(X, Y) \land Q(X, Z) \Rightarrow R(X, W)$$

can be analyzed against the database

\[
\begin{align*}
\text{transaction}(\text{cust1}, \text{sugar}, 20). \\
\text{transaction}(\text{cust2}, \text{sugar}, 10). \\
\text{transaction}(\text{cust1}, \text{oranges}, 30). \\
\text{transaction}(\text{cust2}, \text{oranges}, 40). \\
\text{transaction}(\text{cust1}, \text{apples}, 20). \\
\text{purchase}(X, Y) & \leftarrow \text{basket}(X, Y, \_). \\
\text{avg}_1\_\text{value}(X, \text{avg}(A)) & \leftarrow \text{basket}(X, \_ A).
\end{align*}
\]
by instantiating all the predicate variables, and then executing the right-hand and the left-hand of the obtained instances in order to compare the results. This allows discovering, e.g., the rule

\[
purchase(X, \text{sugar}) \land purchase(X, \text{oranges}) \Rightarrow \text{avg\_value}(X, 20)
\]

which states that, if a purchaser includes in his basket oranges and sugar, then usually the average values of his purchased items is 20.

The most relevant features of the approach can be summarized as follows:

- The source data is represented by first order predicates defined either extensionally or intensionally. This is a major advantage w.r.t. the relational approaches, where the only available sources are extensional predicates.

- The approach concentrates on meta-rule mining, even if in [133] an extension of the approach to classification and clustering is shown. From a theoretical point of view, meta-rule mining is a more expressive extension of association rule mining, since it allows the specification of both unidimensional and multidimensional rules, as well as of different forms of associations (such as the one shown above). From a practical point of view, the main limitation is given by the rigid format of the meta-rules (so that it is difficult, for example, to specify rules of arbitrary length).

- Background knowledge can be easily provided by exploiting the capabilities of the logic database language representing the deductive engine (e.g., \textsc{CdlC++}). For example, it is easy to provide support for conceptual hierarchies.

- Quantitative interestingness measures need to be a priori defined. [132] provides as a quality measure the estimation of the probability of the rule. Notice, however, that some forms of constraints can be easily introduced by explicitly specifying them in the meta-queries. For example, we can instantiate both predicate and term variables, or impose some constraints over such variables.

- Extracted knowledge is represented by means of general clauses. Even if such a representation is quite similar to the representation of the source data and background knowledge, no mapping into the underlying knowledge base has been defined. Hence, there is no closure between the extracted knowledge and the mined knowledge, and the results of the mining phase need to be ad-hoc processed in order to be integrated into the process.

In this approach the logic language is used as a computational engine for subparts of the mining process. The right and left parts of the meta-rule are instantiated by generating all possible queries matching them. Such queries are solved within the deductive component, and the most significant answers (according to some interestingness measure) are chosen. Notice, however, that there's no real integration between deduced and induced knowledge, e.g., between the results of the analysis and user-defined knowledge.
4.2.3 Inductive Logic Programming

A more general framework in the direction of meta-rule guided mining is the Inductive Logic Programming approach [113]. This approach tries to overcome two main limitation of the classical machine learning and data mining approaches:

- the use of a limited knowledge representation formalism (e.g., attribute-value pairs in association rules mining)
- difficulties in using substantial background knowledge in the mining process.

Of particular interest is the technique proposed in [38, 37] that addresses a general Datalog formulation of the frequent pattern discovery problem. The approach can be characterized as follows:

- Source data is specified by logic programs, i.e., both extensional and intensional databases.
- The mining task is frequent query discovery, formulated as discovering frequent statements of the form:
  \[ \exists (Q_1 \land \ldots \land Q_n \Rightarrow Q_{n+1} \land \ldots \land Q_m) \]
  where each \( Q_i \) is a Datalog predicate. Mining such patterns means specifically to traverse the search space of the conjunctions of all instantiations of all predicates. In order to overcome to obvious efficiency problems that could arise, the specific patterns to be mined are formally specified in a way that constrains the search space. For example, in order to mine association rules, the following constraints can be specified:
  \[
  \begin{align*}
  &basketID(-t) \\
  &basket(+t, sugar), basket(+t, orange), basket(+t, apples)
  \end{align*}
  \]
  that enumerate the predicates of interest (basketID and basket), the free variables (the symbol \(-t\), specifying that a certain term \( t \) can be freely generated), and the bound variables (the symbol \(+t\), specifying that the term \( t \) must be the same term generated by the free variable in basketID\((-t)\)).

- The interestingness measure considered is the frequency of a query pattern, i.e., the number of answer substitutions that satisfy a query \( \exists (Q_1 \land \ldots \land Q_m) \) and its left subquery \( \exists (Q_1 \land \ldots \land Q_n) \).

- Since predicates can be defined by means of rules, background knowledge can be easily exploited both in the hypothesis formulation and in the search space specification. It is easy, for example, to specify hierarchies by means of ad-hoc defined is-a predicates. The extension of the search space can be suitably defined as shown above:
  \[
  \begin{align*}
  &basket(+t, -i) \\
  &is-a(+i, fruit), is-a(+i, diet, prod)
  \end{align*}
  \]
• Extracted knowledge is represented, like in the case of metapatterns, by means of logical formulas. For example, a frequent query related to the above specified search space is

$$\exists T A B. (\text{basketID}(T) \land \text{basket}(T, A) \land \text{is}_{\text{a}}(A, \text{fruit}) \Rightarrow \\
\text{basket}(T, B) \land \text{is}_{\text{a}}(B, \text{diet}\_\text{prods}))$$

Again, like in the case of metapatterns, such a formalization is feasible only for knowledge generation, and provides little support to postprocessing of extracted knowledge.

4.3 The Datalog++ Approach

The conceptual model that summarizes the relevant aspects discussed so far is the notion of inductive database [19, 105], that is a first attempt to formalize the notion of interactive mining process. In the following definition, proposed by Mannila, the term inductive database refers to a normal database plus the set of all sentences from a specified class of sentences that are true of the data.

**Definition 4.1** ([106, 105]). Given an instance \( r \) of a relation \( R \), a class \( L \) of sentences (patterns), and a selection predicate \( q \), a pattern discovery task is to find a theory

$$Th(L, r, q) = \{ s \in L | q(r, s) \text{ is true} \}$$

The main idea here is to provide a unified and transparent view of both inferred (deductive) knowledge, and all the derived patterns, (the induced knowledge) over the data. The user does not care about whether he is dealing with inferred or induced knowledge, and whether the requested knowledge is materialized or not. The only detail he is interested in is the high-level specification of the query involving both deductive and inductive knowledge, according to some (either objective or subjective) interestingness quality measure.

The above definition can be refined in order to allow the formulation of a query language for inductive databases.

**Definition 4.2** ([19]). An inductive database schema is a pair \( \mathcal{R} = (R, (Q_R, e, V)) \), where

- \( R \) is a relation schema
- \( Q_R \) is a collection of patterns
- \( V \) is a set of result values
4.3. THE DATALOG++ APPROACH

<table>
<thead>
<tr>
<th>$s_0$</th>
<th>$c(r_0).f$</th>
<th>$c(r_0).e$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \Rightarrow B$</td>
<td>0.25</td>
<td>0.33</td>
</tr>
<tr>
<td>$A \Rightarrow C$</td>
<td>0.30</td>
<td>0.66</td>
</tr>
<tr>
<td>$B \Rightarrow A$</td>
<td>0.25</td>
<td>0.50</td>
</tr>
<tr>
<td>$B \Rightarrow C$</td>
<td>0.50</td>
<td>1.00</td>
</tr>
<tr>
<td>$C \Rightarrow A$</td>
<td>0.50</td>
<td>0.66</td>
</tr>
<tr>
<td>$C \Rightarrow B$</td>
<td>0.50</td>
<td>0.66</td>
</tr>
<tr>
<td>$AC \Rightarrow B$</td>
<td>0.25</td>
<td>0.50</td>
</tr>
<tr>
<td>$BC \Rightarrow A$</td>
<td>0.25</td>
<td>0.50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$s_1$</th>
<th>$c(r_1).f$</th>
<th>$c(r_1).e$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \Rightarrow B$</td>
<td>0.33</td>
<td>0.33</td>
</tr>
<tr>
<td>$A \Rightarrow C$</td>
<td>0.66</td>
<td>0.66</td>
</tr>
<tr>
<td>$B \Rightarrow A$</td>
<td>0.33</td>
<td>1.00</td>
</tr>
<tr>
<td>$B \Rightarrow C$</td>
<td>0.33</td>
<td>1.00</td>
</tr>
<tr>
<td>$C \Rightarrow A$</td>
<td>0.66</td>
<td>1.00</td>
</tr>
<tr>
<td>$C \Rightarrow B$</td>
<td>0.33</td>
<td>0.50</td>
</tr>
<tr>
<td>$AC \Rightarrow B$</td>
<td>0.33</td>
<td>1.00</td>
</tr>
<tr>
<td>$BC \Rightarrow A$</td>
<td>0.33</td>
<td>1.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$s_2$</th>
<th>$c(r_1).f$</th>
<th>$c(r_1).e$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \Rightarrow C$</td>
<td>0.66</td>
<td>0.66</td>
</tr>
<tr>
<td>$B \Rightarrow C$</td>
<td>0.33</td>
<td>1.00</td>
</tr>
<tr>
<td>$AB \Rightarrow C$</td>
<td>0.33</td>
<td>1.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Figure 4.1: Components of an Inductive database instance.

- $e$ is the evaluation function that defines patterns semantics in terms of $\mathcal{V}$ (i.e., it maps an instance $r$ of $\mathcal{R}$ and a pattern $\theta \in \mathcal{Q}_R$ in $\mathcal{V}$).

An inductive database instance is a pair $(r, s)$, where $r$ instance of $\mathcal{R}$ and $s \subseteq \mathcal{Q}_R$. □

An algebra for the above schema defines more general queries than relational algebra. A typical inductive database query can operate on both the components of an inductive database, by querying both the deductive part $r$ and the inductive, virtual part, $s$ (assuming that $s$ is materialized as a table, and that the value $e(r, \theta)$ is available for each value $\theta \in s$). Query evaluation results in traditional relational evaluation for the part concerning $r$, and in execution of mining algorithms for the part concerning $s$.

Example 4.2 ([20]). Let $\mathcal{R} = A_1 \ldots A_n$ be a binary table. Let $f(W, r)$ denote the fraction of the tuples of $r$ having 1 in each attribute of $W \subseteq \mathcal{R}$. Let $c(A \Rightarrow B, x) = f(A \cup B, x) / f(A, x)$. We can then define $Q_{\mathcal{R}} = \{L \Rightarrow R | L, R \subseteq \mathcal{R}\}$, $\mathcal{V} = [0, 1]^2$ and $e(r, A \Rightarrow B) = (f(A \cup B, r), c(A \Rightarrow B, r))$. As an example, consider the relation $\mathcal{R} = ABC$, and the instance $x_0$ with extension shown in fig. 4.1. An example inductive instance is $idb_0 = (x_0, s_0)$, where $s_0$ is shown in fig. 4.1, and represents all the rules that can be generated from $x_0$. From the simple operation $\sigma_{A=1}(x_0) = r_1$, a corresponding operation $\sigma_{A=1}(idb_0) = idb_1$ can be defined, where $idb_1 = (r_1, s_1)$, and $s_1$ is the set of the rules holding over $r_1$. Notice that any change in the deductive part also reflects in the inductive part: for each rule, the evaluation measures are accordingly changed. Inductive operators can be defined as well. For example, the operator $\sigma_{C=x}(s_1)$ can be associate to a higher-level operator $\tau_{C=x}(idb_1) = idb_2$, where $idb_2 = (r_1, s_2)$ and $s_2$ is obtained from $s_1$ by selecting only those rules with $C$ as consequent. □

It is easy to see that definitions 4.1 and 4.2 are equivalent. Each Theory $\mathcal{T}h(\mathcal{L}, r, q)$ can be formalized by an inductive instance $(r, s)$, where $Q_{\mathcal{R}} = \mathcal{L}$, and $e = q$. For the converse, the following proposition states that each inductive instance is an inductive theory.
Proposition 1. For each instance \((r, s)\) of an inductive database schema \(R = (R, (Q_R, e, \mathcal{V}))\), there exist a set \(\mathcal{L}\) and a selection predicate \(q\) such that for each \(\theta \in s\) there exists \(\theta' \in Th(\mathcal{L}, r, q)\).

Proof (sketch). By construction. Let

- \(\mathcal{L} = \{\langle \theta, v \rangle | \theta \in Q_R, v \in \mathcal{V}\}, and\)
- \(q(r, \theta) = true\) if and only if \(\theta = \langle \theta_1, e(r, \theta_1) \rangle\) and \(\theta_1 \in s\).

It is easy to see that \(Th(\mathcal{L}, r, q)\) contains only patterns that are represented in \(s\). \(\square\)

The notion of Inductive Database fits naturally in rule-based languages, such as Deductive Databases [51, 53]. A deductive database can easily represent both extensional and intensional data, thus allowing a higher degree of expressiveness than traditional relational algebra. Such capability makes it viable for suitable representation of domain knowledge and support of the various steps of the KDD process.

The main problem in a deductive approach is how to choose a suitable representation formalism for the inductive part, enabling a tight integration with the deductive part. More specifically, the problem is how to formalize the specification of the set \(\mathcal{L}\) of patterns in such a way that each pattern \(s \in Th(\mathcal{L}, r, q)\) is represented as an independent (logical) entity (i.e., a predicate) and each manipulation of \(r\) results in a corresponding change in \(s\). To cope with such a problem, we introduce the notion of inductive rules, i.e., rules that formalize the dependency between the inductive and the deductive part of an inductive database.

Definition 4.3. Given an inductive database theory \(Th(\mathcal{L}, r, q)\), an inductive rule is a rule

\[
s : H \leftarrow B_1, \ldots, B_n
\]

such that

- The evaluation of \(B_1, \ldots, B_n\) in the iterated stable model \(M_{\lambda R}\) corresponds to the extension \(r\)
- there exist a mapping function \(\phi\) mapping each ground instance \(p\) of \(H\) in \(\mathcal{L}\)
- \(Th\) corresponds to the iterated stable model \(M_{\lambda R}\) i.e.,

\[
p \in M_{\lambda R} \text{ co-implies that } \phi(p) \in Th(\mathcal{L}, r, q)
\]

To summarize, we can formalize the notion of logic-based knowledge discovery support environment, as a deductive database programming language that models inductive rules as well as deductive rules.
4.3. THE DATALOG++ APPROACH

**Definition 4.4.** A logic-based knowledge discovery support environment is a deductive database language capable of specifying:

- relational extensions;
- intensional predicates, by means of deductive rules;
- inductive predicates, by means of inductive rules.

\[\square\]

### 4.3.1 Aggregates as Pervasive Concepts

When trying to formalize the notion of inductive rules, the first fact that is worth observing is that it is particularly easy to deal with data mining tasks in a deductive framework, if we use aggregates as a basic tool.

**Example 4.3.** The following program defines typical (two-dimensional) association rules by using the predefined count aggregate.

\[
\begin{align*}
\text{pair}(I_1, I_2, \text{count}(T)) & \leftarrow \text{basket}(T, I_1), \text{basket}(T, I_2), I_1 < I_2. \\
\text{rules}(I_1, I_2) & \leftarrow \text{pair}(I_1, I_2, c), c \geq 2.
\end{align*}
\]

The first rule generates and counts all the possible pairs, and the second one selects the pairs with sufficient support (i.e., at least 2). As a result, the predicate rules specifies associations, i.e. rules stating that certain combinations of values occur with other combinations of values with a certain frequency. Given the following definitions of basket,

\[
\begin{align*}
\text{basket}(1, \text{fish}). & \quad \text{basket}(2, \text{bread}). & \quad \text{basket}(3, \text{bread}). \\
\text{basket}(1, \text{bread}). & \quad \text{basket}(2, \text{milk}). & \quad \text{basket}(3, \text{orange}). \\
\text{basket}(2, \text{onions}). & \quad \text{basket}(3, \text{milk}). \\
\text{basket}(2, \text{fish}).
\end{align*}
\]

by querying \text{rules}(X, Y) we obtain predicates that model the corresponding inductive instances: \text{rules}(\text{milk, bread}), \text{rules}(\text{bread, milk}), \text{rules}(\text{fish, bread}) and \text{rules}(\text{bread, fish}).

\[\triangleleft\]

**Example 4.4.** The combined use of aggregates with rules makes it easy to define concepts of interestingness of the rules diverse from the usual statistical parameters. If we are interested in discovering the associations between the cities and the products where the sales decreased of more than 30% w.r.t. the average, we can define the following rules:

\[
\begin{align*}
\text{average} & \leftarrow \text{sales}(\text{City}, \text{Product}, \text{Date}, \text{Sales}). \\
\text{avg}_{\text{cp}}(\text{City}, \text{Product}, \text{avg}(\text{Sales})) & \leftarrow \text{sales}(\text{City}, \text{Product}, \text{Date}, \text{Sales}). \\
\text{answer}(\text{City}, \text{Product}) & \leftarrow \text{average}(\text{A}), \text{avg}_{\text{cp}}(\text{City}, \text{Product}, \text{P}), \text{P} \geq 0.70 \times \text{A}.
\end{align*}
\]
The first rule computes the average on the whole sales. The second rule computes the averages related to the tuples \((\text{City}, \text{Product})\), and the third rule selects the relevant rules.

The above examples show how the simple rules specifying aggregates can be devised as inductive rules. Rule containing aggregates satisfy the most important property of inductive rules, i.e., the capability to specify patterns of \(\mathcal{L}\) that hold in \(\mathcal{T}h\) in a “parameterized” way, i.e., according to the whole tuples of an extension \(r\). In this thesis, we use aggregates as the means to introduce mining primitives into the query language. An aggregate defines “naturally” an inductive database schema, where each pattern is directly defined by means of the true facts in the iterated stable model, and an evaluation function can be ad-hoc defined according to the interest measure given by the domain knowledge at hand.

**Lemma 1.** An aggregate defines an inductive database.

**Proof (sketch).** By construction. Let us consider the rule

\[ r_p : p(X_1, \ldots, X_n, \text{aggr}(Y_1, \ldots, Y_m)) \leftarrow r(X_1, \ldots, X_n, Y_1, \ldots, Y_m). \]

We then define

\[ \mathcal{L} = \{ \langle t_1, \ldots, t_n, s \rangle | p(t_1, \ldots, t_n, s) \text{ is ground} \} \]

and

\[ q(r, \langle t_1, \ldots, t_n, s \rangle) = \text{true} \text{ if and only if } p(t_1, \ldots, t_n, s) \in M_{r \cup r} \]

that imposes that the only valid patterns are those belonging to the iterated stable model procedure. \(\square\)

As a result, the following fact trivially holds.

**Corollary 1.** An aggregate rule is an inductive rule.

### 4.3.2 Logic-Based Inductive Databases

The above statements show how the Datalog++ framework, presented in chapter 3, can naturally deal with Inductive database schemas. The question that raises is whether it is capable of expressing all the inductive schemas, or at least the most significant. Practically, given an inductive database formalized by \(\mathcal{T}h(\mathcal{L}, r, q)\), the following general schema can be used to formalize the set of predicates \(q\) that populate \(\mathcal{T}h:\)

\[ q(Z_1, \ldots, Z_k, u \cup \text{aggr}(\langle X_1, \ldots, X_n \rangle)) \leftarrow r(Y_1, \ldots, Y_m). \]

(4.1)

Intuitively, this rule defines the format of any valid pattern \(s\) of \(\mathcal{L}\). The patterns in \(q\) are obtained from a rearranged subset \(X_1, \ldots, X_n\) of the tuples \(Y_1, \ldots, Y_m\) in \(r\).
The structure of $s$ is defined by the formal specification of the aggregate $agg$, in particular by the return rules.

The tuples $q(t_1, \ldots, t_k, s)$ resulting from the evaluation of such rule, represent patterns in $\mathcal{L}$ that actually are in $Th(\mathcal{L}, r, q)$. As a result, the “inductive” predicate $q$ itself can be used in the definition of more complex queries.\footnote{More precisely, the given schema produces different patterns on different clusters of the tuples of $r$ grouped on the basis of the values of the attributes $Z_1, \ldots, Z_k$. Practically, the results of the above schema, given an instance $r$ of $\mathcal{R}$, are the tuples $(\sigma_{C_1}(r), s_1), \ldots, (\sigma_{C_i}(r), s_k)$, where $C_i \equiv Z_1 = v_{i_1} \land \ldots \land Z_k = v_{i_k}$ with $v_{i_j} \in \text{dom}(Z_j)$, and $s_i$ is the set of patterns related to $\sigma_{C_i}(r)$.}

Example 4.5. Consider the relation $\text{transaction}(\text{Date}, \text{Cust}, \text{Item}, \text{Price}, \text{Qty})$. A sample mining scenario for such a table consists in detecting the items in the relation with the average value more than a given threshold. The inductive database has $\mathcal{R} \equiv \text{transaction}$, $Q_R = \{ i \mid i \in \text{dom}(\mathcal{R}[\text{Item}]) \}$ and $e(r, i) = \text{avg}(\{ p \times q \mid (t, i, p, q) \in r \})$. The above inductive schema is formalized, in accordance to (4.1) with the following rule:

$$s(\text{avgThres}((\sigma, \text{Item}, \text{Value}))) \leftarrow \text{transaction}(_{-} \_ \_ \text{Item}, \text{Price}, \text{Qty}),$$

$$\text{Value} = \text{Price} \times \text{Qty}.$$  

Where $\sigma$ represents the given threshold. The results of the evaluation of such a rule are represented by the predicates $s(i, v)$ such that $i \in \text{dom}(\mathcal{R}[\text{Item}])$ and $v \geq \sigma$ is the value of $e(r, i)$.

