CHAPTER 2. DATA MINING USING P-TREE RELATIONAL SYSTEMS

Data mining differs from machine learning in its concern with data structure issues. Most data are stored in table format. Those tables may be part of a relational database or stored as delimited text or image files. Many choices can be made with respect to these tables, one of the most important ones being whether they will be stored in a row-based or column-based format. Work on P-trees [1-3] takes the column-based viewpoint to the limit by not only breaking up tables into one column per attribute but rather one column per bit.

2.1. Rows versus Columns

Tables are the basis for most data storage. Relational databases gain much of their success from the simplicity of manipulating relations. Relations are commonly implemented as tables; i.e., row and column order is considered meaningful. Tables can be seen as data with a two-dimensional structure. Computer memory, however, is inherently one-dimensional. Therefore, mapping must be made, and in most cases, priority is given to rows. Before discussing this choice, it may be worthwhile to determine if the row and column concept is uniquely defined. It could be that one person's row is another's column in which case the debate would be meaningless. This ambiguity does not, however, exist for most data. We commonly expect that rows are very similar to each other, each consisting of the same combination of attributes, possibly with some data missing. Columns, on the other hand, may differ in their data type and domain. In a table of employees with ID, name, and department number, there is no doubt that those three attributes would be considered as columns and individual employee instances as rows.

Why is row-based storage so prevalent in this setting? Clearly, it is more likely that a new employee record will be inserted than that a new attribute will be added. It may also be more likely that the entire employee record has to be retrieved than a listing of, for example, all employee names. These benefits should not, however, obscure the benefits of a column-based storage. Columns hold data of a single data type. If the data in a column are stored in sequence, compression can be used more easily; processor optimization can take effect in computations; and indexes can be implemented more efficiently. Furthermore, some columns may refer to rarely used data and may not be needed in most transactions or data mining operations. The database community is well aware of the benefits of what is called "vertical partitioning" [4,5]. It is clear that the advantages of column-based storage depend strongly on the number of records in a table. For an employee table with 10 rows in a transactional database, the simplicity of record insertion, deletion, and retrieval in a record-based system is likely to outweigh any potential benefits of column-based storage.

The situation changes when thousands or millions of records are concerned and, in particular, when updating is no issue such as in data warehouses. The problems in data warehouses resemble, in some respects, engineering and science problems in which collective properties are of interest rather than the explicit data related to a particular record. Examples of collective properties in engineering and science are equilibrium flow-patterns in fluid dynamics simulations and equilibrium molecule configurations in molecular dynamics simulations. Problems that are discussed in the data-mining context can often be phrased in very similar terms. Association rules, such as "If milk is in a market basket, butter is likely to be in it, too," refer to the collection of all or a subset of rows. They are not usually evaluated with respect to the individual transactions that contributed to the rule. Information on individual customers and their transactions that would make record-retrieval relevant is often not available or not of interest. Lazy kernel-density-based classifiers that are considered in this thesis fit the same pattern. They answer questions related to a particular unknown sample, such as "Is the yield at the site of an image with a given color expected to be high based on similar entries in the database?" Again, we are not interested in the identity of the entries that are considered similar but, rather, in their collective evidence on the unknown sample. This collective evidence is commonly represented by a "count," i.e., the number of rows that satisfy a condition. Counts will, therefore, play an important role in further development.

The scientific and engineering communities have always preferred column-wise storage, which benefits vectorization and SIMD architectures. The first major scientific programming language, FORTRAN, stored multi-dimensional arrays by column. Most modern programming languages use row-based representations. Even the object-oriented programming methodology can be seen as a generalization of a row-based approach in which objects correspond to rows in a table that is defined by the class definitions. We will discuss this issue in more detail in Section 2.5, but first, we will take column-based storage one step further by breaking up columns into bits.

2.2. Columns of Bits

In the last section, we mentioned compression, hardware optimization, and simplicity of constructing indexes as well as flexibility with storing only the much accessed columns in memory as reasons to store data in a column-based format. Any one of these advantages becomes more pronounced in a bit-wise representation. All data in a computer are stored as bits. Any fixed-length data type that is represented by *b* bits can trivially be decomposed into *b* bit sequences. These sequences can be stored as bit-sequential files (bSQ). Differences between data types are only important with respect to operations such as distance evaluation that will be explained in the next section.

