CHAPTER 3

P-TREES: CONCEPTS, IMPLEMENTATION, AND APPLICATION PROGRAMMING INTERFACE

3.1. Concepts

Most storage systems view data as a collection of tables. These tables can be stored row by row as is done in the common record-based storage formats. Chapter 2 motivated a column-wise storage format in which tables are broken up into columns and columns are further broken up into individual bit positions. Each bit position can be considered a bit vector. The bit vectors can be seen as indexes to records that have the corresponding bit set. Identifying records that correspond to a particular attribute value or collection of attribute values, in this setting, requires a bit-wise AND operation on all the bit vectors involved. The bit vectors are likely to contain long sequences of 0 or 1 values. We, therefore, use a compressed format, the P-tree format. P-trees were initially designed for spatial data that show homogeneity due to the spatial continuity of the data [1]. Multimedia data also show homogeneity in the time dimension [2]. Homogeneities in data can occur for other reasons. Join operations in databases lead to replication of some table entries. Depending on the join algorithm, some these replicated entries appear in sequence and can be compressed in a bit-column-wise storage. Sparseness of 1 values furthermore leads to long sequences of 0 values [3]. For data that do not show any homogeneity, a sorting scheme that improves compression of P-trees significantly is introduced. We look at the creation of P-trees as a two-step process in which we first choose an appropriate ordering of records, as explained in Section 3.1.1, and then break up columns into bit vectors and compress them by eliminating pure quadrants; see Section 3.1.2.

3.1.1. Choosing an Ordering

P-trees gain their compression potential from bit-subsequences that are entirely composed of 0 or 1 bits. In image data, spatially close pixels are likely to be similar in other properties because they often belong to the same object or natural environment. It is important to maintain the property of spatial closeness when mapping the two-dimensional structure space to the one-dimensional P-tree representation. Many space-filling curves have been suggested with the goal of maintaining continuity when mapping *n* dimensions to one; see Figure 3.1.

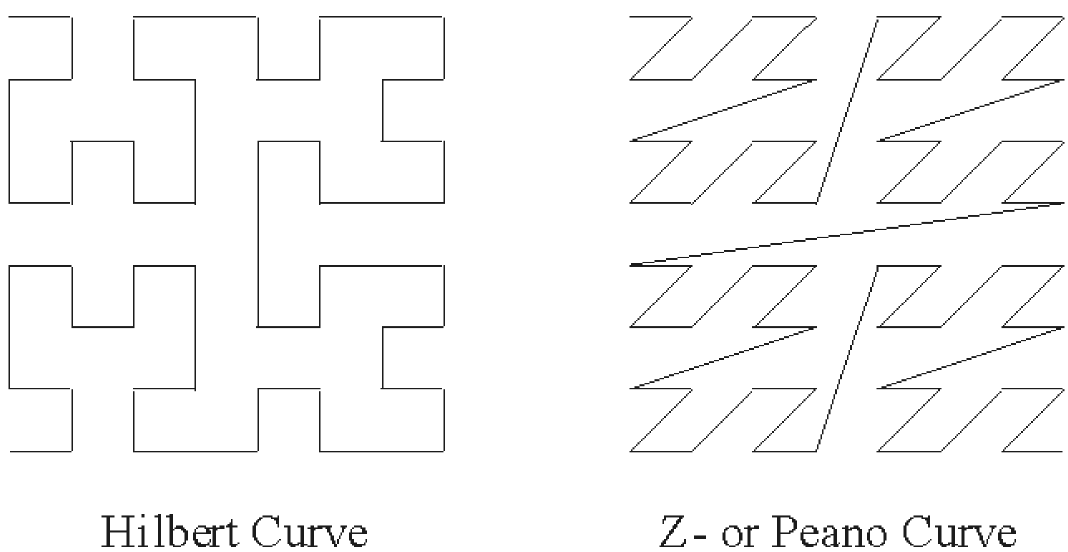

Figure 3.1. Space-filling curves.

While Hilbert ordering is slightly better at keeping close regions close, Peano- or Z-ordering, which is also called recursive raster ordering, has significant algorithmic advantages. In Peano-ordering, the *n* coordinates of the *n*-dimensional space are transformed into one 1-dimensional coordinate by a simple process of interleaving bits.

Figure 3.2 demonstrates the process in two dimensions. The point at $x = 2$ and $y = 1$ will be at position $s = 6$ in the Peano-ordered sequence.