We can, finally, state the main point of this chapter, that formalizes the common interfaces for inductive and deductive reasoning in a suitable knowledge representation formalism.

Claim. An inductive database can be specified by means of a user-defined aggregate.

We will show how such a statement is verified for the data mining tasks whose patterns satisfy the closure property (i.e., they can be suitably represented by means of predicates in a logic-based environment).

Relating aggregate specification with inductive rules makes it easy to provide an interface capable of specifying source data, knowledge extraction, background knowledge and a limited form of interestingness specification. Moreover, even if the “inductive” user-defined aggregates can be simply seen as a black box (like in [65, 59]), we can specify the data mining task under consideration in detail, by exploiting ad-hoc definitions of single, multi, iterate and return iterative user-defined predicates. In fact, a direct specification of the mining algorithm within the deductive environment gives two main opportunities:

- from a conceptual point of view, it allows the use of background knowledge during the exploration of the search space, thus allowing the integration of more significant interest measures, both from a quantitative and a qualitative point of view.

---

\footnote{More precisely, the given schema produces different patterns on different clusters of the tuples of $r$ grouped on the basis of the values of the attributes $Z_1, \ldots, Z_k$. Practically, the results of the above schema, given an instance $r$ of $\mathcal{R}$, are the tuples $(\sigma_{C_1}(r), s_1), \ldots, (\sigma_{C_i}(r), s_k)$, where $C_i \equiv Z_1 = v_{i_1} \land \ldots \land Z_k = v_{i_k}$ with $v_{i_j} \in \text{dom}(Z_j)$, and $s_i$ is the set of patterns related to $\sigma_{C_i}(r)$.}
• from a physical point of view, the opportunity of directly integrating specific optimizations to the application of data mining algorithms, that can be treated no more as “black boxes”, but allow interaction in the various phases of the computational process.

The discussion on how to provide efficient implementations of the algorithms specified by means of user-defined aggregates is given in chapter 7. In section 3.4, instead, we introduced the iterative extension to the framework of user-defined aggregate that are particularly suitable for such an effective formalization of data mining algorithms.

Example 4.6. As a first example of how the above schema can be used to formalize the mining tasks specified as inductive rules, let us consider the inductive database of example 4.5. The aggregate avgThres can be defined by means of the predicates

\[
\begin{align*}
\text{single} & (\text{avgThres}, (T, I, V), (T, I)). \\
\text{multi} & (\text{avgThres}, (T, I), \neg (T, I)). \\
\text{multi} & (\text{avgThres}, (T, I, V), (T, I)). \\
\text{multi} & (\text{avgThres}, (T, I, V, N), (T, I, V, N)) \leftarrow V = V + V, N = N + 1. \\
\text{multi} & (\text{avgThres}, (T, I, V, N), (T, I, V, N)) \leftarrow I \neq I. \\
\text{iterate} & (\text{avgThres}, (T, I), (T, I, 0, 0)). \\
\text{freturn} & (\text{avgThres}, (T, I, V, N), (T, I)) \leftarrow A = V / N, A \geq T.
\end{align*}
\]

The first three rules collect all the possible items. For each item considered, both the sum of the associated values and the count of the occurrences are computed, and finally the average is computed and returned as answer, provided that is greater than a given threshold T.

\[
\downarrow
\]

4.4 Mining Aggregates

Although the notion of iterative aggregate is in some sense orthogonal to the notion of inductive rules defined in section 4.3, the main motivation for introducing iterative aggregates is that the above shown iterative schema is common in many data mining algorithms. Usually, a typical data mining algorithm is an instance of an iterative schema where, at each iteration, some statistics are gathered from the data. The termination condition can be used to determine whether the extracted statistics are sufficient to the purpose of the task (i.e., they determine all the patterns), or whether no further statistics can be extracted.

Example 4.7. The Apriori algorithm for the computation of frequent itemsets is based on the following criteria:
4.4. MINING AGGREGATES

1. At each iteration, frequencies of candidate itemsets are computed, and unfrequent candidate sets are removed. New candidate itemsets are computed by combining the frequent itemsets.

2. The cycle terminates when no further candidate itemsets are generated.

The two items identify the necessary conditions for specifying the computation of frequent itemsets by means of the iterative schema shown before.

Example 4.8. The K-means algorithm for the computation of clusters of tuples is defined by the following steps:

1. Initially, K random cluster centers are chosen.

2. At each iteration, each tuple is assigned to its most similar cluster center, according to some similarity measure. Hence, cluster centers are recomputed averaging the values of each cluster component.

3. No more iteration is performed if the cluster centers are stable, i.e., if they do not change.

Clearly, the specification of such operations within the iterative schema is immediate.

Example 4.9. A general schema for classification algorithms is the following [71]:

1. At each iteration, compute statistics over the attribute-set pairs

2. Iterate until no further useful statistics are needed

Such a recursive schema is common to many classification algorithms, such as decision-trees and Naive Bayes.

From these examples it is clear that the iterative schema shown in the previous section is a good candidate for specifying steps of data mining algorithms at low granularity levels. From an implementation viewpoint, the major drawback of using a purely declarative specification can be overcome using the techniques described in [61] to effectively implement such a specification. In practice, we adopt a compromise between loose and tight-coupling, by enhancing the terms of the practical language LDDL++ that implements the main features of Datalog++ with complex structures, as in the case of Object-Relational Systems. Ad-hoc user-defined predicates can hence be defined, to efficiently implement operations over such data structures. Such a capability, together with the possibility of directly specifying the mining algorithms in a deductive environment gives us the opportunity to apply both database optimizations and application/domain dependent optimizations. We can, for example, directly specify constraints, and apply optimized data structures whenever they are needed in order to speed-up the computation.
The rest of the thesis is devoted at showing how the model is suitable enough to cover some important tasks of the knowledge discovery process. We shall use the basic framework of this chapter, namely inductive rules (specified by means of iterative user-defined aggregates), to formalize the data mining task formulation in the Datalog++ framework. The integration of inductive rules with deductive rules allows the conceptual specification of the various steps of the data mining process:

- deductive rules can specify both the preprocessing and the result evaluation phase, while
- inductive rules can specify the mining phase.
Chapter 5

Frequent Patterns Discovery

Abstract

Discovery of frequent patterns in large data collection is an interesting challenge for a variety of reasons:

- The task is a very effective undirected knowledge discovery technique, that allows the discovery of interesting structures among the data;

- As explained in section 2.5.1, it is difficult to compute generalized patterns; hence, efficient algorithms, often based on heuristic methods, must be exploited;

- The quality of the results of mining has to be accurately evaluated, since the combinatorial nature of the problem may cause the mining of hundreds, even thousands, of results, many of which often uninterpretable, inaccurate or simply uninteresting. Hence, the challenge here is to specify refined quality measure. To this purpose, the role of background knowledge representation and ad-hoc result interpretation is perhaps more important than efficient computation.

This chapter shows how the iterative-user defined aggregate schema defined in the previous chapter offers a viable tool for specifying and implementing the task. We show frequent patterns can be represented and mined by means of inductive rules, and how the resulting inductive predicates can be used to the purpose of directly specifying complex mining objectives at higher abstraction levels. The idea of directly specifying the algorithm in a logic-based environment can raise obvious concerns about efficiency, for a mining task where the main problem that the current literature has been investigating efficient (even parallel) algorithms. We show, however, how such a concern can be easily resolved in practical systems, such as the CDL++ implementation of Datalog++.
5.1 Task Formulation

As shown in chapter 2, the problem of finding association rules can be decomposed in two subproblems: the problem of finding frequent itemsets and consequently the problem to find rules from frequent itemsets. Frequent itemsets are itemsets that appear in the database with a given frequency.

In chapter 2 we specified itemsets as sets of items of variable size, grouped according to some predefined key. In section 3.3 we have seen how the normalization and de-normalization of a table can be easily done in deductive languages with choice, such as Datalog++. In this and in the following chapters we shall refer to the language Datalog++ in a broader sense, i.e., including complex functors such as sets, tuples and lists. Some of these extension actually correspond to the practical LDC++ implementation, and reveal very useful for modeling purposes, as well as amenable to efficient implementations [33].

Example 5.1. Let us consider the example table of fig. 5.1 (a). The extension of the table is represented by extensional predicates of the form

\[
\text{transaction}(\text{Date}, \text{Cust}, \text{Item}, \text{Price}, \text{Qty})
\]  

(5.1)

The de-normalization of such a table is performed by means of the pseudo-aggregate rule

\[
\text{transaction}(D, C, (I)) \leftarrow \text{transaction}(D, C, I, P, Q)
\]

From a semantic viewpoint, such a rule is a syntactic sugar for the chained representation of a set of elements (as shown in section 3.3). From a practical point of view, the implementation of such a predicate can adopt specialized data structures, as it will be explained in greater details in chapter 7. In the LDC++ implementation, the above aggregate provides set structures: for example, grouping the items of transaction by date and customer, we obtain the relation of fig. 5.1 (b).

In definition 4.1 we have described the data mining task as a theory \( T_h \) obtained via a selection predicate \( q \). In the case of frequent patterns discovery, we are interested in itemsets that appear in the instance \( r \) with a given frequency. So, from a conceptual point of view, the theory \( T_h \) is represented by the itemsets that appear in a transaction table \( r \), and that satisfy a minimum support constraint. According to the inductive rules framework of the previous chapter, this theory can be represented by the set of results of an aggregation function.

Definition 5.1. Given a relation \( r \), the patterns aggregate is defined by the rule

\[
p(X_1, \ldots, X_n, \text{patterns} \langle \text{min\_supp}, Y \rangle) \leftarrow r(Z_1, \ldots, Z_m)
\]

(5.2)

where the variables \( X_1, \ldots, X_n, Y \) are a rearranged subset of the variables \( Z_1, \ldots, Z_k \) of \( r \), and the \( Y \) variable denotes a set of elements. The aggregate patterns computes the set of predicates \( p(t_1, \ldots, t_n, s, f) \) where:
Figure 5.1: (a) A sample transaction table. (b) One of its transactional representations, with related frequent patterns (c) and corresponding rules (d).

1. \( t_1, \ldots, t_n \) are distinct instances of the variables \( X_1, \ldots, X_n \), as resulting from the evaluation of \( r \);  
2. \( s = \{l_1, \ldots, l_k\} \) is a subset of the value of \( Y \) in a tuple resulting from the evaluation of \( r \);  
3. \( f \) is the support of the set \( s \), such that \( f \geq \text{min supp} \).

Example 5.2. Let us consider the tables defined in example 5.1. The following rule specifies the computation of 30%-frequent itemsets starting from the transactions defined above:

\[
\text{frequentPatterns}(\text{patterns}((0.3, S))) \leftarrow \text{transaction}(D, C, S). \tag{5.3}
\]

The evaluation of the above rule results in the tuples of figure 5.1 (c), and is computed from the combination of the rules 5.1 and 5.3, where the first rule collects
all the transactions (i.e., performs the data preparation phase), and second rule extracts the relevant patterns from the collection of available transaction.

Although the patterns aggregate only allows the discovery of frequent itemsets, rules can be easily generated from such patterns by means of simple deductive rules.

Example 5.3. The predicate rules computes the available rules from the frequent itemsets:

\[
\text{rules}(L,R,S,C) \leftarrow \text{frequentPatterns}(A,S), \text{frequentPatterns}(R,S_1),
\text{subset}(R,A), \text{difference}(A,R,L), C = S/S_1.
\]

The result of the evaluation of such a rule is shown in fig. 5.1 (d).

5.1.1 Specifying the patterns Iterative Aggregate

The above definition proposes the model of interaction, in a mining scenario, of deductive and inductive parts. However, the format of the above defined aggregate has to be specified by the iterative schema defined in the previous chapter.

The simplest definition adopts the purely declarative approach of generating all the possible itemsets, and then testing the frequency of the itemsets. It is easy to provide such a naive definition by means of the iterative schema proposed in section 3.4:

\[
\begin{align*}
\text{single(patterns,}(Sp,S),((Sp,1),IS)) & \leftarrow \text{subset}(IS,S). \\
\text{multi(patterns,}(Sp,N), (Sp,S), ((Sp,N + 1),IS)) & \leftarrow \text{subset}(IS,S). \\
\text{multi(patterns,}(Sp,N), IS), (Sp,IS)) & \leftarrow \text{subset}(IS,S). \\
\text{multi(patterns,}(Sp,IS,N), (Sp,S), (Sp,IS,N + 1)) & \leftarrow \text{subset}(IS,S). \\
\text{multi(patterns,}(Sp,IS,N), (Sp,S), (Sp,IS,N)) & \leftarrow \neg\text{subset}(IS,S). \\
\text{iterate(patterns,}(Sp,N), IS), (Sp \times N, IS,0)). \\
\text{freturn(patterns,}(Sp,IS,N), (IS,N)) & \leftarrow N \geq Sp.
\end{align*}
\]

Such an aggregate has two main iterations. In the first iteration (specified by the first three rules), for each tuple in the dataset the set of possible subsets are generated. The iterate predicate initializes the counter of each candidate itemset, and activates the computation of its frequency (performed by the remaining multi rules). The computation terminates when all itemsets frequencies have been computed, and frequent itemsets are returned as answers (by mean of the freturn rule). Notice that the freturn predicate defines the output format for the aggregation predicate: a suitable answer is a pair (Itemset,N) such that Itemset is an itemset of frequency \( N > Sp \), where Sp is the minimal support required.
5.1. TASK FORMULATION

Such a specification is clearly extremely inefficient, since it checks the support of all the possible itemsets, in a generate-and-test style. More precisely, the aggregate computation generates \(2^{|I|}\) sets of items, where \(I\) is the set of different items appearing in the tuples considered during the computation. The approach shows the following drawbacks:

- No pruning strategy is exploited by the aggregate computation: pruning of unfrequent subsets is made at the end of the computation of all subsets.

- No optimized data structure is used, that can speed-up the computation during the generation phase.

**Example 5.4.** A simple solution to the first problem can be obtained by computing the maximal frequent itemsets, and then by generating all their subsets starting from these subsets. We start by generating all possible maximal itemsets:

\[
\begin{align*}
\text{single}(\text{patterns}, (\text{Sp}, S), ((\text{Sp}, 1), S)). \\
\text{multi}(\text{patterns}, ((\text{Sp}, N), IS), \neg (\text{Sp}, IS)). \\
\text{multi}(\text{patterns}, ((\text{Sp}, N), \neg (\text{Sp}, S), ((\text{Sp}, N), S)).
\end{align*}
\]

The counting phase is quite similar to the above shown phase:

\[
\begin{align*}
\text{iterate}(\text{patterns}, ((\text{Sp}, N), IS), (\text{Sp} \times N, IS, 0)). \\
\text{multi}(\text{patterns}, (\text{Sp}, IS, N), (\text{Sp}, S), (\text{Sp}, IS, N + 1)) & \leftarrow \text{subset}(IS, S). \\
\text{multi}(\text{patterns}, (\text{Sp}, IS, N), (\text{Sp}, S), (\text{Sp}, IS, N)) & \leftarrow \neg \text{subset}(IS, S).
\end{align*}
\]

However, at the end of each counting phase, we heuristically decrement the size of unfrequent itemsets,

\[
\begin{align*}
\text{iterate}(\text{patterns}, (\text{Sp}, IS, N), (\text{Sp}, SS, 0)) & \leftarrow \text{Sp} > N, \text{member}(I, S), \\text{union}(I, SS, IS).
\end{align*}
\]

and mark frequent itemsets

\[
\begin{align*}
\text{iterate}(\text{patterns}, (\text{Sp}, IS, N), (\text{Sp}, \text{root}, IS, N)) & \leftarrow N \geq \text{Sp}.
\end{align*}
\]

frequent patterns are returned as answers as soon as they are recognized as “frequent”

\[
\begin{align*}
\text{ereturn}(\text{patterns}, (\text{Sp}, \text{root}, IS, N), (\text{Sp}, S), (IS, N)). \\
\text{freturn}(\text{patterns}, (\text{Sp}, IS, N), (\text{Sp}, S), (IS, N)) & \leftarrow N \geq \text{Sp}.
\end{align*}
\]

Again, such an approach requires maintaining a copy of all transactions, and is clearly unfeasible when data size is huge. 

\(\triangleright\)
Example 5.5 ([51]). The problem with the above approaches is the lack of optimized data structures that ease the counting and pruning phase. The possibility of using more complex symbols, such as functor symbols and set structures, gives us the opportunity of exploiting more viable solutions, in the style of [5, 137]. The simplest form is that of collecting all the frequent itemsets, that are incrementally computed exploiting the apriori property. Initially, we collect all the 1-itemsets,

\[
\begin{align*}
\text{single}(\text{patterns}, (S, S), ((S, 1), IS)) & \leftarrow \text{single\_isets}(S, IS). \\
\text{multi}(\text{patterns}, ((S, N), IS), (S, S), ((S, N + 1), IS)) & \leftarrow \text{single\_isets}(S, SS), \\
& \quad \text{union}(SS, IS, ISS).
\end{align*}
\]

where the predicate single\_isets is specified by the rule

\[
\text{single\_isets}(S, \{\{M, 0\}\}) \leftarrow \text{member}(M, S).
\]

The subsequent iterations resemble the steps of the apriori algorithm, by counting the candidate itemsets, pruning unfrequent candidates and generating new candidates:

\[
\begin{align*}
\text{iterate}(\text{patterns}, ((S, N), S), (S \times N, S)). \\
\text{iterate}(\text{patterns}, (S, S), (S, SS)) & \leftarrow \text{prune}(S, S, IS), \\
& \quad \text{generate\_candidates}(IS, SS).
\end{align*}
\]

\[
\begin{align*}
\text{multi}(\text{patterns}, (S, IS), (S, S), (S, ISS)) & \leftarrow \text{count\_isets}(IS, S, ISS). \\
\text{freturn}(\text{patterns}, (S, ISS), (IS, N)) & \leftarrow \text{member}((IS, N), ISS), N \geq S.
\end{align*}
\]

Here, the implementation of the main operations of apriori algorithm is directly specified, through the definitions of the predicates \text{prune}, \text{generate\_candidates} and \text{count\_isets}. Since the computationally-intensive tasks of the algorithm are performed by these predicates, the efficiency of such an approach is parametric to the efficient implementation and evaluation of such predicates. <

A more effective approach, formerly studied in [60], is to allow a compromise between loose and tight coupling, by adopting external specialized algorithms (and specialized data structures). Practically, we extend the allowed types of the practical \textsf{LPLC++} system to include more complex structures, as in the case of Object-Relational systems:

\[
\begin{align*}
\text{single}(\text{patterns}, (S, S), ((S, 1), T)) & \leftarrow \text{init}(S, T). \\
\text{multi}(\text{patterns}, (S, S), ((S, N), T), ((S, N + 1), T)) & \leftarrow \text{init}(S, T), \\
\text{iterate}(\text{patterns}, ((S, N), T), (S \times N, T)) & \leftarrow \text{prune}(S, T), \\
& \quad \text{enahance}(T). \\
\text{multi}(\text{patterns}, (S, S), (S, T), (S, T)) & \leftarrow \text{count}(S, T). \\
\text{iterate}(\text{patterns}, (S, T), (S, T)) & \leftarrow \text{prune}(S, T), \text{enhance}(T).
\end{align*}
\]
\begin{equation}
\text{return}(\text{patterns}, (Sp, T), (I, S)) \leftarrow \text{itemset}(T, (I, S)). \tag{5.6}
\end{equation}

In such a schema, the patterns aggregate is implemented with the \textit{apriori} algorithm for the computation of the association rules [5], shown in Fig. 2.8. Here, the variable \( T \) represents the reference (e.g., the object identifier) to a structure of type Hash-Tree (introduced in [5]; see chapter 7 and [61] for further details). The predicates \textit{init}, \textit{count}, \textit{enhance}, \textit{prune} and \textit{itemset} are \textit{user-defined predicates} that implement, in a procedural language such as C++, complex operators over the given Hash-tree abstract data type. More specifically:

- The \textit{init}(I, T) predicate initializes and updates the frequencies the 1-itemsets available from \( I \) in \( T \).
- The \textit{count}(I, T) predicate updates the frequencies of each itemset in \( T \) according to transaction \( I \).
- The \textit{prune}(M, T) predicates removes all the itemsets in \( T \) whose frequencies are less than \( M \).
- The \textit{enhance}(T) predicates combines the frequent \( k \)-itemsets in \( T \) and generates the candidate \( k + 1 \)-itemsets.
- Finally, the \textit{itemset}(T, S) predicate extracts the frequent itemset \( I \) (whose frequency is \( S \)) from \( T \).