The potential for compression is a major advantage of a bit-wise scheme. In many settings, continuity considerations suggest that adjacent rows are likely to have some matching bit values for some of the attributes. Images, for example, will likely have adjacent pixels with similar color values. If the data show no inherent continuity, appropriate sorting may be considered as an option. The details of these ideas will be discussed in Chapter 3. Different compression techniques are applicable, such as run-length compression or hierarchical compression as in P-trees. It is important to ensure that the compression scheme still allows for fast bit-wise Boolean operations. When dealing with a bit-sequential format, it must be possible to use sequences of bits in place of individual bits in essentially the same operations that we commonly use on binary numbers. This thesis will assume the hierarchical compression of P-trees that is discussed in Chapter 3.

Hardware optimization is another motivation for a bit-sequential representation. Finding those rows that match a particular attribute value, *a*, is equivalent to a Boolean AND operation. Two binary numbers are identical if their bit values are identical for all bit positions. Correspondingly, we must check for all bits if the sequence is equal to the i^{th} bit in *a,* denoted a_i, and combine the result through a Boolean AND. In practice, we check a bit-sequence for equality with 1 by interpreting its 1-values as true (basic P-tree), and for equality with 0 by interpreting 1-values as true and taking the bit-wise complement of the sequence (or complement P-tree). Consequently, the only bit-wise operations that have to be performed over bit sequences to check for equality with a constant are AND operations. Such AND operations are frequently used in video and image processing, and have therefore been a focus of hardware optimization efforts. Single Instruction Multiple Data, SIMD, technology is one example of hardware optimization that can be used for massive ANDs.

A further point in favor of column-wise storage is the simplicity of constructing indexes. The hierarchical nature of P-trees that are used in this thesis allows interpreting them as indexes without further effort. In the context of this thesis, we do not try to access information by row and, therefore, use the index quality of P-trees only to derive counts of data points with matching properties. The use of indexes, however, has a potential for database operations that is more general.

Finally, observe that column-based storage allows loading only the frequently used columns into memory. It will be shown that appropriate intervalization can limit the need to access low-order bits. A bit-wise storage organization can, therefore, be used to further reduce the necessary data in memory.

2.3. Distance Measures and Columns of Bits

In the previous section, we ignored the question of what data types the attributes represent. We were able to do so because we were only concerned with checking for equality in bit-patterns. Data mining algorithms do, however, often rely on more specific distance measures. The main data types we have to distinguish are integers, data types that can be mapped to integers, floating point numbers, and categorical values. For integers, we can continue using the same arithmetic on bit sequences as we do on individual bits. If we choose different distance functions, such as the HOBbit distance [3], we will do so for efficiency reasons. A bit-wise organization of data makes it desirable to approximate similarity between two integers by the maximum number of higher-order bits that have no mismatches. Sections 5.2.2 and 6.2.3 will discuss the HOBbit distance in detail.

An example of a data type that can be mapped to an integer is type string. Strings are defined to have a unique ordering, lexicographical ordering, i.e., the ordering that is used in a dictionary, but it is not identical to the order of integers that represent the strings. One main difference is that lexicographical ordering does not distinguish between upper- and lowercase characters, whereas the integer representation of strings does. There are many other minor differences, such as the treatment of apostrophes. In this thesis, the domain of existing strings was always small and their order irrelevant. Therefore, an encoding as categorical data, see below, was chosen.

Categorical data, i.e., data with no ordering, can only have one of two distances: 1 (attribute values different) and 0 (values the same). Categorical data can be labeled using consecutive numbers, called labels, that are implemented as integers but interpreted differently. If labels are to be described in a framework of distance metrics, each bit must be represented by one integer number in the interval [0,2). The distance between two numbers is then the result of the MAX metric [6] as applied to the individual bit positions, corresponding to a representation in which the individual bit positions are treated as dimensions in a $\lceil \log n \rceil$-dimensional coordinate system, where $\lceil x \rceil$ denotes the ceiling function. The bit-value of each bit position is considered as an integer number that can be

0 or 1. The MAX metric evaluates any distance between points in this representation to be 1. The distance between any two attributes will, therefore, be the same unit distance. For an example of a categorical attribute with a domain of size 4, the representation becomes 00, 01, 10, and 11. It can easily be seen that the MAX metric distances between any two of these numbers is 1 since any two of them differ by exactly 1 in at least one of the bit positions.

It is worthwhile discussing a special case in which attributes can have many categorical values, i.e., multi-valued attributes. If an attribute can have more than one value, we normally create more than one table with the goal of normalizing the database. Alternatively, we can consider each domain value of the multi-valued attribute as one Boolean attribute. Note that this representation leads to n Boolean attributes for a domain with n elements, compared with $\lceil \log n \rceil$ for a single-valued attribute as we saw in the previous paragraph.