The above schema is of great significance, since

- it maintains a declarative view of the above specification, that is parametric to the intended meaning of the user-defined predicates adopted;
- it only reduces the “black-box” structure of the algorithm, needed to obtain fast counting of candidate itemsets and efficient pruning, to the minimal optimizations necessary;
- it provides many opportunities of optimizing the execution of the algorithm both from a database optimization perspective and from a “constraints” embedding perspective [118].

In order to implement the above schema, we exploit the open-architecture provided by the \( {\mathcal{L}D{L}}+ + \) system. However, we postpone the implementation details concerning the above schema, to chapter 7. To the purpose of this section, it is important to put in evidence that the above schema maintains the structure of the results and guarantees a substantial efficiency in the query processing phase.

The above specifications give evidence of the claim of section 4.3.2 for the case of frequent patterns discovery. The following definition provides an instance of the \textit{frequent pattern discovery inductive schema}:

\begin{itemize}
\item ...
\end{itemize}
Definition 5.2. Let \( r \) be an instance of the table \( R = A_1 \ldots A_n \), and \( \sigma \in [0,1] \). For given \( i, j \leq n \), let

- \( \mathcal{L} = \{ W \mid W \subseteq \text{dom}(R[A_i]) \} \), and
- \( q(r, s) = \text{true if and only if } \text{freq}(s, r) \geq \sigma \)

Where \( \text{freq}(s, r) \) is the (relative) frequency of \( s \) in the set of the transactions in \( r \) grouped by \( A_j \). The theory \( \mathcal{T}(\mathcal{L}, r, q) \) defines the frequent patterns discovery task.

As a result of the above specifications, we obtain:

**Proposition 1.** The patterns aggregate corresponds to the frequent patterns discovery inductive schema.

Proof (sketch). There is a correspondence between patterns in \( \mathcal{T}(\mathcal{L}, r, q) \) and the predicates resulting from the evaluation of the following rules:

\[
\begin{align*}
q(\text{patterns}((\sigma, T))) & \leftarrow r\text{Set}(I, T). \\
r\text{Set}(A_j, (A_i)) & \leftarrow r(A_1, \ldots, A_n).
\end{align*}
\]

From rule 5.6, each predicate \( q(i, s) \) in the iterated stable model of the above program represents an itemset \( i \) and its frequency \( s \). The sets of rules 5.4 and 5.5 guarantee that only frequent itemsets are maintained in the hash tree structure \( T \).

\[\square\]

### 5.2 Using the patterns Aggregate

This section shows some examples of complex queries within the resulting logic language. We shall refer to the table with schema and contents exemplified in fig. 5.1, and show how high-level specifications can be formalized in the logical framework obtained by adding the patterns aggregate to \( \mathcal{LDC}++ \).

**Example 5.6.** The high-level specification task “Find patterns with at least 4 occurrences from the daily transactions of each customer” can be formalized be the following set of rules:

\[
\begin{align*}
\text{frequentPatterns}(\text{patterns}((0.25, S))) & \leftarrow \text{transSet}(D, C, S). \\
\text{transSet}(D, C, (I)) & \leftarrow \text{transaction}(D, C, I, P, Q).
\end{align*}
\]

The aggregate rule specifies the data mining task, that is applied against the data prepared by the second rule. By querying \( \text{frequentPatterns}(F, S) \) we obtain the answers of fig. 5.1 (c).
5.2. USING PATTERNS IN KDD

Example 5.7. The high-level specification "Find patterns with at least 3 occurrences from the transactions of each customers" can be specified as follows:

\[
\text{frequentPatterns(patterns}(\langle 1.0, S \rangle)) \leftrightarrow \text{transSet}(C, S).
\]
\[
\text{transSet}(C, \langle I \rangle) \leftrightarrow \text{transaction}(D, C, I, P, Q).
\]

Differently from the previous example, where transactions were grouped by customer and by date, the previous rules group transactions by customer. We then compute the frequent patterns on the restructured transactions

\[
\text{transSet}(\text{cust1}, \{\text{beer, chips, jackets, pasta, wine}\})
\]
\[
\text{transSet}(\text{cust2}, \{\text{beer, chips, col_shirts, jackets, pasta, wine}\})
\]
\[
\text{transSet}(\text{cust3}, \{\text{beer, brown_shirts, chips, col_shirts, pasta, wine}\})
\]

obtaining the answer frequentPatterns(\{beer, chips, pasta, wine\}, 1.0). 

Example 5.8. The high-level specification "Find association rules with a minimum support 30% from daily transactions of each customer" can be formalized by the rules of example 5.6. Hence, by querying rules(L, R, S, C), we obtain the association rules of fig. 5.1 (d).

We can further postprocess the results of the aggregation query. For example, the query rules(\{A, B\}, \{beer\}, S, C) specifies constraints over the rules to be computed: only rules with 2 antecedents, and with the beer item as the only consequent are returned as answers. An answer is (\{chips, wine\}, \{beer\}, 3, 1).

Example 5.9. The high-level task “find patterns from daily transactions of high-spending customers (i.e., customers with at least 70 of total expense ad at most 3 items brought), such that each pattern has at least 4 occurrences” can be formalized as follows:

\[
\text{frequentPatterns(patterns}(\langle 3, S \rangle)) \leftrightarrow \text{transSet}(D, C, S, I, V), V > 70, I \leq 3.
\]
\[
\text{transSet}(D, C, \langle I \rangle, \text{count}(I), \text{sum}(V)) \leftrightarrow \text{transaction}(D, C, I, P, Q), V = P \times Q.
\]

Again, such a rule models two main steps of the knowledge discovery process: preprocessing and mining, in which the dataset to be mined is tuned according to more refined objectives. The query frequentPatterns(F, S) returns the answer predicates frequentPatterns(\{beer\}, 3), frequentPatterns(\{chips\}, 4) and frequentPatterns(\{beer, chips\}, 3) that characterize the class of high-spending customers.

Example 5.10 ([60]). Background information can be easily included in the mining process. The high-level specification “find patterns from daily transactions of each customer, at each generalization level, such that each pattern has a given occurrence depending from the generalization level” is formalized as follows:
Figure 5.2: (a) Roll-up of the transaction table, according to (b) the category table. (c) Resulting multi-level rules.

\[
\text{itGeneralize}(0, D, C, I, P, Q) \leftarrow \text{transaction}(D, C, I, P, Q).
\]
\[
\text{itGeneralize}(I + 1, D, C, AI, P, Q) \leftarrow \text{itGeneralize}(I, D, C, S, P, Q),
\quad \text{category}(S, AI).
\]
\[
\text{itGenSet}(I, D, C, (S), \text{count}(S), \text{sum}(V)) \leftarrow \text{itGeneralize}(I, D, C, S, P, Q),
\quad V = P \times Q.
\]
\[
\text{freqAtLevel}(I, \text{patterns}((\text{Supp}, S))) \leftarrow \text{itGenSet}(I, D, C, S, \ldots).
\]

The generalization of the itemsets is done recursively. At each level, frequent patterns are computed. The above program is the result of a tighter coupling of data preprocessing and result interpretation and postprocessing: we investigate the behaviour of rules over an item hierarchy. The first three rules exploit the category relation shown in fig. 5.2 (b), and produce the table exemplified (at level 1) in fig. 5.2 (a). The results of mining can be obtained querying \text{freqAtLevel}(I, F, S). The evaluation of such query produces as results tuples like \((0, \{ \text{beer, chips} \}, 0.33)\) and \((1, \{ \text{drinks, food} \}, 0.5)\). As pointed out in [77], it can be unrealistic to compute rules at a given level by using the same support threshold used in lower levels. Hence, the last rule can be easily modified to handle such kind of problem,

\[
\text{freqAtLevel}(I, \text{patterns}((\text{Supp}, S))) \leftarrow \text{itGenSet}(I, D, C, S, \ldots),
\quad \text{suppAtLevel}(I, \text{Supp}).
\]
where the $\text{suppAtLevel}$ predicate tunes the support threshold at a given item hierarchy.

The perhaps most important capability of is that of combining the results of the mining phase in order to evaluate the subjective measures.

**Example 5.11.** The high-level task “find rules that are interestingly preserved by drilling-down an item hierarchy” specifies a complex mining objective, that can be pursued by combining different mining results. For example, preserved rules are defined as those rules valid at any generalization level, such that their confidence is greater than their generalization. The following rules formalize such a notion of interest measure:

$$
\text{rulesAtLevel}(I, L, R, S, C) \leftarrow \text{freqAtLevel}(I, A, S), \text{freqAtLevel}(I, R, S_1), \\
\quad \text{subset}(R, A), \text{difference}(A, R, L), C = S/S_1.
$$

$$
\text{preservedRules}(L, R, S, C) \leftarrow \text{rulesAtLevel}(I + 1, L_1, R_1, S_1, C_1), \\
\quad \text{rulesAtLevel}(I, L, R, S, C), \text{setPartOf}(L, L_1), \\
\quad \text{setPartOf}(R, R_1), C > C_1.
$$

The first clause computes rules at a given level of abstraction. The result of the evaluation of such a clause can be exploited to compute the second rule, that extracts rules having a correspondent at a higher abstraction level with lower confidence. <

**Example 5.12.** More complex reasoning schemes can be easily formalized by combining background knowledge with mining. Suppose that background information concerning events happening in a certain time interval is modeled by means of clauses. For example, the following fact defines the interval of a promotion.

$$
$$

A high-level objective is the study of the effects of such an event over the product purchases. For example, we can study which patterns are established by the promotion, by finding those patterns which did not hold before the promotion, which were raised during the promotion and persisted after the promotion:

$$
\text{freqPat} (\text{before}, \text{patterns} ((0.3, S))) \leftarrow \text{transSet}(L, D, C, S), \\
\quad \text{interval} (\text{promo}, S, E), D < S.
$$

$$
\text{freqPat} (\text{after}, \text{patterns} ((0.8, S))) \leftarrow \text{transSet}(L, D, C, S), \\
\quad \text{interval} (\text{promo}, S, E), D > S.
$$

$$
\text{freqPat} (\text{promo}, \text{patterns} ((0.3, S))) \leftarrow \text{transSet}(L, D, C, S), \\
\quad \text{interval} (\text{promo}, S, E), S \leq D \leq E.
$$

$$
\text{preservedPatterns}(S) \leftarrow \text{freqPat} (\text{promo}, S, \_), \\
\quad \neg \text{freqPat} (\text{before}, S, \_), \\
\quad \text{freqPat} (\text{after}, S, \_).
$$

Frequent patterns are computed on different partitions, with different support levels. The last clause selects patterns that do not actually hold in the time partition preceding the promotion, but happen to “survive” to the promotion. <
5.3 Special Cases

In this section we show how the framework used to define the patterns aggregate can be adapted to the most important extensions to the frequent pattern discovery task, shown in section 2.5.1. There are two main modifications to the above specification:

- result representation, i.e. how mined knowledge is represented;
- data reorganization, i.e., which kind of data is mined.

Some of the extensions also need a substantial modification of the aggregate specification, through the adoption of specialized algorithms. The mechanism of iterative aggregates allows the formalization of such task with minor modifications to the schema shown in the previous section.

5.3.1 Multidimensional patterns

Multidimensional patterns are patterns that relate values of different attributes. As shown in section 2.5.1, mining of multidimensional association rules are just a matter of representation, since the tuples of a relation can be represented as itemsets, and transactions can be represented by binary relations. In fact, by suitably representing the input attributes, we can make use of the formalization 5.2:

\[ p(X_1, \ldots, X_n, \text{patterns}((\min\text{Supp}, \{Y_1, \ldots, Y_m\})) \leftarrow r(Z_1, \ldots, Z_m) \]

In this rule, the variables \(X_1, \ldots, X_n, Y_1, \ldots, Y_m\) are a rearranged subset of the variables \(Z_1, \ldots, Z_k\) of \(q\), and each \(Y_i\) is a representation of a dimension to mine. The aggregate patterns computes the set of predicates \(p(t_1, \ldots, t_n, s, f)\) where:

1. \(t_1, \ldots, t_n\) are distinct instances of the variables \(X_1, \ldots, X_n\), as resulting from the evaluation of \(r\);
2. \(s = \{1_1, \ldots, 1_k\}\) is a rearranged subset of the values of \(Y_1, \ldots, Y_m\) in a tuple resulting from the evaluation of \(q\).

**Example 5.13.** Let us consider again the relation \(R = ABC\) shown in fig. 2.9 (a). The multidimensional patterns mining task can be specified by the following rule:

\[ \text{mDPats}(\text{patterns}((0.6, \{\text{first\_attr(B), second\_attr(C)}\}))) \leftarrow r(A, B, C). \]

To each attribute we associate a tag that uniquely identifies the values associated to the dimension it represents. The evaluation of such rule produces, e.g., the frequent itemset

\[ \text{mDPats}\{\{\text{first\_attr(c_1), second\_attr(b_1)}\}, 0.6\} \]

Example 5.14. Let us consider the transaction table of fig. 5.1 (a). The query “find rules relating the day-of-week of purchase and the item purchased, with support greater than 6%" can be formalized by the following rules:

\[
\text{frqMDimPat}((0.06, \{\text{dow}(D), \text{item}(I)\})) \leftarrow \text{transaction}(D, C, I, P, Q), \\
\text{day of week}(D, D).
\]

\[
\text{answer}(\{\text{dow}(D)\}, \{\text{item}(I)\}, S, C) \leftarrow \text{frqMDimPat}(\{\text{dow}(D), \text{item}(I)\}, S), \\
\text{frqMDimPat}(\{\text{dow}(D)\}, S_1), C = S/S_1.
\]

The first clause specifies the mining of two-dimensional patterns. The second clause computes, from frequent patterns, rules having a day of week as antecedent, and an item as consequent. The result of the evaluation of such rules contains, e.g., the predicate \text{answer}(\{\text{dow(wednesday)}\}, \{\text{item(beer)}\}, 5, 0.3).

5.3.2 Item Hierarchies

The problem of mining rules within a concept hierarchy can be specified in many different ways: example 5.10 shows how the roll-up/down interaction can be formalized using the \text{patterns} aggregate. We can be also interested in frequent patterns containing elements of heterogeneous hierarchy levels [138]. Hence, it is interesting to see how the \text{patterns} aggregate can be made viable for mining drill-through patterns. As shown in [137], the problem of mining such generalized patterns can be easily tackled by means of suitable preprocessing.

Example 5.15. Let us consider the transaction table of fig. 5.1, and the category table of fig. 5.2. The simplest way of computing generalized patterns, as suggested by [137], is to enlarge the transactions, by including the ancestors of all the items in the transactions:

\[
\text{extTrans}(D, C, I, P, Q) \leftarrow \text{transaction}(D, C, I, P, Q), \\
\text{extTrans}(D, C, A, P, Q) \leftarrow \text{extTrans}(D, C, I, P, Q), \\
\text{category}(I, A),
\]

\[
\text{extTransSet}(D, C, \{I\}) \leftarrow \text{extTrans}(D, C, I, P, Q), \\
\text{extFreqPat}(\langle \text{min\_supp}, \text{IS} \rangle) \leftarrow \text{extTransSet}(D, C, \text{IS}).
\]

The patterns resulting from the evaluation of the above program can hence be “normalized": no itemsets containing both an item and its ancestor is returned as answer:

\[
\text{validPatterns}(\text{IS}, \text{F}) \leftarrow \text{extFreqPat}(\text{IS}, \text{F}), \neg \text{related}(\text{IS}). \\
\text{related}(\text{IS}, \text{F}) \leftarrow \text{member}(I, \text{IS}), \text{member}(A, \text{IS}), \text{is}\_\text{a}(I, A).
\]

The \text{is}\_\text{a} predicate computes the transitive closure of the category table. The evaluation of the above rules drops all frequent itemsets containing both an item and its ancestor.
In practice, we can adopt the optimizations described in [137] directly in the specification of the mining aggregate. First of all, we can adopt a slightly different representation of items:

- all ancestors of an item are precomputed, and

- each item is represented by a more complex structure (e.g., a list) that explicitly specifies all its ancestors in straight line. For example, the list \([c, b, a]\) represents the item \(a\), its parent \(b\) and its grandparent \(c\).

**Example 5.16.** The above rules allow the pre-computation of all ancestors of the items of table `transaction`:

\[
\begin{align*}
\text{transHier}(D, C, [I], Q, P) & \leftarrow \text{transaction}(D, C, I, Q, P), \\
\text{transHier}(D, C, [A, I|Is], Q, P) & \leftarrow \text{transHier}(D, C, [I|Is], Q, P), \text{category}(I, A), \\
\text{transHier}(D, C, [I|Is], Q, P) & \leftarrow \text{transHier}(D, C, [I|Is], Q, P), \neg \text{category}(I, \ldots).
\end{align*}
\]

The first two clauses compute in a list the transitive closure of each item in the `transaction` relation. The last clause selects the items that are represented with all its ancestors. 

The schema shown in subsection 5.1.1 can hence be refined. In particular, we need to refine the user-defined predicates of clauses 5.4, 5.5 and 5.6, in order to take advantage of such a different representation, and mine generalized patterns:

- the `init` predicate can be modified to update the hash-tree structure with both the items and its ancestors: for each “extended” item \([a_1, \ldots, a_n]\), the `init` predicate generates \(n\) nodes, namely \([a_1], [a_1, a_2], \ldots, [a_1, \ldots, a_n]\).

- the `count` predicates can be modified to update the counters of itemsets by exploiting hierarchical information contained into the pre-computed transactions: given a transaction \(\{l_1, \ldots, l_n\}\), the counter of an itemset \(\{c_1, \ldots, c_k\}\) is incremented if and only if for each \(c_i = [a_{i_1}, \ldots, a_{i_m}]\) there exist an extended item \(l_j = [a_{i_1}, \ldots, a_{i_h}]\) (with \(m \geq h \geq 1\)).

- the `enhance` predicate can be modified in order to generate candidate itemsets such that no itemset contains an item \([a_1, \ldots, a_n]\) and one of its ancestors \([a_1, \ldots, a_i]\).

### 5.3.3 Sequential Patterns

In order to deal with sequential patterns, we need to organize the transactions in sequences, according to a given sequence identifier and a precedence relation.
5.3. SPECIAL CASES

<table>
<thead>
<tr>
<th>Customer</th>
<th>Itemset</th>
</tr>
</thead>
<tbody>
<tr>
<td>cust1</td>
<td>{beer, chips, wine}</td>
</tr>
<tr>
<td></td>
<td>{chips, beer}</td>
</tr>
<tr>
<td></td>
<td>{pasta, chips}</td>
</tr>
<tr>
<td></td>
<td>{jackets}</td>
</tr>
<tr>
<td></td>
<td>{pasta, wine, chips, beer}</td>
</tr>
<tr>
<td>cust2</td>
<td>{wine, beer, pasta, chips}</td>
</tr>
<tr>
<td></td>
<td>{wine, pasta}</td>
</tr>
<tr>
<td></td>
<td>{beer, chips}</td>
</tr>
<tr>
<td></td>
<td>{chips, col_shirts}</td>
</tr>
<tr>
<td>cust3</td>
<td>{wine, beer}</td>
</tr>
<tr>
<td></td>
<td>{chips, col_shirts}</td>
</tr>
<tr>
<td></td>
<td>{pasta}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>{beer}</td>
<td>1.0</td>
</tr>
<tr>
<td>{beer}</td>
<td>1.0</td>
</tr>
<tr>
<td>{beer, wine}</td>
<td>1.0</td>
</tr>
<tr>
<td>{beer, wine}</td>
<td>1.0</td>
</tr>
<tr>
<td>{chips}</td>
<td>1.0</td>
</tr>
<tr>
<td>{wine}</td>
<td>1.0</td>
</tr>
<tr>
<td>{wine}</td>
<td>1.0</td>
</tr>
<tr>
<td>{pasta}</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Figure 5.3: (a) Sequences of purchases in the transaction table. (b) Frequent patterns.