Floating point numbers can also be used in bit-sequence-based calculations. In this thesis, floating-point numbers are always mapped to integers by multiplication with a constant scaling factor and rounding. Occasionally, other transformations are applied first, such as taking the logarithm of the data.

2.4. Concept Hierarchies and Concept Slices

A bit-sequential representation naturally suggests looking at individual bits as separate attributes and discussing their relationship in the data-mining context. A commonly discussed relationship between attributes is a concept hierarchy. Concept hierarchies are defined as a sequence, or more generally, a directed graph, of mappings from a set of low-level concepts to more general concepts, such as "city" < "province_or_state" < "country" [7]. Bit levels in integers clearly do not satisfy such a condition. We must, instead, combine all higher-order attributes to get a concept hierarchy. For binary integer number 1011, the concept hierarchy would be "four bits 1011" < "three high order bits 101" < "two high order bits 10" < "highest order bit 1"; i.e., each bit individually is not part of a concept hierarchy. We will call each bit a concept slice. Concept slices can also be identified among other attributes. Attributes "year," "month," and "day_of_month" can be seen as concept slices. Defining a concept hierarchy for these attributes requires combining attributes ("year","month","day_of_month") < ("year","month") < "year". Note that the concept hierarchy that was developed for integer data can also be seen as successive intervalization. Efficient data mining on numerical attributes normally requires values within some interval to be considered together. In data mining, it is often useful to consider a variety of levels with different interval widths leading to a concept hierarchy based on intervals of integer-valued attributes. Using the concept slices, i.e., individual bit sequences, we can map out a concept hierarchy, i.e., hierarchy of intervalizations, by bit-wise "AND" operations. At the highest level, membership in the components is entirely determined by the highest-order bit. For lower levels, the successive bit values represent differences; i.e., the lower-order bits do not constitute a lower level in a concept hierarchy by themselves, but rather represent changes with respect to the next higher level. The important aspect of going to the finest level of differences, bits, is that we can use a standard bit-wise AND to describe intervals at every level. The bit sequences that are produced by a multi-way AND of all bit-levels equal to or higher than some given resolution can be interpreted, in database language, as bitmap indexes for data points with attribute values within the represented interval.

It is important to note that, although we break up both integer and categorical attributes into bit columns, we do so with different motivation. For integer attributes, the individual bits are considered concept slices that can be used within a framework of concept hierarchies. Knowing which bit position is represented is essential for the correct evaluation of distance, means, etc. For a categorical attribute that is represented by labels, the individual bits are considered equivalent. Subsets of the full label cannot be combined to form a concept hierarchy; i.e., the bit positions do not represent concept slices. Several categorical attributes can, however, collectively define a concept hierarchy.

2.5. Partitions

A general concept that is suitable to describe the potential of P-trees is the partition. A partition is a mutually exclusive, collectively exhaustive set of components. One possible basis for the partitioning of a table is the value of one or more attributes. In database systems, such a partition is often implemented as an index, i.e., a table that maps from the attribute value to the tuples of the partition component. A common reason to implement an index is to provide a fast access path to components of the partition. Data warehouses use bitmap indexes for data mining queries. Bitmap indexes are bit vector representations of regular indexes based on the value of an attribute. The result of a query is represented by a bitmap that partitions a table into rows that are of interest, labeled by the value 1, and those that are not, labeled by 0. Bitmap indexes are very much like P-trees, but are usually uncompressed and are stored in addition to rather than replacing the data. Many data mining algorithms use tree-structured indexes to represent hierarchical partitions. Examples of such indexes are B+-trees, R-trees [8], Quad-trees [9], and P-trees [1-3]. Not every partition has to be implemented using an index. While an index always defines a partition, defining a partition without an index on it may well be useful. In a record-based database, a "select" query creates a partition of the table into rows that satisfy a given condition and those that do not but will not produce an index to those rows.

In the context of the classification techniques in this thesis, partitioning is used to distinguish points that can provide evidence for a test sample from those that cannot. The cardinality of the component of relevant training points allows determining the most likely class label. The indexing property of P-trees could, furthermore, be used to retrieve those relevant points or to store the result set, leading to additional potential for P-trees.