Example 5.17. Let us consider the transaction table of example 5.1. A suitable representation of sequences of transactions can be obtained using lists. For example, the sequence of purchases of cust3 can be represented by the fact

\[
\text{transSeq}(\text{cust3}, [\{\text{wine, beer}\}, \{\text{chips, col_shirts}\}, \{\text{pasta}\}])
\]

There are many ways of preprocessing the transaction table in order to obtain the above representation. For example, we can collect elements of the transSet table

\[
\text{transSeq}(\text{C, slist}((\text{D, S}))) \leftarrow \text{transSet}((\text{D, C}, \text{S})).
\]

using the slist aggregate, that extracts ordered lists according to the D term:

\[
\text{single(slist, (D, S), [(D, S)])}.
\]

\[
\text{multi(slist, [ ], (D, S), [(D, S)])}.
\]

\[
\text{multi(slist, [(D, S), [Ls]], (D, S), [(D, S), (D_1, S_1)[Ls]]}) \leftarrow D \leq D_1.
\]

\[
\text{multi(slist, [(D, S), [Ls]], (D, S), [(D, S), (D_1, S_1)[Ls]]}) \leftarrow D > D_1,
\]

\[
\text{multi(slist, Ls, (D, S), Ls_1)}.
\]

\[
\text{return(slist, Ls, L) \leftarrow remove_dates(Ls, L)}.
\]

\[\triangleq\]

We can specify the problem of mining sequential patterns by adopting a new user-defined aggregate, the seq aggregate, conceptually similar to pattern. The seq aggregate can be used in rules like the following,

\[
p(\text{seq}((\min\_\text{supp}, L))) \leftarrow r(X_1, \ldots, X_n)
\]

where
• L is a list of itemsets, resulting from the evaluation of r

• the evaluation of the rule returns an answer predicates \( p(l', f) \), where \( 1 = \{s_1, \ldots, s_n\} \) is a sequence supported by an instance of L in a tuple resulting from the evaluation of r.

**Example 5.18.** The rule

\[
\text{freqSequences(seq}(1.0, L)) \leftarrow \text{transSeq}(C, L).
\]

specifies the computation of frequent pattern sequences from the transaction table. The results of the evaluation of such rule are shown in fig. 5.3 (b).

Again, the formal specification of the seq aggregate can be done by means of user-defined predicate, that use a specialized data structure, as shown in [137, Chapt. 5], and capable to implement ad-hoc candidate generation and pruning, as well as candidate counting (as shown in fig. 2.10).

### 5.4 Clustering and Frequent Patterns

In chapter 2 we described the clustering and association techniques as examples of undirected knowledge discovery techniques. There are many applications in which it makes sense to apply these techniques in combination, in order to find interesting relationships that otherwise could be hardly found. For example, association rules can help in explaining clustering results: we mine each different cluster for association rules, and provide an interpretation of the given cluster by means of the discovered rules. An example application in which such a combination is profitable is customer profiling.

Clustering is perhaps the most straightforward example of data mining task that provides a suitable representation of results as relational tables. In a relation \( R \) with instance \( r \), the main objective of clustering is that of labeling each tuple \( \mu \in r \). In relational terms, this correspond in adding a set of attributes \( A_1, \ldots, A_n \) to \( R \), so that a tuple \( \langle a_1, \ldots, a_n \rangle \) associated to a tuple \( \mu \in r \) represents a cluster assignment for \( \mu \). For example, we can enhance \( R \) with two attributes \( C \) and \( M \), where \( C \) denotes the cluster identifier and \( M \) denotes a probability measure. A tuple \( \mu \in r \) is represented in the enhancement of \( R \) by a new tuple \( \mu' \), where \( \mu'[A] = \mu[A] \) for each \( A \in R \), and \( \mu'[C] \) represents the cluster to which \( \mu \) belongs, with probability \( \mu'[M] \). An inductive definition formalizing such concepts is immediate.

**Definition 5.3.** Given a relation \( R = A_1 \ldots A_n \) with extension \( r \), such that tuples in \( r \) can be organized in \( k \) clusters, an inductive database modeling clustering is defined by \( \text{TH}(L, r, q) \), where

\[
L = \{ (<\mu, i>) | \mu \in \text{dom}(A_1) \times \ldots \times \text{dom}(A_n), i \in \mathbb{N} \}, \text{ and}
\]
5.4. CLUSTERING AND FREQUENT PATTERNS

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>age</th>
<th>income</th>
</tr>
</thead>
<tbody>
<tr>
<td>cust1</td>
<td>pisa</td>
<td>50</td>
<td>50K</td>
</tr>
<tr>
<td>cust2</td>
<td>rose</td>
<td>30</td>
<td>30K</td>
</tr>
<tr>
<td>cust3</td>
<td>pisa</td>
<td>45</td>
<td>45K</td>
</tr>
<tr>
<td>cust4</td>
<td>florence</td>
<td>24</td>
<td>30K</td>
</tr>
<tr>
<td>cust5</td>
<td>pisa</td>
<td>60</td>
<td>50K</td>
</tr>
<tr>
<td>cust6</td>
<td>rose</td>
<td>26</td>
<td>30K</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>age</th>
<th>income</th>
<th>cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>cust7</td>
<td>pisa</td>
<td>50</td>
<td>50K</td>
<td>1</td>
</tr>
<tr>
<td>cust2</td>
<td>rose</td>
<td>30</td>
<td>30K</td>
<td>2</td>
</tr>
<tr>
<td>cust3</td>
<td>pisa</td>
<td>45</td>
<td>45K</td>
<td>1</td>
</tr>
<tr>
<td>cust4</td>
<td>florence</td>
<td>24</td>
<td>30K</td>
<td>2</td>
</tr>
<tr>
<td>cust5</td>
<td>pisa</td>
<td>60</td>
<td>50K</td>
<td>1</td>
</tr>
<tr>
<td>cust6</td>
<td>rose</td>
<td>26</td>
<td>30K</td>
<td>2</td>
</tr>
</tbody>
</table>

(b)

<table>
<thead>
<tr>
<th>cluster</th>
<th>pattern</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{s_{\text{p}}(\text{50K})}</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>{f_{\text{p}}(\text{pisa})}</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>{s_{\text{p}}(50K), f_{\text{p}}(\text{pisa})}</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>{f_{\text{rose}}}</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>{s_{\text{p}}(\text{30K})}</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>{f_{\text{rose}}, s_{\text{p}}(\text{30K})}</td>
<td>2</td>
</tr>
</tbody>
</table>

(c)

Figure 5.4: a) Sample customer table. b) Cluster assignments. c) Cluster explanations with frequent patterns.

- \(q(r, (\mu, i))\) is true if and only if \(\mu \in r\) is assigned to the \(i\)-th cluster.

As shown in example 4.8, it is particularly intuitive to specify the clustering data mining task as an iterative aggregate.

**Definition 5.4.** Given a relation \(r\), the cluster aggregate is defined by the rule schema

\[
p(X_1, \ldots, X_n, \text{clusters}((Y_1, \ldots, Y_k))) \leftarrow r(Z_1, \ldots, Z_m).
\]

where the variables \(X_1, \ldots, X_n, Y_1, \ldots, Y_k\) are a rearranged subset of the variables \(Z_1, \ldots, Z_m\) of \(r\). \text{clusters} computes the set of predicates \(p(t_1, \ldots, t_n, s_1, \ldots, s_k, c)\), where:

1. \(t_1, \ldots, t_n, s_1, \ldots, s_k\) are distinct instances of the variables \(X_1, \ldots, X_n, Y_1, \ldots, Y_k\), as resulting from the evaluation of \(r\);
2. \(c\) is a label representing the cluster to which the tuple \(s_1, \ldots, s_k\) is assigned, according to some clustering algorithm.

**Example 5.19.** Consider a relation \(\text{customer(name, address, age, income)}\), storing information about customers, as shown in fig. 5.4 a). We can define a “clustered” view of such a database by means of the following rule:

\[
\text{custCView(clusters((N, AD, AG, I)))} \leftarrow \text{customer}(N, AD, AG, I).
\]
The evaluation of such rule, shown in fig. 5.4 b), produces two clusters. We can easily exploit the patterns aggregate to find an explanation of such clusters:

\[ \text{frqPat}(C, \text{patterns}((0.6, \{f \_a(AD), s \_a(I)\})) \leftarrow \text{custView}(N, AD, AG, I, C). \]

Table 5.4 c) shows the patterns resulting from the evaluation of such a rule. A simple analysis of such results shows that cluster 1 is mainly composed by high-income people living in Pisa, while cluster 2 is mainly composed by low-income people living in Rome.

The specification of the clusters aggregate as an iterative aggregate depends, as usual, from the algorithm we aim at implementing. For example, as exemplified in section 4.4, it is easy to specify the K-Means algorithm: we start by randomly choosing some cluster centers, and then we iterate until the cluster means are stable.

\[
\begin{align*}
\text{single} & (\text{clusters}, X, (\text{init}, T)) & \leftarrow \text{random}(X, T). \\
\text{multi} & (\text{clusters}, (\text{init}, T), X, (\text{init}, T)) & \leftarrow \text{random}(X, T). \\
\text{iterate} & (\text{clusters}, (\text{init}, T), (T)). \\
\text{multi} & (\text{clusters}, (T), X, (T)) & \leftarrow \text{assign}(X, T). \\
\text{iterate} & (\text{clusters}, (T), (T)) & \leftarrow \neg \text{stable}(T). \\
\text{iterate} & (\text{clusters}, (T), (\text{final}, T)) & \leftarrow \text{stable}(T). \\
\text{multi} & (\text{clusters}, (\text{final}, T), X, (\text{final}, T)). \\
\text{return} & (\text{clusters}, (\text{final}, T), X, (X, C)) & \leftarrow \text{min\_dist}(X, T, C).
\end{align*}
\]

We use a complex data structure, represented by the variable T, to contain all the cluster means. The random predicate randomly chooses a tuple X as a cluster mean. The stable predicate is used to check the stability condition of the cluster. The assign predicate recomputes the cluster assignment for the tuple X, and the min\_dist predicate returns the cluster assignments. The above specification is worth two observations:

- In principle, the above specification can be “purely declarative”: for example, by specifying T as a list of tuples, and directly defining the predicates that manipulate it. Of course, as for the case of association rules, such a solution is unpractical for efficiency reasons.

- We didn’t specify which metric (and distance function) is used in the algorithm: rather, any implementation detail is hidden by the user-defined predicates. This is clearly a modeling choice: different metrics define different clustering methods, and hence different aggregates.
5.4. CLUSTERING AND FREQUENT PATTERNS

<table>
<thead>
<tr>
<th>date</th>
<th>customer</th>
<th>itemset</th>
<th>cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>11-2-97</td>
<td>cust1</td>
<td>[beer, chips, wine]</td>
<td>2</td>
</tr>
<tr>
<td>11-2-97</td>
<td>cust2</td>
<td>[wine, beer, pasta, chips]</td>
<td>2</td>
</tr>
<tr>
<td>13-2-97</td>
<td>cust1</td>
<td>[chips, beer]</td>
<td>1</td>
</tr>
<tr>
<td>13-2-97</td>
<td>cust2</td>
<td>[jackets, col_shirts]</td>
<td>0</td>
</tr>
<tr>
<td>13-2-97</td>
<td>cust3</td>
<td>[wine, beer]</td>
<td>2</td>
</tr>
<tr>
<td>15-2-97</td>
<td>cust1</td>
<td>[pasta, chips]</td>
<td>2</td>
</tr>
<tr>
<td>16-2-97</td>
<td>cust1</td>
<td>[jackets]</td>
<td>0</td>
</tr>
<tr>
<td>16-2-97</td>
<td>cust2</td>
<td>[wine, pasta]</td>
<td>2</td>
</tr>
<tr>
<td>16-2-97</td>
<td>cust3</td>
<td>[chips, col_shirts]</td>
<td>0</td>
</tr>
<tr>
<td>18-2-97</td>
<td>cust1</td>
<td>[pasta, wine, chips, beer]</td>
<td>2</td>
</tr>
<tr>
<td>18-2-97</td>
<td>cust2</td>
<td>[beer, chips]</td>
<td>1</td>
</tr>
<tr>
<td>18-2-97</td>
<td>cust3</td>
<td>[pasta]</td>
<td>2</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>cluster</th>
<th>representative</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[jackets, col_shirts]</td>
</tr>
<tr>
<td>1</td>
<td>[beer, chips]</td>
</tr>
<tr>
<td>2</td>
<td>[beer, wine, chips, pasta]</td>
</tr>
</tbody>
</table>

(b)

Figure 5.5: a) Cluster assignments for the transaction table. b) Cluster centers.

5.4.1 Transactional Clustering

In the previous section we have mainly dealt with multidimensional data, and have shown how a clustering task over such data can be specified in Datalog++. Interestingly, the clustering data mining task shows a duality with the problem of frequent patterns discovery, since clustering of transactions can be particularly useful in many application areas, ranging from discovery of behavioral patterns of navigation Web URLs, to market basket analysis and customer profiling (e.g., for detecting purchasing patterns). Like in the case of unidimensional and multidimensional patterns, we can redefine the clusters aggregate in order to deal with transactions.

The implementation of such an aggregate can exploit the transaction clustering algorithm as shown in section 2.5.2, that is a fine tuning of the classical K-Means algorithm with suitable distance and cluster mean definitions.

Example 5.20. The following rule defines clustering of the transactions resulting from the transaction table of fig. 5.1, with $\gamma$ value set to 0.1:

$$\text{transClust}\left(\text{clusters}\left(\langle0.1,S\rangle\right)\right) \leftarrow \text{transSet}\left(D,C,S\right).$$

The results of the evaluation of such rule are shown in fig. 5.5 a). Figure 5.5 shows the cluster centers that are computed by the transaction clustering algorithm. Such centers provide a suitable explanation of the contents of the transactions of each cluster. For example, transactions in cluster 1 usually contain the items beer or chips (and are very likely to contain both of them). Cluster 2 contains transactions that, differently from cluster 1, are likely to contain also wine and chips. 

\(\blacktriangleleft\)
Chapter 6

Classification in Deductive Databases

Abstract

In this chapter we consider the problem of tuning the logic-based framework defined in the previous chapters in order to deal with a classification task. Classification is fundamentally a statistical estimation problem: given a set \( X \) of attributes, the problem is to estimate the probability that the class variable \( C \) takes on some value \( c \). As shown in section 4.4, the problem can be decomposed in two main phases. In the first phase, we collect sufficient statistics over the data. In the second phase, we exploit the statistics to estimate class probabilities. The attempt to formalize a classification task in a logic-based framework must tackle both the two phases accurately. The first phase is mainly the collection of multiple aggregations over the same data set. As pointed out in [29], the nature of such aggregations is different from that needed in the CUBE construct shown in section 2.2.2. The aggregations should be performed with varying filter expressions, and require computing counts for each combination of attribute value and class value.

As a further contribution of this chapter, it is particularly interesting to see how the classification framework can be tuned with ad-hoc preprocessing tasks. In particular, the combined use of discretization techniques with classification has been shown very effective both in learning speed and in prediction accuracy [41]. Hence, discretization tasks are worth a detailed study within the logic-based framework defined in chapter 4.

6.1 Task Formulation

We are interested in specifying a classification construct based on the notion of user-defined aggregate. We extend the approach shown in the previous chapters to
Naive Bayes Classification, that is particularly attractive in our framework for the following reasons [111, 159, 42]:

1. It is one of the most practical approaches to several types of learning problems;

2. It can be specified in a straightforward way in query languages capable of expressing and computing distributive aggregates.

It is particularly simple to specify Naive Bayes classification by means of an inductive database schema. Let us consider a relation $R$ with attributes $A_1, \ldots, A_n, C$. For simplicity, we shall assume that all the attribute represent discrete values. This is not a major problem, since

- it is well-known that classification algorithms perform better with discrete-valued attributes (e.g., in [129] an analysis of the behavior of the C4.5 algorithm over numeric-valued data is presented);

- supervised discretization (surveyed in section 2.2.1) combined with discrete classification allows a more effective approach. As we shall see, the framework based on iterative user-defined easily allows the specification of discretization tasks.

To summarize, given an instance $\mathbf{r}$ of $R$, we aim at computing the function

$$\max_c \Pr(C = c | A_1 = a_1, \ldots, A_n = a_n, \mathbf{r})$$

(6.1)

where $c \in \text{cod}(C)$ and $a_i \in \text{cod}(A_i)$. As explained in section 2.4, by repeated application of Bayes’ rule and the assumption that $A_1, \ldots, A_n$ are independent, we obtain

$$\Pr(C = c | A_1 = a_1, \ldots, A_n = a_n, \mathbf{r}) = \Pr(C = c | \mathbf{r}) \times \prod_i \Pr(A_i = a_i | C = c, \mathbf{r})$$

Now, each factor in the above product can be estimated from the training data by means of the following equation

$$\tilde{\Pr}(A_j = a_j | C = c, \mathbf{r}) = \frac{\text{freq}(c, \sigma_{A_j = a_j}(\mathbf{r}))}{\text{freq}(c, \mathbf{r})}$$

A key insight [71] is that the above statistics can be computed directly and efficiently by means of aggregation queries.

**Example 6.1.** Let us consider the table of figure 6.1. We would like to predict the probability of playing tennis, given the values of the other attributes. In order to obtain the statistics $\tilde{\Pr}(A_j = a_j | \text{Play} = \text{yes})$ and $\tilde{\Pr}(A_j = a_j | \text{Play} = \text{no})$ for each value $a_j$ of each attribute $A_j$, we can simply specify the following rules:
6.1. Task Formulation

<table>
<thead>
<tr>
<th>Outlook</th>
<th>Temperature</th>
<th>Humidity</th>
<th>Wind</th>
<th>Play</th>
</tr>
</thead>
<tbody>
<tr>
<td>overcast</td>
<td>hot</td>
<td>normal</td>
<td>weak</td>
<td>yes</td>
</tr>
<tr>
<td>sunny</td>
<td>hot</td>
<td>high</td>
<td>weak</td>
<td>no</td>
</tr>
<tr>
<td>sunny</td>
<td>hot</td>
<td>high</td>
<td>strong</td>
<td>no</td>
</tr>
<tr>
<td>overcast</td>
<td>hot</td>
<td>high</td>
<td>weak</td>
<td>yes</td>
</tr>
<tr>
<td>overcast</td>
<td>cool</td>
<td>normal</td>
<td>strong</td>
<td>no</td>
</tr>
<tr>
<td>rain</td>
<td>cool</td>
<td>normal</td>
<td>weak</td>
<td>yes</td>
</tr>
<tr>
<td>rain</td>
<td>mild</td>
<td>high</td>
<td>weak</td>
<td>yes</td>
</tr>
<tr>
<td>sunny</td>
<td>mild</td>
<td>high</td>
<td>weak</td>
<td>no</td>
</tr>
<tr>
<td>sunny</td>
<td>cool</td>
<td>normal</td>
<td>weak</td>
<td>yes</td>
</tr>
<tr>
<td>rain</td>
<td>mild</td>
<td>normal</td>
<td>weak</td>
<td>yes</td>
</tr>
<tr>
<td>overcast</td>
<td>mild</td>
<td>high</td>
<td>strong</td>
<td>yes</td>
</tr>
<tr>
<td>rain</td>
<td>mild</td>
<td>high</td>
<td>strong</td>
<td>yes</td>
</tr>
</tbody>
</table>

Figure 6.1: Sample playTennis table.

\[
\text{statistics}_{\text{Out}}(O, P, \text{count}(\cdot)) \leftarrow \text{playTennis}(O, T, H, W, P).
\]
\[
\text{statistics}_{\text{Temp}}(T, P, \text{count}(\cdot)) \leftarrow \text{playTennis}(O, T, H, W, P).
\]
\[
\text{statistics}_{\text{Hum}}(H, P, \text{count}(\cdot)) \leftarrow \text{playTennis}(O, T, H, W, P).
\]
\[
\text{statistics}_{\text{Wind}}(W, P, \text{count}(\cdot)) \leftarrow \text{playTennis}(O, T, H, W, P).
\]
\[
\text{statistics}_{\text{Play}}(P, \text{count}(\cdot)) \leftarrow \text{playTennis}(O, T, H, W, P).
\]

The results of the evaluation of such rules can be easily combined in order to obtain the classifier.

The above example shows that we mainly need a set of counts for the number of co-occurrences of each attribute value with each class variable. More specifically, we need a count table [71, 29], giving the set of co-occurrences of each attribute with each class. Hence, an inductive database schema can be defined as modeling the inductive theory by means of the count table resulting from the instance of \( R \).

**Definition 6.1.** Let \( R = A_1 \ldots A_n, C \) be a relation schema. Given an instance \( r \) of \( R \), we define

- \( \mathcal{L} = \{ \langle A_i = a_i \land C = c, v_A, v_C \rangle | a_i \in \text{dom}(A_i), c \in \text{dom}(C) \text{ and } n_A, n_C \in \mathbb{R} \} \).

- \( q(r, \langle A_i = a_i \land C = c, n_A, n_C \rangle) = \text{true if and only if } n_A = \Pr(A_i = a_i | C = c, r) \) and \( n_C = \Pr(C = c | r) \).

The resulting theory \( \mathcal{Th}(\mathcal{L}, r, q) \) formalizes a naive bayesian classification task. □

As usual, we can model the inductive theory by means of a user-defined aggregate.

**Definition 6.2.** Given a relation \( r \), the aggregate \( \text{nbayes} \) is defined by a rule schema

\[
s(x_1, \ldots, x_n, \text{nbayes}((\{(1, A_1), \ldots, (n, A_n)\}, C))) \leftarrow r(z_1, \ldots, z_k).
\]

where
Figure 6.2: classifier tuples for the playTennis table.

- The variables $X_1, \ldots, X_n, A_1, \ldots, A_n, C$ are a (possibly rearranged) subset of the values of $Z_1, \ldots, Z_k$ resulting from the evaluation of $r$

- The result of such an evaluation is a predicate $s(t_i, \ldots, t_n, c, (i, a_i), v_i, v_c)$, representing the set of counts of all the possible values $a_i$ of the $i$-th attribute $A_i$, given any possible value $c$ of $C$.

Example 6.2. Let us consider the playTennis table with extension shown in fig. 6.1. The count table for such a relation is built by means of the rule

$$\text{classifier(nbayes}((\{(1, 0), (2, T), (3, H), (4, W)\}, P))) \leftarrow \text{playTennis}(O, T, H, W, P).$$

that is equivalent, w.r.t. the answers provided, to the set of rules of example 6.1. The evaluation of the query $\text{classifier}(C, F, C_F, C_C)$ returns as answer the tuples of fig. 6.2

It is straightforward to build a classifier from the classification model resulting from the evaluation of the $\text{nbayes}$ aggregate. Given a tuple to classify, we simply need to evaluate the formula 6.1.

Example 6.3. The following rule defines the Naive Bayesian classifier:
6.1. TASK FORMULATION

classify(O,T,H,W, \text{argmax}(C,P)) \leftarrow classify(C, (1,0), C_o, C_c),
classify(C, (2,T), C_o, C_c),
classify(C, (3,H), C_o, C_c),
classify(C, (4,W), C_o, C_c),
P = C_c \times C_o / C_c \times C_o / C_c
\times C_n / C_c.

where the \text{argmax} aggregate is defined in example 3.8. Given a tuple \( (o,t,h,w) \),
where \( o \in \text{dom}(\text{Outlook}) \), \( t \in \text{dom}(\text{Temperature}) \), \( h \in \text{dom}(\text{Humidity}) \) and
\( w \in \text{dom}(\text{Wind}) \), the instance classify\((o,t,h,w,T)\) guesses the value to which \( T \)
must be instantiated, according to the classification model defined by the relation
classifier. For example, the query classify\((\text{overcast},\text{hot},\text{normal},\text{strong},T)\)
returns as answer the value yes for \( T \).

The rule schema shown in the previous examples can be generalized: given a
relation \( r \) with a target attribute and \( n \) features attributes, we can define a clause
defining a classifier\(_r\) predicate (which exploits the nbayes aggregate), and a
clause defining a classify\(_r\) predicate (exploiting the \text{argmax} aggregate and the
classifier\(_r\) predicate).

Definition 6.3. Given a predicate \( p \) defined according to the schema of definition 6.2, a classifier predicate classify\(_p\) is obtained by the following clause schema:

classify\(_p\)(X\(_1\), \ldots, X\(_m\), A\(_1\), \ldots, A\(_n\), \text{argmax}(C,P)) \leftarrow p(X\(_1\), \ldots, X\(_m\), C, (1, A\(_1\)), C\(_1\), C\(_c\)),
p(X\(_1\), \ldots, X\(_m\), C, (2, A\(_2\)), C\(_2\), C\(_c\)),
\vdots
p(X\(_1\), \ldots, X\(_m\), C, (n, A\(_n\)), C\(_n\), C\(_c\)),
P = C_1 / C_c \times C_2 / C_c
\times \ldots \times C_n / C_c.

6.1.1 Specifying the nbayes Iterative Aggregate

As explained in [71], a simple way for computing the count tables is that of exploiting
an unpivoted table, i.e., a sort of “transactional” representation of a flat table.
The motivations [71] for introducing such constructs are mainly based on the con-
sideration that count tables can only be computed in a relational environment using
a union query, i.e., by combining the results of different queries (whose execution is
optimized separately). For example the evaluation of the program of example 6.1,
requires the computation of four different aggregates.

Example 6.4 ([71, 147]). Consider the playTennis table of fig. 6.1. An unpivoted
view of such a table represents each tuple with 4 new tuples, where each new tuple
contains a pair with an attribute value and a class value. A possible representation
of such a view is given by a predicate unpivotPT(RecID, ColID, Value, Play). The statistics of example 6.1 can hence be computed by means of the following rule:

\[
\text{classifier}((C,V), P, \text{count}(\ast)) \leftarrow \text{unpivotPT}(R, C, V, P).
\]

By providing a suitable definition of the unpivotPT(RecID, ColID, Value, Play) predicate, the computation of the statistics can hence be specified in a constant number of scans.

A simple approach to the definition of unpivoted views can be that of exploiting user-defined aggregates, in the style of [147].

Example 6.5. The unpivotPT predicate can be defined by exploiting the unpvt aggregate,

\[
\text{unpivotPT(unpvt}((\{(1,0), (2,T), (3,H), (4,W)\}, P))) \leftarrow \text{playTennis}(O,T,H,W,P).
\]

defined by means of the following clauses:

\[
\begin{align*}
\text{single}(\text{unpvt}, (F, C), (1, I, A, C)) & \leftarrow \text{member}((I, A, F)). \\
\text{multi}(\text{unpvt}, (N, \ldots, N), (F, C), (N + 1, I, A, C)) & \leftarrow \text{member}((I, A, F)). \\
\text{return}(\text{unpvt}, (N, I, A, C), (N, I, A, C)). & \leftarrow \text{member}((I, A, F)). \\
\end{align*}
\]

The single and multi predicates disassemble each tuple \((\{(1, a_1), \ldots, (n, a_n)\}, c)\) in \(n\) tuples \((j, i, a_1, c)\), where \(j\) is a given record identifier.

The approach based on the unpvt aggregate has the main advantage of allowing two simple scans of a relation \(r\) with \(n\) attributes, while instead the approach based on union queries requires \(n\) scans. However, the use of the unpivoted table (that can result very huge) can be avoided by observing that it suffices to identify all the grouping sets (i.e., all the distinct pairs attribute value/class value), and then counting the occurrences of each grouping set. The definition of the nbayes aggregate refines the unpvt approach in this direction:

\[
\begin{align*}
\text{single}(\text{nbayes}, (F, C), (I, A, C)) & \leftarrow \text{member}((I, A, F)). \\
\text{multi}(\text{nbayes}, (I, A_1, C_1), (F, C), (I, A, C)). & \leftarrow \text{member}((I, A, F)). \\
\text{multi}(\text{nbayes}, (I, A, C_1), (F, C), (I, A, C)) & \leftarrow \text{member}((I, A, F)). \\
\text{multi}(\text{nbayes}, (I, A, C_A, C_C), (F, C), (I, A, C_A + 1, C_C + 1)) & \leftarrow \text{member}((I, A, F)). \\
\text{multi}(\text{nbayes}, (I, A, C_A + 1, C_C), (F, C), (I, A, C_A, C_C + 1)) & \leftarrow \text{member}((I, A, F)). \\
\text{multi}(\text{nbayes}, (I, A, C_A, C_C), (F, C_1), (I, A, C_A, C_C)) & \leftarrow C \neq C_1. \\
\text{iterate}(\text{nbayes}, (I, A, C), (I, A, 0, C, 0)). & \\
\text{freturn}(\text{nbayes}, (I, A, C_A, C_C), (C, (I, A), C_A, C_C)). \\
\end{align*}
\]
In such rules, the tuple \((F, C)\) represents a tuple in the dataset: \(C\) is the target attribute and \(F\) is the collection of the relevant features. The first three rules build the list of tuples \((i, a, c)\), representing respectively the attribute identifier, the attribute value, and the class value. The `iterate` predicate activates the count phase, performed by the remaining `multi` rules: the occurrences of each tuple obtained from the previous iteration are counted in the dataset. Finally, the count table is returned as an answer.

The above approach performs essentially two iterations, avoiding to build an "unpivoted" view of the entire database. The first is needed to collect the grouping sets, i.e., all the distinct tuples \((i, a, c)\). From a practical implementation viewpoint, we can avoid such an iteration by exploiting, as shown in the previous chapter, more complex data structures and demanding time-consuming computations to the management of such structures:

\[
\text{single(nbayes, (F, C), T)} \leftarrow \text{member}((I, A), F), \text{init}((I, A), C, T).
\]
\[
\text{multi(nbayes, T, (F, C), T)} \leftarrow \text{member}((I, A), F), \text{update}((I, A), C, T).
\]
\[
\text{freturn(nbayes, T, (C, (I, A), C_A, C_C))} \leftarrow \text{path}((I, A), C, T, C_A, C_C).
\]

Again, \(T\) represents a reference to a hash-tree structure, with two only levels, where each level corresponds to an attribute value. Paths from the root to the leaves corresponds to pairs feature/target, and the leaf nodes contain the counts associated to the corresponding pairs. Figure 6.3 shows the example fragment of an hash tree, where the first level correspond to the target attribute, and the second level to the \textbf{Outlook} attribute. The following user-defined predicates implement operations on such a structure:

- the `init` method inserts the pair \((I, A)\) into the tree, and initializes the counter to 1.
- the `update` method checks whether the pair \((I, A), C\) is contained by the tree (in which case it increments the associated counter); if the pair is not contained, it is inserted.
- the `path` method extracts from each path of the tree the tuple that must be returned as answer.

As a result of the above specification, the following proposition trivially holds.

\textbf{Proposition 1.} The `nbayes` aggregate corresponds to the inductive naive Bayesian classification inductive schema.

\textbf{Proof (sketch).} There is a correspondence between patterns in \(Th(L, r, q)\) and the predicates resulting from the evaluation of the rule

\[
q(\text{nbayes}([(\{1, A_1\}, \ldots, (n, A_n)\}, C)) \leftarrow r(A_1, \ldots, A_n, C).
\]

Indeed, the `freturn` rule returns as answers of the clause the tuples \((C, (i, a), n_a, n_c)\), such that:
• from the single and multi rules, \( c \in \text{cod}(C) \) and \( a \in \text{cod}(A_i) \);
• from the iterate and multi rules, \( n_a = freq(c, \sigma_{A_i = a}(r)) \) and \( n_c = freq(c, r) \).

Hence, for each pattern \( \langle A_i = a \land C = c, p_A, p_C \rangle \) there is a corresponding predicate \( q(c, (i, a), n_a, n_c) \) such that \( p_A = n_a/n_c \) and \( p_C = n_c/m \), where \( m \) is the size of \( r \). □

Finally, it is worth to mention that:

• it is easy to adopt a slight modification of the above shown schema, in which the patterns that are returned as answers include the probability estimations of \( \Pr(A_i = a_i | C = c) \) and \( \Pr(C = c) \). We simply need to keep track of the number of tuples, and use them to estimate the given probabilities.

• The above approach fails to deal with 0 probabilities. However, simple corrections can be done, as suggested, e.g., in [111, chapter 6]. For example, we can weight the proposed estimate of \( \Pr(A | B) \) with prior uniform probabilities.

### 6.2 Using the nbayes Aggregate

The rest of the section shows some examples of complex mining tasks specified by exploiting the nbayes aggregate. We shall refer to the playTennis table as a running example.

We adopt a slight modification of the aggregate schema: the nbayes aggregate shall be defined by rules like

\[
s(X_1, \ldots, X_m, \text{nbayes}((P, [A_1, \ldots, A_n], C))) \leftarrow r(Z_1, \ldots, Z_k).
\]

where the variables \( X_1, \ldots, X_m, P, A_1, \ldots, A_n, C \) are a (possibly rearranged) subset of \( Z_1, \ldots, Z_k \) and \( r(Z_1, \ldots, Z_k) \) is either an extensional or an intensional predicate. The result of such an evaluation is the set of conditional probabilities of each of the possible values of \( A_1 \), given any possible value of \( C \), and a weight \( P \) associated to the tuple \( A_1, \ldots, A_n \). For example, a simple classifier is built by means of the rule

\[
\text{classifier(nbayes}((1, [0, T, H, W], P))) \leftarrow \text{playTennis}(0, T, H, W, P).
\]

that gives the same weight to each tuple.
Example 6.6: It is easy to build a predicate that evaluates the goodness of such a classifier. The high-level specification task “Compute the misclassification rate of the trained model” can be formalized by the following rule:

\[
\text{misclassified}(O, T, H, W, \text{Play}, \text{Predicted}) \leftarrow \text{testSet}(O, T, H, W, \text{Play}), \\
\quad \text{classify}(O, T, H, W, \text{Predicted}) \\
\text{Play} \neq \text{Predicted}.
\]

When run against the playTennis table, the misclassified relation returns the predicates

\[
\text{misclassified(overcast, cool, normal, strong, no, yes)} \\
\text{misclassified(overcast, mild, high, strong, yes, no)} \\
\text{misclassified(rain, mild, high, strong, no, yes)}
\]

representing the misclassified portion of the table.

In the previous example, the predicate testSet can be defined either as an extensional predicate or as an intensional predicate. This suggests a way for explicitly defining a boosting technique.

Example 6.7 (Boosting). The high-level specification task for boosting “compute an increasing sequence of classifiers such that each classifier is built by increasing the weight of the tuples misclassified by the preceding classifier” can be modeled by the following set of rules:

\[
\text{case}(0, 1, O, T, H, W, P) \leftarrow \text{playTennis}(O, T, H, W, P). \\
\text{case}(I + 1, WW + 1, O, T, H, W, P) \leftarrow \text{case}(I, WW, O, T, H, W, P), \\
\quad \text{misclassified}(I, O, T, H, W, P). \\
\text{case}(I + 1, WW, O, T, H, W, P) \leftarrow \text{case}(I, WW, O, \text{Temp}, H, W, P), \\
\quad \neg \text{misclassified}(I, O, T, H, W, \_). \\
\text{classifier}(I, \text{nbayes}([WW, [O, T, H, W], P])) \leftarrow \text{case}(I, WW, O, T, H, W, P). \\
\text{misclassified}(I, O, T, H, W, P) \leftarrow \text{case}(I, WW, O, T, H, W, P), \\
\quad \text{classify}(I, O, T, H, W, \text{Pred}), \\
\quad P \neq \text{Pred}.
\]

Classifiers are incrementally built and identified by the stage argument \(I\). At each stage, the training set is built by incrementing the weight of the misclassified tuples. In order to obtain the misclassification rate of each classifier, we have to count the misclassified tuples:

\[
\text{totMisclassified}(I, \text{count}(*)) \leftarrow \text{misclassified}(I, O, T, H, W, P).
\]

The first argument of the totMisclassified predicate represents the classifier identifier, and the second argument represents the number of tuples misclassified by
the classifier. By considering the first 10 classifiers built we obtain the following misclassification rate for each classifier:

\[
\begin{align*}
\text{totMisclassified}(0,3) & \quad \text{totMisclassified}(1,3) & \quad \text{totMisclassified}(2,3) \\
\text{totMisclassified}(3,2) & \quad \text{totMisclassified}(4,3) & \quad \text{totMisclassified}(5,2) \\
\text{totMisclassified}(6,1) & \quad \text{totMisclassified}(7,5) & \quad \text{totMisclassified}(8,3) \\
\text{totMisclassified}(9,2) & & \\
\end{align*}
\]

\[\triangledown\]

Example 6.8 (Meta-Learning). Auto-focusing mechanisms can be easily defined. We can “train a classifier as a coordinator of a set of classifiers built by boosting, where each classifier has a voting weight depending by its misclassification rate”:

\[
\begin{align*}
\text{votation}(O, T, H, W, C, \text{sum}(V)) & \quad \leftarrow \text{classify}(I, O, T, H, W, C, P), \\
 & \quad \text{totMisclassified}(I, T), V = 1/T. \\
\text{boostClassify}(O, T, H, W, \text{argmax}(\langle C, N \rangle)) & \quad \leftarrow \text{votation}(O, T, H, W, C). \\
\end{align*}
\]

The misclassification rate of the boosting classifier is then computed by the rule

\[
\begin{align*}
\text{boostMisclassified}(\text{count}(\ast)) & \quad \leftarrow \text{playTennis}(O, T, H, W, C), \\
 & \quad \text{boostClassify}(O, T, H, W, \text{Pred}), \\
 & \quad C \neq \text{Pred}. \\
\end{align*}
\]

\[\triangledown\]

Example 6.9 (Cross-Validation). A cross-validation technique can be used to evaluate the prediction accuracy of the classification task against a given dataset. Practically, we can randomly split a dataset into \(K\) different training sets and test-sets, and then compute the average prediction accuracy of the adopted classification method.

\[
\begin{align*}
\text{crValidate}(K, N, \text{nbayes}(\langle 1, [O, T, H, W], P \rangle)) & \quad \leftarrow \text{nthTrainingSet}(K, N, O, T, H, W, P). \\
\text{nthTrainingSet}(K, N, O, T, H, W, P) & \quad \leftarrow \text{playTennis}(O, T, H, W, P), \\
 & \quad \text{\neg belongs}(O, T, H, W, P, K, N). \\
\text{nthTestSet}(K, N, O, T, H, W, P) & \quad \leftarrow \text{playTennis}(O, T, H, W, P), \\
 & \quad \text{belongs}(O, T, H, W, P, K, N). \\
\text{totMisclassified}(K, N, \text{count}(\langle O, T, H, W \rangle)) & \quad \leftarrow \text{nthTestSet}(K, N, O, T, H, W, P), \\
 & \quad \text{classify}(K, N, O, T, H, W, \text{Pred}), \\
 & \quad P \neq \text{Pred}. \\
\text{crossValidation}(K, \text{avg}(C)) & \quad \leftarrow \text{totMisclassified}(K, N, C). \\
\end{align*}
\]

Here, the predicate \(\text{belongs}(O, T, H, W, P, K, N)\) specifies the splitting policy of the dataset. For example, any tuple can be randomly assigned to any value \(n\) between 0 and \(n\). \[\triangledown\]
6.3 Generalizations

The Naive Bayes classification task is particularly easy to specify as a user-defined aggregate, since it allows suitable representation of the patterns of $\mathcal{L}$ in a logic-based language. However, it suffers of a main drawback: the assumption that attributes are independent, which is a very strong assumption in practice. This section is devoted to analyze whether different classification techniques allow a similar tight-coupling. In principle, it is always possible to consider a classifier model as a “black box” object that takes as input a tuple and returns as answer the class label associated to the tuple. For example, we can define the predicate classify($A_1, \ldots, A_n, C$) as a user-defined predicate, that takes as input an instance of the variable $A_1, \ldots, A_n$ and returns as answer a binding for the variable $C$. The definition of classify, as well as the data mining model adopted, is hidden by its implementation in a procedural language. Approaches of this kind have been investigated in [64, 16]. However, such a decoupled approach does not allow the definition of any user-defined measure of the quality of the data mining model and, moreover, does not allow to explicitly exploit background knowledge in the process of building the model.

6.3.1 Simplification

A naive Bayes classifier makes strong independence assumptions. It is not surprising that these assumptions are likely to fail in real-life applications. If they fail, the classifier may be worse than necessary. To cope with this problem, one may try to simplify the Naive Bayes Classifier, using a simple greedy attribute selection procedure [18]. With this procedure it can be hoped that a subset of attributes is found for which the independence assumptions hold at least approximately.

The attribute selection methods that can be used are mainly two.

- In the first method we start with a classifier that simply predicts the majority class. That is, we start with a classifier that does not use any attribute information. Then we add attributes one by one. In each step we select the attribute which, if added, leads to the smallest number of misclassifications on the training data. We stop adding attributes when adding any of the remaining attributes does not reduce the number of errors.

- The second method is a reversal of the first. We start with a classifier that takes into account all available attributes and then removes attributes step by step. In each step we select the attribute which, if removed, leads to the smallest number of misclassifications on the training data. We stop removing attributes when removing any of the remaining attributes leads to a larger number of errors.

Example 6.10. A simplified classifier on the playTennis table based on the first of the above described techniques can be built by means of the following rules:
\[
\text{classifier}(0, [\text{nbayes}([[, P])]) \leftarrow \text{playTennis}(0, T, H, W, P).
\]
\[
\text{classifier}(I + 1, L, \text{nbayes}(Xs, P)) \leftarrow \text{playTennis}(0, T, H, W, P),
\text{eligible}(I, Ls),
\text{extend}(Ls, L),
\text{sel_atrrs}(L, [0, T, H, P], Xs).
\]
\[
\text{misclassified}(I, L, \text{count}(*)) \leftarrow \text{playTennis}(0, T, H, W, P),
\text{classify}(I, L, 0, T, H, W, \text{Pred}),
P \neq \text{Pred}.
\]
\[
\text{extend}(N, Ls, [J|Ls]) \leftarrow \text{misclassified}(I, L, V).
\]
\[
1 \leq J \leq N, \neg \text{member}(I, Ls).
\]

Each classifier is parametric to the list of the attributes that it uses. Initially, a classifier with no attribute is built. At a given stage, a pattern is chosen among the available patterns in the previous stage, that provides the minimum error rate. Such a pattern is extended and classifiers are computed for each extension. The algorithm terminates when no further extensions can be made.