2.6. Column-based Structures and Software Engineering

The row-versus-column question extends beyond the vertical-versus-horizontal partitioning debate in database systems. The major current programming paradigm is object-oriented programming. Object-oriented programming is closely related to the record-based storage concept employed in most current databases. The Unified Modeling Language, UML, is suitable to the modeling of entities and relationships in databases while being equally appropriate to classes in object-oriented program design. The design of object-oriented systems is often closely related to the design of the underlying databases [10] although there may not be a one-to-one mapping between classes and relational database tables for any given system. What can be observed, however, is that the structure of classes that define attributes, and have objects as their instances, is not unrelated to the structure of tables that also define attributes and have records that form the extension. Trying to implement a column-based storage in an object-oriented system is non-trivial. The implementation has to guarantee that rows are added and deleted such that columns remain consistent. There must be ways of efficiently defining and executing non-trivial functions on columns. Storage and functions on columns of attribute values have to benefit from hardware optimization. Some of these issues have been addressed through generic programming. There does not, however, appear to exist a model for integrating the vertical partitioning concept systematically into an object-oriented design.

Engineers and scientists often respond to these problems by abandoning object-oriented design entirely. The simple reason given is commonly that object-oriented programs are too slow. This statement is based on the assumption that any data in table format have to be mapped naively with each row represented by one object. There is little doubt, however, that such a mapping is not efficient for tables with thousands or millions of rows. If each row is represented by one object and a method is to be called for each row, this design results in thousands or millions of method calls for a single iteration through the data. Object-oriented programming can still be productively employed for the purpose of data mining or scientific computing. It is not unusual at all in object-oriented programming to use classes and objects in unconventional ways that have been shown to work in one particular type of setting. Such special uses are called design patterns and are considered a major strength of object-oriented design [11]. Design patterns are often valuable even if they violate standard guidelines for object-oriented modeling. One example is the singleton design pattern in which only one object can exist for a particular class. This design pattern violates the more general guideline that something should only be considered a class if it will have multiple objects. By the same reasoning, it must be possible to formulate a design pattern that implements column-based storage efficiently. In this thesis, we address the special case of a data structure with bit-wise columns.

Why is it important to integrate column-based data structures into object-oriented design if it may be easier to define such data structures in other programming languages? The answer to this question is that programs will not, normally, consist only of classes for which column-based storage is beneficial. The vast majority of classes will benefit from a row- or object-based organization just as in any other object-oriented program. In our data mining programs, only one table is data mining relevant and must be stored column-wise. Sacrificing the benefits that come from object-oriented design of the program as a whole can have a serious impact on reusability. Large programs in engineering and science that are written in procedural programming languages are often very hard to maintain and extend. The problems are not limited to the "old" disciplines as was discussed in Section 2.1. Data mining problems are very similar in their requirements on data structures. Section 3.4 discusses how the P-tree API and implementation can be seen as an example of column-based object oriented design.

2.7. References

[1] Q. Ding, M. Khan, A. Roy, and W. Perrizo, "P-tree algebra," ACM Symposium on Applied Computing (SAC'02), Madrid, Spain, 2002.

[2] Q. Ding, Q. Ding, and W. Perrizo, "Association Rule Mining on Remotely Sensed Images Using P-trees," Pacific-Asian Conference on Knowledge Discovery and Data Mining, PAKDD-2002, Taipei, Taiwan, May 2002.

[3] M. Khan, Q. Ding, and W. Perrizo, “K-nearest Neighbor Classification of Spatial Data Streams Using P-trees,” Pacific-Asian Conference on Knowledge Discovery and Data Mining, PAKDD-2002, Taipei, Taiwan, May 2002.

[4] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou, "Vertical Partitioning Algorithms for Database Design," ACM Transactions on Database Systems, Vol. 9 No. 4, pp. 680-710, 1984.

[5] C. Fung and K. Karlapalem, "An Evaluation of Vertical Partitioning for Query Processing in Object-oriented Databases," IEEE Transactions on Knowledge and Data Engineering, March 2001.

[6] K. Yoshida, "Functional Analysis," 3rd Edition, Springer Verlag, Berlin, 1971.

[7] J. Han and M. Kamber, "Data Mining: Concepts and Techniques," Morgan Kaufmann Publishers, San Francisco, 2001.

[8] A. Gutman, "R-trees: A Dynamic Index Structure for Spatial Searching," Proceedings ACM SIGMOD Conference on Management of Data, Boston, 47-57, 1984.

[9] R. A. Finkel and J. L. Bentley, "Quad-trees: A Data Structure for Retrieval on Composite Keys," ACTA Informatica, Vol. 4, pp. 1-9, 1974.

[10] A. Dennis, B. Wixom, and D. Tegarden, "System Analysis and Design-An object Oriented Approach with UML," Wiley Publishers, New York, 2002.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-oriented Software," Addison Wesley, Reading, Massachusetts, 1995.