### 6.3.2 Perceptron Learning

The boosting technique shown so far is an example on how further classification schemas can be built starting from the basic approach. Indeed, the similarity between the boosting approach and some neural network approaches [42] suggest a simple way of formalizing the Perceptron training phase (the Gradient-Descent algorithm [111, 82, 14]) as an iterative user defined aggregate,

\[
p(\text{perceptron}(\epsilon, \eta, [X_1, \ldots, X_n], T) \leftarrow r(X_1, \ldots, X_n, T).
\]

where \( T \) represent a binary class variable, \([X_1, \ldots, X_n]\) the feature vector, \( \epsilon \) the minimum error threshold and \( \eta \) the update step. Practically, at each iteration, we compute the update to the vector of weights \( \vec{w} \) by using the delta rule

\[
w_i \leftarrow \eta (o_d - t_d)x_{id}
\]

for each tuple \( d \) such that the target value is \( t_d \) and the observed value \( o_d \) is obtained by means of the formula

\[
o_d = \text{sign}(\sum_i w_i x_{id})
\]

The weights are updated until the termination condition

\[
\sum_d (o_d - t_d)^2 < \epsilon
\]

is met. The specification is straightforward. Let us assume, for simplicity, that the feature vector \([X_1, \ldots, X_n]\) has only one element \( X \). Then, we can specify the steps as follows:
6.3. Generalizations

\[
\begin{align*}
\text{single} & (\text{perceptron}, (P, G, X, T), (D, S, P)) \leftarrow \text{sign}(X, 0), \\
D & = G \times (O - T) \times X, \\
sqr & (O - T, S).
\end{align*}
\]

\[
\begin{align*}
\text{multi} & (\text{perceptron}, (W, E, P), (P, G, X, T), (W + D, E + S, P)) \leftarrow \text{sign}(W \times X, 0), \\
D & = G \times (O - T) \times X, \\
sqr & (O - T, S).
\end{align*}
\]

iterate(\text{perceptron}, (W, E, P), (W, 0, P)) \leftarrow P < E.
\text{return}(\text{perceptron}, (W, E, P), \text{W})
\text{sign}(V, 1) \leftarrow V \geq 0.
\text{sign}(V, -1) \leftarrow V < 0.

Here, the single rule analyzes the first tuple under the assumption that the weight is initialized to 1. single and multi compute updates to both the error rate E and the weight W. The procedure iterates until the error is lowered to the given threshold P. As a result, the aggregate returns the computed weight, that can be exploited to classify unknown instances:

\[
\text{classify}(X, 0) \leftarrow p(W), \text{sign}(W \times X, 0).
\]

6.3.3 Decision tree classification

In section 2.4 we have seen the general schema for a decision-tree classification algorithm. Such a recursive schema can be easily fitted into an iterative schema, provided that a suitable representation formalism is defined for trees (e.g., a tabular representation of decision nodes). Notice in fact that, like in the case of Bayesian classification, the tree-learning algorithm requires two phases: the computation of count tables at each node (corresponding to a selection of the rows of the original dataset), and the use of such tables to compute statistics. However, in order to define an iterative user-defined aggregate that builds a decision tree, we have to deal with the problem that, as soon as a tree is built, the number of rows that need
to be considered may decrease. For example, if a decision tree is trained over the 
playTennis table using the gain-ratio measure, the first decision node is assigned to
the Outlook attribute. Such an attribute produces three partitions: the partitions
corresponding to the values sunny and rain (that need to be further analyzed),
and the partition corresponding to the value overcast (that produces a leaf node).
The following iterations do not need to further analyze the tuples of the dataset
containing the value overcast. However, an iterative schema shall consider all the
available tuples at each iteration.

In [147], a relational user-defined aggregate is defined, that exploits internal
tables to copy the relevant portions of the table, and recursion to grow the decision
tree over such portions. Such an approach can be also adopted in our framework,
by demanding the decision-tree construction to external user-defined predicates:

\[
\begin{align*}
\text{single} & (\text{tree}, (F, C), T) \leftarrow \text{insert} (F, C, T). \\
\text{multi} & (\text{tree}, T, (F, C), T) \leftarrow \text{insert} (F, C, T). \\
\text{freturn} & (\text{tree}, T, R) \leftarrow \text{grow} (T), \text{tree-nodes} (T, R).
\end{align*}
\]

Practically, the \text{insert} user defined predicate copies the \((F, C)\) tuple into an internal
table, and the \text{grow} predicate uses the internal copy of the tuples to grow the tree.
Finally, the \text{tree-nodes} predicate extracts a suitable representation of the decision
nodes from the tree. As we can see, such an approach delegates the management of
the tuples needed to build the classifier to external data structures, and hides the
steps of the learning algorithm (in which portions of the initial table are progressively
reorganized or discarded, in order to speed-up the learning phase) in the definition
of \text{grow}.

6.4 Classification and Discretization

The classification framework shown in section 6.1 considers only discrete-valued
attributes. It is natural to ask whether the approach can be tuned in order to deal
with continuous-valued attributes, too. The most convenient approach is to combine
classification with discretization techniques. It is remarkable that a discretization
task fits naturally as a preprocessing step for many data mining tasks: for example,
in mining of multidimensional association rules [139], and in classification, where,
even for algorithms that explicitly deal with continuous features, such as C4.5,
learning is more accurate and faster if data is discretized [41]. Hence, it is interesting
to investigate a whether a discretization task can be formalized in the logic-based
framework defined in chapter 4.

Given an instance \(r\) of a relation \(R\) with a numeric attribute \(A\), the main idea is to
provide a mapping among the values of \(\text{cod}(A)\) and some given labels. More precisely,
we define \(\mathcal{L}\) as the pairs \((a, i)\), where \(a \in \text{dom}(A)\) and \(i \in \mathcal{V}\) represents an interval
label (i.e., \(\mathcal{V}\) is a representation of all the intervals \([l, s]\) such that \(l, s \in \text{dom}(A)\)). A
valid discretization of the tuples of \(r\), formalized as a theory \(T h(\mathcal{L}, r, q)\), is obtained
### 6.4. Classification and Discretization

<table>
<thead>
<tr>
<th>bar</th>
<th>beer</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Bud</td>
<td>100</td>
</tr>
<tr>
<td>A</td>
<td>Becks</td>
<td>120</td>
</tr>
<tr>
<td>C</td>
<td>Bud</td>
<td>117</td>
</tr>
<tr>
<td>D</td>
<td>Bud</td>
<td>130</td>
</tr>
<tr>
<td>D</td>
<td>Bud</td>
<td>150</td>
</tr>
<tr>
<td>E</td>
<td>Becks</td>
<td>140</td>
</tr>
<tr>
<td>E</td>
<td>Becks</td>
<td>122</td>
</tr>
<tr>
<td>F</td>
<td>Bud</td>
<td>121</td>
</tr>
<tr>
<td>G</td>
<td>Bud</td>
<td>133</td>
</tr>
<tr>
<td>H</td>
<td>Becks</td>
<td>125</td>
</tr>
<tr>
<td>H</td>
<td>Bud</td>
<td>160</td>
</tr>
<tr>
<td>I</td>
<td>Bud</td>
<td>135</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>value</th>
<th>level</th>
<th>interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>120</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>117</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>130</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>150</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>140</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>122</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>121</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>133</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>125</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>160</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>135</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 6.5: a) The beer.distributions relation. b) Unsupervised discretization in 6 intervals. c) ChiMerge Discretization.

by suitably defining $q(x, \langle a, i \rangle)$ in a way that relates $a$ to the interval $i$, according to some discretization objective. In section 2.2.1 we have shown two main classes of discretization methods. In the following we shall analyze them in deeper details.

#### 6.4.1 Unsupervised Discretization

Unsupervised discretization techniques have the objective of producing $k$ intervals, for a given $k$.

**Definition 6.4.** Let $x$ be an instance of a relation $R$. Given an attribute $A \in R$, an inductive database theory $Th(\mathcal{L}, x, q)$ represents an unsupervised discretization task if

- either $\text{dom}(A) = \mathbb{N}$ or $\text{dom}(A) = \mathbb{R}$;
- $\mathcal{L} = \{ \langle a, k, i \rangle | a \in \text{dom}(A), k, i \in \mathbb{N} \}$;
- $q(x, \langle a, k, i \rangle) = true$ if and only if there is a discretization of $\pi_A(x)$ in $k$ intervals, and $a$ belongs to the $i$-th interval.

Such a definition is quite straightforward: the inductive database provides a (virtual) table mapping each continuous value of the desired attribute into an interval, at a desired interval level. Figure 6.5 shows an example inductive instance, corresponding to the toy beer.distributions relation.

As usual, we can provide a model of the above inductive theory in terms of user-defined aggregates.
Definition 6.5. Given a relation \( r \), the aggregate \( \text{discr} \) defines the rule schema

\[
s(Y_1, \ldots, Y_k, \text{discr}(Y, K)) \leftarrow r(X_1, X_2, \ldots, X_n).
\]

where

- \( Y \) is a continuous-valued variable, and \( K \) is an integer-valued variable;
- the variables \( Y_1, \ldots, Y_k, Y, K \) are a rearranged subset of \( X_1, X_2, \ldots, X_n \) in \( r \).

The result of the evaluation of such a rule is given by predicates \( p(t_1, \ldots, t_k, v, k, i) \), where \( t_1, \ldots, t_k, v, k \) are distinct instances of the variables \( Y_1, \ldots, Y_k, Y, K \), and \( i \) \((1 \leq i \leq k)\) is an integer value representing the \( i \)-th interval in an unsupervised discretization of the values of \( Y \) in \( k \) intervals.

**Example 6.11.** The following rule

\[
\text{intervals} \left( \text{discr}(\langle \text{Price}, 5 \rangle) \right) \leftarrow \text{beer.distributions}(\text{Bar, Beer, Price}).
\]

defines the unsupervised discretization of the \text{price} attribute in 5 intervals of the relation \text{beer.distributions} shown in fig. 6.5.

As seen in chapter 2, there are two mostly used approaches to unsupervised discretization.

**Natural binning.** For a given number \( k \) of intervals to be generated, we compute the discretization interval width \( \delta = (x_{\text{max}} - x_{\text{min}})/k \), and then assign each value \( x \) to the interval \( j \) such that \( x \in [x_{\text{min}} + j\delta, x_{\text{min}} + (j + 1)\delta) \). The specification of two simple iterations over the data can easily allow the formalization of such a method as an iterative user-defined aggregate: first, in order to compute the width of the intervals, we collect the minimum and the maximum values. Next, in the second iteration we assign each element to the corresponding interval:

- \( \text{single} \left( \text{discr}, \langle X, K \rangle, \langle X, X, K \rangle \right) \).
- \( \text{multi} \left( \text{discr}, \langle \text{Min}, \text{Max}, N \rangle, \langle X, K \rangle, \langle \text{Min}, X, N \rangle \right) \leftarrow X > \text{Max} \).
- \( \text{multi} \left( \text{discr}, \langle \text{Min}, \text{Max}, N \rangle, \langle X, K \rangle, \langle \text{Min}, \text{Max}, N \rangle \right) \leftarrow X \leq \text{Max}, X > \text{Min} \).
- \( \text{multi} \left( \text{discr}, \langle \text{Min}, \text{Max}, N \rangle, \langle X, K \rangle, \langle X, \text{Max}, N \rangle \right) \leftarrow X \leq \text{Min} \).
- \( \text{multi} \left( \text{discr}, \langle D, N \rangle, \langle X, K \rangle, \langle D, N \rangle \right) \).
- \( \text{iterate} \left( \text{discr}, \langle \text{Min}, \text{Max}, N \rangle, \langle (\text{Max} - \text{Min})/N, N \rangle \right) \).
- \( \text{return} \left( \text{discr}, \langle M, D \rangle, \langle X, K \rangle, \langle X, K, I \rangle \right) \leftarrow \text{interval} \left( X, M, 0, I \right) \).
- \( \text{interval} \left( X, D, J, J \right) \leftarrow D \times J \leq X < D \times (J + 1) \).
- \( \text{interval} \left( X, D, J, I \right) \leftarrow X \geq D \times (J + 1), \text{interval} \left( X, D, J + 1, I \right) \).

The use the \( \text{return} \) predicate is possible here because, after the first iteration, we only need to compute the interval membership of each tuple, and we don’t need to complete the execution of the second iteration for this.
6.4. CLASSIFICATION AND DISCRETIZATION

Equal frequency binning. This method produces balanced intervals, i.e., intervals with a fixed number \( f \) of elements, given by \( f = \lfloor N/k \rfloor \). The assignment of each element to the corresponding interval is done by looking at the rank of the element: the \( j \)-th element \( x_j \) is assigned to interval \( i \) if \( i \times f \leq j < (i+1) \times f \). The specification of an efficient way for computing the rank of the elements is problematic. A naive definition can be easily provided: in a way similar to the specification of the median aggregate of example 3.13, we specify two iterations, in order (i) to collect all the distinct values and (ii) to "nonterministically" compute the rank of each value in the sorted order:

\[
\begin{align*}
\text{single} & (\text{discr}, (X,K),(X,K)). \\
\text{multi} & (\text{discr}, (X,K), \neg (X,K)). \\
\text{multi} & (\text{discr}, \neg (X,K),(X,K)). \\
\text{iterate} & (\text{discr}, (X,K),(X,0,K,0)). \\
\text{multi} & (\text{discr}, (X,N,K),(Y,K),(X,R+1,K,N+1)) \leftarrow Y < X. \\
\text{multi} & (\text{discr}, (X,N,K),(Y,K),(X,N,K,N+1)) \leftarrow Y \geq X. \\
\text{freturn} & (\text{discr}, (X,R,K,N),(X,K,I)) \leftarrow \text{interval}(R,N/K,0,I).
\end{align*}
\]

However, such a specification requires \( O(n^2) \) comparisons, where \( n \) is the number of distinct values of \( X \). In order to improve the efficiency, we can make use of more complex structures, to be managed by means of user-defined predicates. The following specification, for example makes use of a heap structure to sort the elements and compute the intervals.

\[
\begin{align*}
\text{single} & (\text{discr}, (X,K),(T,K,1)) \leftarrow \text{heap-insert}(X,T). \\
\text{multi} & (\text{discr}, (T,K,N),(X,K),(T,K,N+1)) \leftarrow \text{heap-insert}(X,T). \\
\text{freturn} & (\text{discr}, (T,N),(X,K,I)) \leftarrow \text{heap-nth-el}(T,X,R), \\
& \text{interval}(R,N/K,0,I).
\end{align*}
\]

The above specification only needs one iteration. The \text{heap-insert} predicate inserts an element into the heap structure \( T \) (with \( \log(n) \) comparisons). In the \text{freturn} rule, the \text{heap-nth-el} predicate extracts element \( X \) with rank \( R \) from \( T \). For each element, the corresponding interval is then computed. As a result, the overall process requires \( O(n \log n) \) comparisons.

6.4.2 Supervised Discretization with ChiMerge

As explained in section 2.2.1, the objective of supervised method is to discretize continuous values in homogeneous intervals, i.e., intervals that preserve a predefined property (which, in practice, is represented by a label associated to each continuous value). The formalization of supervised discretization as an inductive theory can be tuned accordingly.
Definition 6.6. Let \( r \) be an instance of a relation \( R \). Given an attribute \( A \in R \), and a discrete-valued attribute \( C \in R \), an inductive database theory \( Th(\mathcal{L}, r, q) \) defines a supervised discretization task if

- either \( \text{dom}(A) \in \mathbb{N} \) or \( \text{dom}(A) \in \mathbb{R} \);
- \( \mathcal{L} = \{(a, C, i) | a \in \text{dom}(A), i \in \mathbb{N}\} \);
- \( q(r, (a, C, i)) = \text{true} \) if and only if there is a discretization of \( \pi_A(r) \) in homogeneous intervals with respect to the attribute \( C \), such that \( a \) belongs to the \( i \)-th interval.

The corresponding aggregate definition is straightforward.

Definition 6.7. Given a relation \( r \), the aggregate \( \text{discr} \) defines the rule schema

\[
s(Y_1, \ldots, Y_k, \text{discr}(\langle Y, C \rangle)) \leftarrow r(x_1, x_2, \ldots, x_n).
\]

where

- \( Y \) is a continuous-valued variable, and \( C \) is a discrete-valued variable;
- \( Y_1, \ldots, Y_k, Y, C \) are a rearranged subset of the variables \( x_1, x_2, \ldots, x_n \) in \( r \).

The result of the evaluation of such rule is given by predicates \( p(t_1, \ldots, t_k, v, i) \), where \( t_1, \ldots, t_k, v \) are distinct instances of the variables \( Y_1, \ldots, Y_k, Y \), and \( i \) is an integer value representing the \( i \)-th interval in a supervised discretization of the values of \( Y \) labelled with the values of \( C \).

In the ChiMerge approach shown in section 2.2.1, we adopt a bottom-up interval generation procedure, determining at each iteration how "distinct" are two adjacent intervals, and merging the less distinct intervals. In order to formalize such a discretization process as a user-defined aggregate, we need to specify the desired level of significance \( \alpha \), identifying the probability that two adjacent intervals have independent label distributions.

\[
s(Y_1, \ldots, Y_k, \text{discr}(\langle Y, C, \alpha \rangle)) \leftarrow r(x_1, x_2, \ldots, x_n).
\]

Example 6.12. The following rule

\[
\text{intervals(discr(\langle \text{Price}, \text{Beer}, 0.9 \rangle))} \leftarrow \text{beer.distributions(\_ \_ \_ , \text{Beer}, \text{Price})}.
\]

defines a 0.9 significance level supervised discretization of the price attribute, according to the labels in beer, of the relation beer.distributions shown in fig. 6.5. The results of the evaluation of such rule are shown in fig. 6.5 c).
It is particularly easy to combine such a discretization technique with a classification task, by exploiting the logical formalizations.

Example 6.13. A typical dataset used as a benchmark for classification tasks is the Iris classification dataset [98]. The dataset can be represented by means of a relation containing 5 attributes:

\[
\text{iris(Sepal\_length, Sepal\_width, Petal\_length, Petal\_width, Specie)}
\]

Each tuple in the dataset describes the relevant features of an iris flower. The first four attributes are continuous-valued attributes, and the Specie attribute is a nominal attribute corresponding to the class to which the flower belongs (either iris-setosa, iris-versicolor, or iris-virginica). In order to exploit the \text{nbayes} aggregate to classify the tuples of the relation, we need a preprocessing phase in which continuous attributes are discretized:

\[
\begin{align*}
\text{intervals}_\text{SL}(\text{discr}(\langle \text{SL}, \text{C}, 0.9 \rangle)) & \leftarrow \text{iris}(\text{SL}, \text{SW}, \text{PL}, \text{PW}, \text{C}). \\
\text{intervals}_\text{SW}(\text{discr}(\langle \text{SW}, \text{C}, 0.9 \rangle)) & \leftarrow \text{iris}(\text{SL}, \text{SW}, \text{PL}, \text{PW}, \text{C}). \\
\text{intervals}_\text{PL}(\text{discr}(\langle \text{PL}, \text{C}, 0.9 \rangle)) & \leftarrow \text{iris}(\text{SL}, \text{SW}, \text{PL}, \text{PW}, \text{C}). \\
\text{intervals}_\text{PW}(\text{discr}(\langle \text{PW}, \text{C}, 0.9 \rangle)) & \leftarrow \text{iris}(\text{SL}, \text{SW}, \text{PL}, \text{PW}, \text{C}).
\end{align*}
\]

The predicates defined by such rules provide a mapping of the continuous values to the intervals shown in fig. 6.6. The specification of the classification phase is done by exploiting such predicates:

\[
\begin{align*}
\text{irisCl}(\text{nbayes}((\{\text{SL}, \text{SW}, \text{PL}, \text{PW}\}, \text{C}))) & \leftarrow \text{iris}(\text{SL1}, \text{SW1}, \text{PL1}, \text{PW1}, \text{C}), \\
\text{intervals}_\text{SW}(\text{SW1}, \text{SW}), \\
\text{intervals}_\text{SL}(\text{SL1}, \text{SL}), \\
\text{intervals}_\text{PW}(\text{PW1}, \text{PW}), \\
\text{intervals}_\text{PL}(\text{PL1}, \text{PL}).
\end{align*}
\]

The count-tables resulting from the evaluation of the predicate defined by the above rule are shown in fig. 6.6. The definition of the corresponding classifier predicate can exploit the discretization results as well. For example, a classifier associated to the attributes \text{sepal\_length} and \text{sepal\_width} is defined by the following rule:

\[
\begin{align*}
\text{classify}(\text{SL}, \text{SW}, \text{argmax}(\langle \text{C}, \text{P} \rangle)) & \leftarrow \text{intervals}_\text{SW}(\text{SW}, \text{SW1}), \\
\text{irisCl}(\text{C}, \text{SW1}, \text{CSW}, \text{C}_\text{c}), \\
\text{intervals}_\text{SL}(\text{SL}, \text{SL1}), \\
\text{irisCl}(\text{C}, \text{SL1}, \text{CSL}, \text{C}_\text{c}), \\
P = \text{CSL} \times \text{CSW}/\text{C}_\text{c}.
\end{align*}
\]

As an example, the evaluation of the tuple (4.3, 3.4) returns as answer the predicate \text{classify}(4.3, 3.4, \text{setosa}). \raisebox{1pt} {\topfill}
Figure 6.6: Bayesian tables using ChiMerge.

Like in the case of decision-tree classification, a "purely declarative" specification of the ChiMerge discretization aggregate is problematic, since it requires a sorting of the tuples, and a further iterative procedure in which intervals are merged until their $\chi^2$ level of significance is relevant (as shown in fig. 2.3). Each iteration works with a different manipulation of the initial dataset (i.e., a set of intervals, represented as pairs $(i, s)$). The originay tuples are used only to provide the initial intervals, and are of no further usefulness. Hence, it is not practical to specify the phases of the algorithm by means of iterations over such tuples.

In order to overcome this, we can, as usual, resort to an internal (hidden) representation of intervals, and code the main computations of the algorithm by means of user-defined predicates that manipulate such internal structure:

- `single(discr, (X, C, A), (T, A)) ← chi_insert(X, T).`
- `multi(discr, (T, A), (X, C, A), (T, A)) ← chi_insert(X, T).`
- `iterate(discr, (T, A), (T)) ← chi_eval(T, A).`
- `multi(discr, (T), (X, C, A), (T)).`
- `return(discr, (T), (X, C, A), (X, I)) ← chi_interval(X, T, I).`

The above rules specify two iterations. In the first one, elements are collected into an internal structure $T$, representing the intervals. The `iterate` predicate is defined in terms of the `chi_eval(T, A)` user-defined predicate, which computes the intervals according to the ChiMerge procedure of fig. 2.3. Finally, the last iteration assigns each element to the corresponding interval.
Chapter 7
The Physical Level

Abstract

Logic-based database languages are extremely hard to tune to high performance, since for this there's the need to understand how the engine works (e.g., with or without tail recursion optimization makes a huge difference); and the order of the clauses may as well affect the performance severely. On the other hand, performance fine tuning is the heart of any data mining algorithm, since data mining methods by definition, run against huge data. So this may look like a conflict.

However, the problem of efficiently coupling data mining with database systems is common to many approaches. Thus, achieving a reasonable performance tuning at least in the relational fragment provides the basis of a significant comparison with different approaches. Moreover, it results in a conservative approach, since further optimization techniques, aiming at improving the integration of data mining tasks with recursion and negation, can be studied starting from such basis.

This chapter is aimed at showing in greater details how to obtain an efficient implementation of the iterative aggregates shown in the previous sections. The semantics provided in chapter 3, although worth being used as a practical implementation, is not adequate from an efficiency point of view. The $\mathcal{CDC}++$ system adopts a more efficient (but semantically equivalent) implementation schema, that can be easily modified in order to deal with iterative aggregates, as we will show in section 7.2.1.

A further aspect to be investigated is the notion of user-defined predicate. In chapters 5 and 6 we claimed that an extensive use of such predicates, aimed at efficiently implementing costly operations, can reasonably solve the problem of evaluating “data mining” queries. Section 7.2.2 details the implementation of the Apriori algorithm, as envisaged in chapter 5. The performance analysis of section 7.3, accomplished using some standard benchmarks commonly used in the current literature, gives evidence of the validity of the claim.
7.1 Problem Formulation

The problem of efficiently coupling data mining with database systems is common to many approaches, and provides diverse architectural alternatives. Directly coding the algorithm within the DBMS environment (e.g., by means of Datalog queries in our framework) makes it difficult to adopt ad-hoc data structures and procedures that allow specific optimizations [112, 129]. It has been experimentally shown [4] that specialized algorithms (provided with specialized data structures) have a better performance than database-oriented approaches. On the other side, using external ad-hoc algorithms and exchanging data on demand with the DBMS results both in high context-switching costs and in difficulties in approaching applicative problems (e.g., they usually provide a fixed data mining paradigm, not easily adaptable to requirement changes).

A tradeoff between loose and tight coupling is the Cache-Mine approach [4], which provides a good interface between databases and mining algorithms at the cost of extra storage for caching both the data to be mined and the results. In our implementation, we aim at exploiting such a trade-off. The $\mathcal{LDL}++$ system provides an open architecture, that makes it possible to interface with a variety of external systems and components [8]. This feature reveals very useful for integrating the core language with new functionalities. In particular, it allows the definition of user predicates whose evaluation is demanded to external functions. In our system, we exploit such capabilities to implement the core functionalities of the mining tasks.

7.2 The $\mathcal{LDL}++$ System Prototype

In our experiences, we adopted the $\mathcal{LDL}++$ system [155] a main-memory deductive database system that inherits many important features of its predecessor $\mathcal{LDL}$ [103], and implements most of the extensions described in chapter 3.

Unlike logic programming systems like prolog, $\mathcal{LDL}++$ draws a sharp distinction between facts and rules. Rules are treated as a program, and thus compiled and executed against the database. The target for this compilation is based on relational algebra operators, and use sophisticated techniques for recursion, negation and XY-stratification (see, e.g., [57, 114] for the description of optimization techniques for XY-stratification).

To the purpose of this chapter, we analyze two main features of the $\mathcal{LDL}++$ system. First of all, we analyze the implementation of user-defined aggregates, and show how the basic schema can be improved in order to deal with the iterative extension proposed in chapter 4. Open architecture is another important feature of the $\mathcal{LDL}++$ system, supporting the definition of user-defined predicates and the embedding of complex data structures, such as the hash-tree structures defined in chapters. We shall see how such features are exploited in order to develop the integration described in chapters 5 and 6.
7.2. THE CDL++ SYSTEM

\[
p(K_1, \ldots, K_n, v) \leftarrow \text{readTable}(K_1, \ldots, K_n, A, A, \text{Flag}), \\
\quad \text{if } \text{Flag} = 1 \text{ then} \\
\quad \quad \text{if } (\text{iterate}(\text{aggr}, A, A_0)) \text{ then} \\
\quad \quad \quad \text{multi}(\text{aggr}, A, A_0), \\
\quad \quad \quad \text{writeTable}(K_1, \ldots, K_n, A_0), \\
\quad \quad \quad \text{return}(\text{aggr}, A, A_0, V)) \\
\quad \quad \text{else} \\
\quad \quad \quad \text{return}(\text{aggr}, A, A_0)) \\
\quad \text{else} \\
\quad \quad \text{if } (\text{Flag} = 2 \text{ then} \\
\quad \quad \quad \text{single}(\text{aggr}, A, A_0), \\
\quad \quad \quad \text{writeTable}(K_1, \ldots, K_n, A_0), \\
\quad \quad \quad \text{return}(\text{aggr}, A, A_0, V)) \\
\quad \text{else} \\
\quad \quad \quad \text{multi}(\text{aggr}, A, A_0)) \\
\quad \quad \quad \text{writeTable}(K_1, \ldots, K_n, A_0), \\
\quad \quad \quad \text{return}(\text{aggr}, A, A_0, V))).
\]

Figure 7.1: Extended Rule Rewriting for Iterative Aggregates.

7.2.1 Aggregate Implementation

Aggregation in databases is typically supported in two forms, called scalar aggregates and aggregate functions, i.e., aggregates specifying group-by constraints) [70]. Algorithms for aggregate functions require grouping, so that one output is computed for each group. In order to compute such aggregation functions, an internal table is needed, that stores information concerning groups, intermediate results and final results.

In CDL++, the evaluation of a query containing an aggregate consists in two phases. Consider a typical group-by aggregate rule,

\[
p(K_1, \ldots, K_n, \text{aggr}(A)) \leftarrow \text{Rule body}.
\]

where \(K_1, \ldots, K_n, A\) are derived from the terms in the rule body. The evaluation of the query \(p(v_1, \ldots, v_n, v)\) is done by first compiling the above program, and then executes the query on the compiled program. In the compiling phase, the program can be rewritten as shown in section 3.4. However, in order to improve the efficiency of the computation, we can rewrite the above rule in a slightly different (and actually not declarative) way. Figure 7.1, directly adapted from [159], shows how iterative aggregate rules are compiled by the system. Such a rewriting exploits two built-in predicates, namely \text{readTable} and \text{writeTable}, which retrieve from, and insert rows into the internal table. During the execution phase, the rule body is evaluated, possibly storing resulting tuples in temporary memory. When \text{readTable} is called, it gets values from the results of the evaluation of the rule body and from the internal table.

The \text{Flag} argument in \text{readTable} keeps track of the status of the reading.
• If Flag = 1, all the tuples resulting from the evaluation of the rule body have been considered. In such case, either the execution can be iterated, and the status of the readTable predicate needs to be reset, or we reached the end of the computation, and the final state can be returned as an answer.

• If Flag = 2, the first tuple is considered. The result of the evaluation of the single predicate is inserted into the internal table by means of the writeTable predicate.

• If Flag = 3, we are in an intermediate state. We combine the intermediate result with the current tuple by evaluating the multi predicate, and store the result into the internal table.

The critical point in the evaluation of such a program is the memory management. The tuples resulting from the evaluation of the rule body can be generated either lazily or eagerly. Another key issue is the management of the internal tables performed by the built-in predicates readTable and writeTable. First of all, the storing and retrieval of intermediate results requires accurate management, i.e., efficient garbage collection techniques and efficient access to multiple values. Lazy or eager materialization of such intermediate results has a great impact over the system performance. Moreover, the maintenance of the group-by table has to be done efficiently. There are three main techniques that can be used [70], based on nested loops, sorting and hashing.

• In the nested loop approach, for each input tuple \((k_1, \ldots, k_m, A)\) a linear search for the corresponding value \(A_o\) is performed, using the key \((k_1, \ldots, k_m)\). Obviously, this algorithm is quite inefficient for large inputs. However, such a method is effective and simple to implement when input items are not divided into disjoint equivalence classes, but a single input item may contribute to multiple outputs.

• In the sorting method, items are sorted according to the values of their grouping attributes \((k_1, \ldots, k_m)\). Such an approach is particularly suitable for datasets of large size, where the output results hardly fit into memory.

• The most effective method is based on hashing: the internal table is modeled by a hash table, where each bucket contains a value of an aggregate, and the key for each bucket is \((k_1, \ldots, k_m)\). Since hash table is designed to fit into memory, the above described operations on the hash table are very easy to design and implement, and they outperform the other two methods.

### 7.2.2 User-Defined Predicates

In order to support complex database applications, many major relational database systems support user-defined functions. Such functions can be invoked in queries,
making it easier for developers to implement their applications with significantly greater efficiency. The adoption of such features in a logic-based system reveals even more impact, since they allow a user to develop large programs by hot-spot refinement [33]. The user writes a large LDL++ program, validates its correctness and identifies the hot-spots, i.e., predicates in the program that are highly time consuming. Then, he can rewrite those hot-spots more efficiently in a procedural language, such as C++, maintaining the rest of the program in LDL++.

The formal model for Datalog programs with external predicates is developed in [33], where the notion of computed database is introduced. The computed database is a finite set of infinite relations. For example, <, > and other arithmetic predicates fall into this category. Conceptually, any external procedure, e.g., a program written in C++, can be viewed as a computed predicate, i.e., an infinite relation with some finiteness constraints. Such constraints specify the input/output requirements. A computed predicate p is said to satisfy a finiteness constraint of the form \( \text{X} \rightarrow \text{Y} \) if and only if for each tuple \( \mu \) in p, the set of tuples \( \{ \nu[Y] | \nu \in p \text{ and } \nu[X] = \mu[X] \} \) is finite. The finiteness constraint must be explicitly stated in the case of a computed predicates. This is done by specifying the input arguments X of the procedure corresponding to the computed predicate. It is assumed that this procedure can compute the (finite) set of values for Y in finite time, given a value for X. It is the responsibility of the query evaluator to invoke the procedure only after a value for X is known. This imposes an ordering on the predicate occurrences of a program.

LDL++ allows the definition of external predicates written in C++, by providing mechanisms to convert objects between the LDL++ representation and the external representations. The ad-hoc use of such mechanisms reveals very useful to provide new data types inside the LDL++ model, in the style of Object-relational databases [1]. For example, a reference to a C++ object can be returned as an answer, or passed as input, and the management of such a user-defined object is demanded to a set of external predicates.

**Example 7.1.** A vector object \( V \) is a collection of elements of type float. Such an object defines the operations \( \text{dot}_\text{prod} \) and \( \text{sum} \) that can be implemented as user-defined predicates. For example, the \( \text{dot}_\text{prod} \) predicate can be implemented (in pseudo-code) as follows:

```c
LDL_STATUS dotprod(Ld1Object v1, Ld1Object v2, Ld1Object res) {
    if (first invocation) {
        vector* V1 = ldl_get_int(v1);
        vector* V2 = ldl_get_int(v2);
        int Vres;
        Vres = \sum_i V_{1i} \times V_{2i};
        res = ldl_create_int(Vres);
        return LDL_SUCCESS;
    }
    else
        return LDL_FAIL;
}
```
The vector objects, which represent the input of the relation, are represented in 
\( \mathcal{DL}++ \) as integers. Notice that, since the external predicates may return multiple values, an internal status that keeps track of the invocations is maintained. In such an example, only one value is returned as output, so that only the first call returns the computed value.

As seen in the previous chapters, we can adopt such a model to implement hot-spot refinements. In the following we describe the implementation of an enhanced version of the Apriori algorithm, described in [5], by means user-defined predicates\(^1\).

As described in chapter 5, we use a Hash-tree structure, which is essentially a prefix-tree with an hash table associated to each node. An edge is labelled with an item, so that paths from the root to an internal node represent itemsets. Figure 7.2 show an example tree.

Each node is labelled with a tag denoting the support of the itemset represented by the path from the root to the node. An additional tag denotes whether the node is suitable to generate candidate itemsets. As described in chapter 5, the main operations that can be performed over such a tree structure are exemplified in fig. 7.3:

- Initialization (fig. 7.3 a)). For each item found in the transaction, either the item is already into the tree (in which case its counter is updated), or it is inserted and its counter set to 1. A sample code for the init function performing such an operation is given below:

---

\(^1\)Deeper technical details concerning the implementation shall be omitted. In the following we shall make use of a pseudo-code, by putting in evidence only the parts that are worth being analytically described.
Figure 7.3: a) Tree initialization. b) Pruning. c) Tree enhancement and counting. d) Pruning. e) Tree enhancement. f) Cutting.

```c
LDL_STATUS init(Ld1Object I, Ld1Object T) {
    if (first invocation) {
        HashTree* tree;
        if (T is not created)
            tree = new HashTree();
        else
            tree = ld1_get_int(T);
        foreach e \in I 
            if (tree->root->contains(e))
                (tree->root->counter(e))++;
            else
                tree->root->new_bucket(e,1);
        T = ld1_create_int(tree);
        return LDL_SUCCESS; }
    else
        return LDL_FAIL; }
```

- Counting (fig. 7.3 c)). A simple recursive procedure is defined that, starting from the first element of the current transaction, traverses the tree from the root to the leaves, by either exploiting the current element or ignoring it. When a leaf at a given level is found, the counter is incremented.
void count(vector S, int i, HashTree tree, node e)
    if (tree->leaf(e))
        tree->counter(e)++;
    else
        for (j = i; j < S.size(); j++)
            if (tree->e->contains(S[j])
                count(S, j + 1, tree, tree->e->son(S[j])); }

LDL_STATUS count(Ld1Object S, Ld1Object T){
    if (first invocation){
        HashTree* tree = ldl_get_int(T);
        vector<Ld1Object> set = ldl_get_set(S);
        count(set, 0, tree, tree->root);
        return LDL_SUCCESS; }
    else
        return LDL_FAIL; }

• Pruning (figures 7.3 b) and d)). Leaf nodes at a given depth (representing
    the size of the candidates) are removed if their support is lower than a given
    threshold. The following pseudo-code formally shows the computation:

LDL_STATUS prune(Ld1Object Sp, Ld1Object T){
    if (first invocation){
        HashTree* tree = ldl_get_int(T);
        int support = ldl_get_int(Sp);
        int 1 = tree->get_curr_level();
        foreach leaf node e ∈ tree at level 1
            if (tree->counter(e) < support)
                tree->delete(e);
        return LDL_SUCCESS; }
    else
        return LDL_FAIL; }

• Enhancing. New candidates are generated, in two steps. In the first step,
    a leaf node is merged with each of its siblings, and new sons are generated.
    For example in fig. 7.3 e)), the node labelled by the path beer − chips is
    merged with its sibling beer − wine, generating the new node labelled by
    beer − chips − wine. In order to ensure that every new node represents an
    actual candidate of size n + 1, we need to check whether all the subsets of
    the itemset of size n are actually in the hash tree. Such an operation consists
    in a traversal of the tree from the enhanced node to the root node; for each
    node analyzed, we simply check whether its subtree is also a subtree of its
    ancestor. Subtrees that do not satisfy such a requirement are cut (fig. 7.3 e)).
    The following pseudo-code implements such operations within the enhance
    predicate:
7.3. PERFORMANCE ANALYSIS

LDL_STATUS enhance(LdlObject T) {
    if (first invocation) {
        HashTree* tree = ldl_get_int(T);
        int l = tree->get_curr_level();
        foreach leaf node e ∈ tree at level l {
            tree->enhance(e);
            tree->cut(e); }
        if (tree->alive())
            return LDL_SUCCESS;
        else
            return LDL_FAIL; }
    else
        return LDL_FAIL; }

- Itemset generation. Since each leaf node represents an itemset, generation of itemsets is quite simple. The tree is traversed and itemsets are built accordingly. The itemset predicate implements such operations:

LDL_STATUS itemset(LdlObject T, LdlObject I) {
    HashTree* tree = ldl_get_int(T);
    if (tree->get_leaf(e)) {
        Vector<LdlObject> is;
        LdlObject s = ldl_create_int(tree->support(e));
        do {
            is.add(tree->get_label(e));
            tree->get_next_node(e);
        } while (e != tree->root());
        I = ldl_create_tuple(ldl_create_set(is), s);
        return LDL_SUCCESS; }
    else
        return LDL_FAIL; }

Notice that, differently from the previous predicates, where only one invocation was allowed, the itemset predicate allows multiple calls, providing one answer for each itemset found.

7.3 Performance Analysis

In this section we analyze the impact of the architecture we described in the previous sections to the process of extracting association rules from data. The performance analysis that we undertook compared the effect of mining association rules according to different architectural choices:

1. DB2 Batch, an Apriori implementation that retrieves data from a SQL DBMS, stores such data in an intermediate structure and then performs the basic steps
of the algorithm using such structures. Such an implementation conforms to the Cache-Mine approach. The main motivation is to compare the effects of such an implementation with a similar one in the $\mathcal{LDC}++$ deductive database. Conceptually, such an implementation can be thought of as the architectural support for an SQL extension like, e.g., the MINE RULE construct shown in section 4.2.1.

2. $DB2$ interactive, an Apriori implementation in which data is read tuple by tuple from the DBMS. This approach is very easy to implement and manage, but has the main disadvantage of the high context switching cost between the DBMS and the mining process. Since user-defined predicates provide such a context switching, too, it is interesting to investigate how the approach behaves compared to the $\mathcal{LDC}++$ approach.

3. $\mathcal{LDC}++$, the implementation of the rules mining aggregate patterns, specified in the previous sections, by means of the Apriori algorithm. We adopted a Cache-Mine approach: the pattern aggregate was implemented by means of single, multi and return ad-hoc external predicates (in the same way as described in the previous section), where

- single and multi simply collect the set of relevant tuples in an intermediate cache structure and
- return computes the rules using the cache, and returns the results as usual.

We compared two different versions of the $\mathcal{LDC}++$ system. The first one, referred as $\mathcal{LDC}++$ (nested loop), uses nested loops to implement aggregates, and uses a "lazy" materialization of intermediate results (i.e., all the intermediate results are maintained). The second one uses hashing and implements a "eager" materialization (i.e., intermediate results are deleted as soon as they are consumed). As we shall see, such differences have strong impact in the performance.

In addition, we compared the above solutions with a plain Apriori implementation (Apriori in the following), that reads data from a binary file. We used such an implementation to keep track of the actual computational effort of the algorithm on the given data size when no data retrieval and context switching overhead is present.

We tested the effect of a very simple form of mining query, that retrieves data from a single table and applies the mining algorithm. In $\mathcal{LDC}++$ terms, the experiments were performed by querying $\text{ans} \left( \text{min supp}, \text{min conf}, R \right)$, where $\text{ans}$ was defined as

$$\text{ans} \left( S, C, \text{patterns} \left( \left( S, C, \text{ItemSet} \right) \right) \right) \leftarrow \text{transaction} \left( \text{ID}, \text{ItemSet} \right).$$
and the transaction(ID,ItemSet) relation is a materialized table. Obviously, in the 
\(\mathcal{LCL}++\) system it is possible to define more complex mining queries; however, such
queries are hardly definable (and sometimes even not definable at all) in relational
systems based on SQL. Hence, a comparison of SQL-based systems and Datalog-
Based systems is significant only on equivalent fragments of the languages.

Since the main objective of the experiments is to compare the performances
of the above explained architectural choices, we fixed the confidence of each rule
(the term min\textunderscore conf) to 80, and analyzed the performance on the given datasets for
varying values of the support.

### 7.3.1 Synthetic Data Generation

In order to populate the transaction predicate (and its relational counterpart), we
used the synthetic data generation utility available from [136]. Data generation can
be tuned according to the following parameters [137, section 2.4.3]:

- \(|T|\), the number of transactions
- \(|I|\), the average size of the transactions
- \(|I|\), the average size of the maximal potentially frequent itemsets
- \(|I|\), the number of maximal potentially frequent itemsets
- \(N\), the number of items

The size of a transaction is picked from a Poisson distribution with mean \(|T|\).
Then, frequent itemsets (or fractions of them) are assigned to the transaction. A
frequent itemset in \(I\) is generated by picking its size from a Poisson distribution
with mean \(I\), and then randomly assigning items to it. Notice that there is an
inverse relationship between \(|I|\) and the average support of the frequent patterns.
This means that, for high values of \(|I|\), we must expect a high number of patterns
only on low values of the support.

<table>
<thead>
<tr>
<th>dataset</th>
<th>30</th>
<th>20</th>
<th>10</th>
<th>5</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>T10.I2.D100</td>
<td>26</td>
<td>3</td>
<td>54</td>
<td>97</td>
<td>2000</td>
<td>2181</td>
</tr>
<tr>
<td>T10.I2.D500</td>
<td>6</td>
<td>0</td>
<td>28</td>
<td>6</td>
<td>132</td>
<td>120</td>
</tr>
<tr>
<td>T10.I2.D1K</td>
<td>6</td>
<td>0</td>
<td>28</td>
<td>6</td>
<td>110</td>
<td>135</td>
</tr>
<tr>
<td>T10.I2.D5K</td>
<td>4</td>
<td>0</td>
<td>22</td>
<td>9</td>
<td>98</td>
<td>84</td>
</tr>
<tr>
<td>T10.I2.D10K</td>
<td>4</td>
<td>0</td>
<td>24</td>
<td>9</td>
<td>100</td>
<td>87</td>
</tr>
<tr>
<td>T10.I2.D50K</td>
<td>4</td>
<td>0</td>
<td>22</td>
<td>9</td>
<td>94</td>
<td>63</td>
</tr>
<tr>
<td>T5.I2.D100K</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 7.4: Frequent itemsets and rules over synthesized data.

...
Figure 7.5: Performance Graph.

We fixed \(|I|\) to 2, and \(|\mathcal{I}|\) to 10, since such parameters affect the size of the discovered rules. All the remaining parameters were adjusted according to augmenting values of \(D\): as soon as \(D\) increases, \(|\mathcal{I}|\) and \(N\) are increased as well.

Figure 7.4 describes the datasets used as benchmarks. For each dataset, the table describes the number of frequent itemsets and relevant associations at the given support values. The first dataset is useful mainly to estimate the contribution of the LDL++ implementation to the generation of results, i.e., how performance is affected when the number of generated rules is huge. Datasets with size ranging from 500 to 100K are useful to estimate the scalability of the approach with respect to the size of the dataset. In particular, the last dataset allows to measure the contribution of the tuple retrieval phase from the DBMS (or the LDL++ internal system) in the cache population phase.

### 7.3.2 Performance Evaluation

The architectural choices 1 and 3 are equivalent from a conceptual viewpoint, since they both comply to the Cache-Mine approach. Hence, the significance of the experiments is given by their comparison. From a pragmatic point of view, they should be of the same order of magnitude, and their performances should be similar modulo a (almost) constant factor, representing the different implementation choices for the different architectures. Figure 7.5 shows the expected performance relationship among the various architectural choices.

The following figures show how the performances of the various solutions change according to increasing values of \(D\) and decreasing values of the support. For each experiment, performances are shown both in actual and in log values. Experiments were made on a Linux system with two 400Mhz Intel Pentium II processors, with 128Mb RAM. Alternatives 1 and 2 were implemented using the IBM DB2 universal database v6.1.

From the analysis of figures from 7.6 to 7.11 it can be seen that, as expected, the DB2 (interactive) solution gives the worst results: since a cursor is maintained against the internal buffer of the database server, the main contribution to the cost
is given by the frequent context switching between the application and the database server [4]. Moreover, decreasing values of the support parameter strongly influence its performance: lower support values influence the length of the frequent patterns, and hence multiple scans over the data are required.

Figures from 7.6 to 7.11 show that the \textit{LDL++ (hash)} outperforms the \textit{DB2 (Batch)} approach. However, as soon as the size of the dataset is increased, the difference between the two approaches tends to decrease: the graphs show that the \textit{LDL++ (hash)} performance gradually worsens, and we can expect that, for larger datasets, \textit{DB2 (Batch)} can outperform \textit{LDL++ (hash)} Such a behavior finds its explanation in the processing overhead of the deductive system with respect to the relational system, which can be quantified, as expected, by a constant factor\(^2\).

The \textit{LDL++ (nested loop)} approach, instead, shows that performances are quite similar in graphs 7.6, 7.6, 7.7 and 7.8. For high values of \(D\) (i.e., with more than 10,000 transactions) the performance of the system shows a significant worsening. The first graph in fig. 7.14 summarizes the performance of the \textit{LDL++} system for

\(^2\)We must mention, however, that such limitations are also influenced by the current implementation of aggregates in \textit{LDL++}. A new version of the system is being developed with better aggregate performance [153].
Figure 7.8: $D = 5.000$, $N = 3.000$, $|I| = 100$.

Figure 7.9: $D = 10.000$, $N = 3.000$, $|I| = 100$.

Figure 7.10: $D = 50.000$, $N = 5.000$, $|I| = 100$. 
different values of the data size. In particular, the second graph shows the behavior of the system when the support is set to 3%. The system shows a quadratic behavior. Notice that, by the contrary, the performance graph of the $\mathcal{L}DL++$ (hash) approach in fig. 7.13 has a smoother (almost linear) curve. The graphs of fig. 7.15 provide a curve-fitting of the performances of the two approaches, in which the linear vs. quadratic behavior is revealed.

The difference between the two approaches is even more evident in figure 7.11 where, except for the lowest value of the support, the cost of the computation is mainly given by the data scan. Here, only low supports influence the result values (i.e., rules are generated only with support $\leq 1\%$), and hence the contribution of the mining algorithm to the overall cost is not relevant. The ratio between the data preprocessing of $\mathcal{L}DL++$ and the application of the Apriori algorithm is shown in figure 7.12. The ratio is 1 when the internal management phase is predominant with respect to the application of the algorithm.

To conclude, it is interesting to observe the contribution of the result coding phase to the performance of the approach. In such an implementation, a rule is represented as a complex term of the form $\text{rule}(\text{lhs}, \text{rhs}, \text{support}, \text{confidence})$. For example, an answer of the $\mathcal{L}DL++$ system to the $\text{ans}(30, 80, R)$ is given by the
Figure 7.13: Performance of the hash approach (w.r.t. the data size).

Figure 7.14: Performance of the nested-loop approach (w.r.t. the data size).

predicate \texttt{ans(30, 80, rule({beer}, {chips}, 5, 100))}. The coding of such a result has an influence to the overall cost, as shown in fig. 7.16, where the performance of the \texttt{LDLC++} approaches have a sensible performance worsening, even over a very small dataset. Such a behaviour is mainly due to the generation of a very high number of rules generated, as can be seen in fig. 7.4.

### 7.4 A Discussion on Optimizations

The implementation shown in the previous sections is a first step towards a significant integration between inductive tasks and deductive tasks. As a prototype, however, it suffers of several drawbacks.

- **Scalability** is the first issue. The benchmarks have shown a progressive (linear) performance worsening related to increasing data sizes. In order to make the system scalable, we need to augment the verticality of the performance line. Moreover, the \texttt{LDLC++} system is mainly a main-memory system, so that whenever the blocks of memory devoted to store relations are full, swapping strategies must be adopted in order to deal with larger tables. This obviously
7.4. OPTIMIZATIONS

Figure 7.15: Curve fitting of the performances of the \( \mathcal{LDL}++ \) approaches.

Figure 7.16: \( D = 100, \ N = 100, \ |z| = 100, \ |T| = 5 \).

can influence the system performance considerably.

- An issue that raises from the proposed implementation is the optimization of the execution of user-defined predicates. As an example, it is well-known from the literature [5, 137] that the count phase is by far very expensive. Thus, the compiler has to provide an optimized execution plan, in order to ensure that the user-defined predicate implementing such phase is called with the minimum impact over the overall performance [30].

- The by far most important problem is concerned with optimizing query evaluation. The application of a mining algorithm can be by itself a costly operation, and such cost reflects on its implementation as an aggregate. The evaluation of more complex rules defined using aggregate predicates has to be studied in deep, since it is worth substantial optimization [83, 116]. For example, if we consider the rule

\[
p(S, \text{patterns}(S, I)) \leftarrow r(T, I).
\]

where the \( r \) relation has a huge extension, the evaluation of the query \( p(S, R, C) \) is expensive, no matter how efficiently we implement the aggregate. However,
let us suppose that $p$ is used to define the following predicate:

$$q(L, S_1) \leftarrow p(0.001, \{L, R\}, S_1), S_1 > 0.5, s(R, \_).$$

There are many important considerations concerning the evaluation of $q$:

1. In a way similar to traditional constant pushing techniques, we can move the constraint $S_1 > 0.5$ in the evaluation of $p$, since it is less expensive to evaluate $p(0.5, \{L, R\}, S_1)$.

2. The predicate $p(0.5, \{L, R\}, S_1)$ is worth further optimizations, since it requires only itemsets of size 2. This corresponds to a better tuning of the compiler, that should be able to recognize such situations and rewrite the rule defining $p$ into simpler, more efficient rules:

$$\text{tmp}_p(V, \text{count}(S)) \leftarrow r(T, I), \text{member}(V, I).$$

$$\text{tmp}_p\_2(S, L, R, \text{count}(T)) \leftarrow \text{tmp}_p\_1(L, C_1), \text{tmp}_p\_1(R, C_2),$$

$$C_1 > S, C_2 > S, r(T, I), \text{member}(R, I), \text{member}(L, I).$$

$$p(S, \{L, R\}, S_1) \leftarrow \text{tmp}_p\_2(S, L, R, S_1), S_1 > S.$$  

3. The join between $p$ and $s$ can be moved in the evaluation of $p$, allowing a further performance gain. For example, the second rule can be rewritten as

$$\text{tmp}_p\_2(S, L, R, \text{count}(T)) \leftarrow \text{tmp}_p\_1(L, C_1), \text{tmp}_p\_1(R, C_2),$$

$$s(R, \_), C_1 > S, C_2 > S, r(T, I), \text{member}(R, I), \text{member}(L, I).$$

To summarize, the originary program

$$p(S, \text{patterns}((S, I))) \leftarrow r(T, I).$$

$$q(L, S_1) \leftarrow p(0.001, \{L, R\}, S_1), S_1 > 0.5, s(R, \_).$$

can be rewritten, during the compilation phase, into the more efficient program

$$\text{tmp}_p(V, \text{count}(S)) \leftarrow r(T, I), \text{member}(V, I).$$

$$\text{tmp}_p\_2(S, L, R, \text{count}(T)) \leftarrow \text{tmp}_p\_1(L, C_1), \text{tmp}_p\_1(R, C_2),$$

$$s(R, \_), C_1 > S, C_2 > S, r(T, I), \text{member}(R, I), \text{member}(L, I).$$

$$p(S, \{L, R\}, S_1) \leftarrow \text{tmp}_p\_2(S, L, R, S_1), S_1 > S.$$  

$$q(L, S_1) \leftarrow p(0.5, \{L, R\}, S_1).$$

that is worth further traditional deductive databases optimizations, such as the ones described in [146, 158, 1, 57].
7.4. OPTIMIZATIONS

The above examples show that a thorough modification of the underlying logic abstract machine needs to be investigated. Other interesting ways of coping with efficiency can be investigated as well. An interesting way, for example, is that of identifying a subset of relevant features that can be transferred into more specialized languages, that allow more efficient implementations [93, 102].

In this respect, it is interesting to notice that similar approaches to the formalization of user-defined aggregates in relational query languages allow the direct specification of data mining tasks in SQL. For example, the AXL approach [147] has many similarities with the approach proposed in this thesis (and indeed AXL is a successful attempt to introduce user-defined aggregates in SQL, by adopting both procedural and declarative data manipulation).

There are some significant differences between the approach of this thesis and the AXL approach. For example, the possibility to express recursive aggregates is very powerful in AXL. Many algorithms, such as the ChiMerge algorithm or decision-tree learning algorithms, are easy to specify without exploiting external user-defined predicates. From the opposite side, however, the use of procedural features, such as updating an internal table or checking for the status of the interpreter (i.e., keeping track of the executions that are actually computed) give raise to too complex formalizations, while instead the adoption of external predicates, with a clear meaning and a clear input/output interface, still maintains a simple semantics. Moreover, in the AXL approach it is difficult to exploit ad-hoc optimizations provided by complex data structures, such as the hash-tree structure in frequent patterns discovery.

Notwithstanding, the AXL approach shows that specialized languages for mining/olap tasks could benefit of even a subset of the features of a logic database language, easy to implement in an efficient way (such as, for example, the mechanism of rules for describing the process, or standard input/output interfaces for the interaction between mining and querying).
Chapter 8

Concluding Remarks

8.1 Summary

The main purpose of flexible knowledge discovery systems is to obtain, maintain, represent, and utilize high-level knowledge. This includes representation and organization of domain and extracted knowledge, its creation through specialized algorithms, and its utilization for context recognition, disambiguation, and needs identification. Current knowledge discovery systems provide a fixed paradigm that does not sufficiently supports such features in a coherent formalism. On the contrary, logic-based databases languages provide a flexible model of interaction that actually supports most of the above features in a powerful, simple and versatile formalism. This motivated the study of a logic-based framework for intelligent data analysis.

The main contribution of this thesis is the systematic development of a logic database language with elementary data mining mechanisms to model extraction, representation and utilization of both induced and deduced knowledge. In the development of this thesis, we followed three main directions.

- The Datalog++ logic database language has a formal iterated fixpoint semantics that makes the language viable for efficient implementation. Its formal foundation accommodates temporal, nonmonotonic and nondeterministic reasoning. We showed how such distinguishing features are amenable to model complex applications.

- In particular, Datalog++ allows to define iterative user-defined aggregates. Aggregates provide a standard interface for the specification of data mining tasks in the deductive environment: i.e., they allow to model mining tasks as operations unveiling pre-existing knowledge. Iterative aggregate have the advantage of allowing the specification of such data mining tasks at the desired abstraction level: from a conceptual point of view, they allow a direct use of background knowledge in the algorithm specification; from a physical point of view, they give the opportunity of directly integrating proper knowledge extraction optimizations. We used such main features to model a set
of data mining primitives: frequent pattern discovery, Bayesian classification, clustering and discretization.

- We studied an in-depth implementation of the above described features, in order to provide an efficient run-time support for the resulting language. We provided an implementation of the above aggregates by modeling the most computationally intensive phases by means of hot-spot refinements, i.e., specialized data structures and user-defined predicates, which efficiently accomplish such phases. As a result, we compared the effectiveness of the approach with respect to standard approaches in literature, demonstrating that the underlying system provides a viable trade-off between expressiveness and efficiency.

The resulting system has been successfully applied to real-world case studies.

- In the Datasiift project [58, 65, 59], we transferred the results of our research towards a real-life application: market-basket analysis of supermarket sales data. A prototype system has been realized that supports high-level business-related analyses of the raw data deriving from transactions.

- In the Fiscal Fraud Detection project [54, 65, 16] we have shown how the language allows the integration and interaction of advanced data mining tasks, thus allowing the definition of new analysis schemes that strengthen the traditional methods aimed at fiscal fraud detection.

- Moreover, the approach revealed very fruitful even in domains, such as web mining [17], that for their complexity allow the usage of the system only for prototyping purposes, leaving the actual deployment of the resulting application to procedural approaches.

8.2 Open Problems and Further Research

There are some issues that are worth further investigations in the approach that we proposed. We can devise three orthogonal research lines: **languages and models**, **systems** and **applications**.

**Languages and models.** A very important issue is the practical complexity of correctly specifying iterative aggregates. Implicit recursion, combined with complex types manipulation, can cause aggregates that actually specify non-terminating computation, or produce incorrect results. However, the static structure of an iterative aggregate, i.e., its dependence upon the single, multi and iterate user-defined predicates, makes it particularly easy to define a structured approach, in the style of the approaches shown in [128, 15], in order to prove i) termination, and ii) correctness.

Another issue worth further study is an orthogonal logic-based approach, in which mining problems can be represented by means of constraints, and mining
algorithms can be thought as constraint solvers [22]. The constraint logic programming approach, extensively studied both from a semantic viewpoint and an application-oriented viewpoint [92], has as a practical advantage the capability to deal with practical problems within a formal framework. The approach still preserves the declarative nature of its underlying logical model (being a semantically conservative extension of the declarative paradigm), and allows a clear separation between modeling and resolution. Moreover, it can provide a natural way to tune the approach in order to deal with different types of data in an coherent formalism (as an example, many kinds of spatial data are modeled by means of constraints [96]).

Finally, it is interesting to provide representation metaphores for further mining tasks, in order to make the approach well-suited for larger classes of applications. Providing suitable representations for data characterization or time series in terms of iterative aggregates, are examples of extensions that may complete the sets of mining primitives dealt by the language.

**Systems.** Tailoring of optimization techniques to mining queries is a major research topic in each database-oriented approach: it should be no surprise that such a topic is even more substantial in deductive-based approaches, like the one developed in this thesis. In section 7.4 we have shown some examples of how a logic based language can be benefit of a thorough modification of the underlying abstract machine, and how other interesting ways of coping with efficiency can be investigated (for example, by extracting expressive enough subsets viable for efficient implementation). Such modifications have to be formally defined, in order to provide a mappings of deductive mining query specifications to query plan generations and optimizations, both logical and physical.

**Applications.** From an application viewpoint, the definition of a data mining query language is the first step towards a systematic program development that answers the application needs. Furthermore, it is very important to define substantial interaction mechanism, useful composition strategies and suitable knowledge representations, aimed at modeling different classes of applications. Clearly, different application problems influence different choices: for example, the inductive logic programming approach raises from the need to mine some kind of knowledge that traditional knowledge discovery tools are not able to represent and/or extract. Application-oriented knowledge discovery support environments capable of dealing efficiently and effectively with specific application needs are in this respect more attractive than general-purpose query languages, and are worth an in-depth study.
Bibliography


[33] CHIMENTI, D., GAMBOA, R., AND KRISHNAMURTHY, R. Towards an Open Architecture for LDC. In *Proc. 15th Int. Conf. on Very Large Data Bases (VLDB89)* (1989), pp. 195–204.


[77] Han, J., and Fu, Y. Discovery of Multiple-Level Association Rules from Large Databases. IEEE Trans. on Data Knowledge and Engineering 11, 5 (1999), 798–804.


[80] Han, J., Lakshmanan, L., and Ng, R. Constraint-Based, Multidimen-
sional Data Mining. COMPUTER 32, 8 (1999), 46–50.

Cubes Efficiently. In Proc. ACM Conf. on Management of Data (SIGMOD96)

[82] Haykin, S. Neural Networks: A Comprehensive Foundation. Prentice Hall,
1999.

[83] Hellerstein, J. Optimization Techniques for Queries with Expensive Meth-

[84] Hoaglin, D., Mosteller, F., and Tukey, J. Understanding Robust and


[86] Holseimer, M., and Siebes, A. Data mining: The search for knowledge

[87] Huang, Z. Extensions to the k-Means Algorithm for Clustering Large Data
Sets with Categorical Values. Journal of Knowledge Discovery and Data Min-
ing 2, 3 (1998), 283–304.

[88] Hussain, F., Liu, H., Tan, C., and Dash, M. Discretization: An enabling
technique. Tech. Rep. TRC6/99, School of Computing, National University of

[89] Imielinski, T., and Mannila, H. A Database Perspective on Knowledge

[90] Imielinski, T., and Virmani, A. MSQL: A Query Language for Database
Mining. Journal of Knowledge Discovery and Data Mining 3, 4 (1999), 373–
408.

[91] Imielinski, T., Virmani, A., and Abdulghani, A. DMajor - Application
Programming Interface for Database Mining. Journal of Knowledge Discovery
and Data Mining 3, 4 (1999), 347–372.

of Constraints Logic Programming. Journal of Logic Programming 37, 1–3