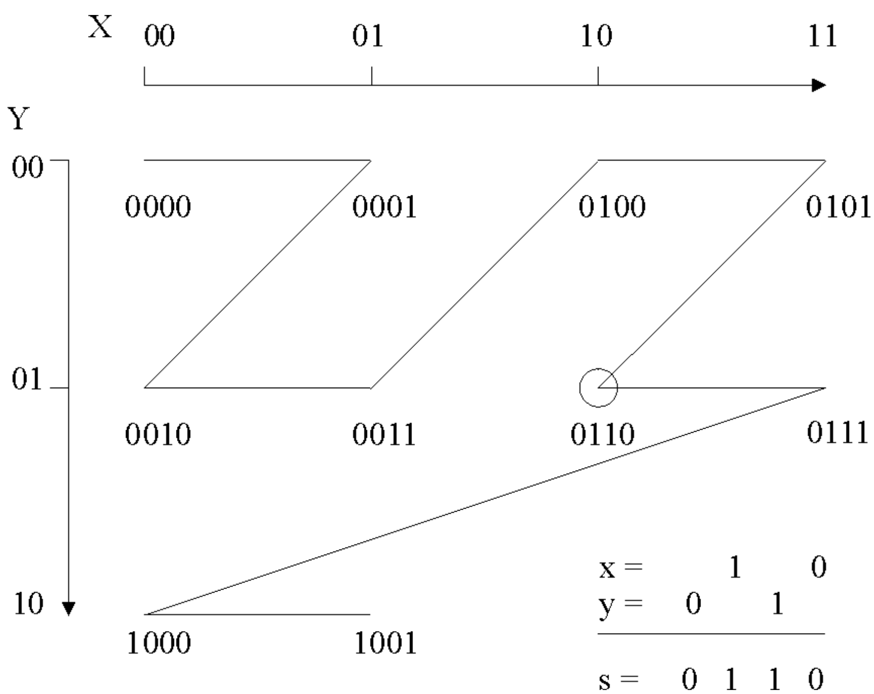

Figure 3.2. Construction of a Peano-ordered sequence through interleaving of bits.

In general, for d attributes, with b bits each, a particular structural position, **p**, will have index s in the sequence, where s is given by the following definition:

$$s = \sum_{i=0}^{b-1} \sum_{j=0}^{d-1} 2^{di+j} p_{b-1-i}^{(j)} , \tag{1}$$

where bit number 0 is the highest-order bit for all position attributes and $p_i^{(j)}$ is bit number i of the j^{th} structural attribute.

It is interesting to examine this transformation from a different viewpoint. The highest-order bits in the original coordinates give the coarsest grouping of data points. In Chapter 2, we referred to them as the highest level in a concept hierarchy.

Correspondingly, they are the most relevant ones in grouping data points according to their location. Therefore, we first use the highest-order bit in each dimension to determine the place in the Peano sequence. Once we have used all highest-order bits, we continue by progressing down the concept hierarchy.

For image data, the spatial coordinates themselves do not have to be stored because each pixel is represented. Starting coordinates, resolution, and the definition of the pixel order, therefore, uniquely define the position for each pixel in the image. Spatial coordinates are neither stored in our P-tree representation nor in common image formats. Other data do not necessarily have such structural dimensions. Many data sets that are used in machine learning and data mining have key attributes that do not fully explore their domain, or use arbitrary identification numbers as keys that have no relationship with the remaining data. If the key attributes do not fully explore their domain, a representation in the domain space of the key attributes can still be used but may incur a high storage cost. Attribute combinations that are not represented in the data set would now have to be included, and an additional mask would have to be constructed to identify meaningful points. When using P-trees, we do not commonly take this route. Instead, we represent all attributes as P-trees and construct any necessary indexes on the fly by an AND operation.

3.1.2. Generalized Peano-order Sorting

Whereas spatial data are given in a form that is sorted according to their spatial coordinates, other data commonly are not sorted at all or sorted according to an irrelevant identification number. It may then be advisable to choose an ordering that benefits compression; i.e., an ordering that has long sequences of 0 values and 1 values. If we sort according to a particular bit, b_0, this bit will have no more than one contiguous sequence of 0's and one of 1's, which will lead to very good compression for this bit. If we sort according to the combination of two bits, b_0 b_1, i.e., consider them a two-bit integer with higher order bit b_0, the compression of b_0 will be as before, and b_1 will consist of up to four contiguous sequences of 0 and 1 values. It is straightforward to use all bits of all attributes for sorting. We try to optimize compression by pursuing a second goal. If two bits of two attributes, b_i and b_j, are highly correlated, sorting according to b_i will also benefit the compression of b_j; i.e., we would like to choose those bits that are correlated most strongly with others as highest-order bits for the purpose of sorting. One solution to this goal is closely related to the Peano ordering concept. For Peano order, the highest-order bits of all attributes determine the ordering before lower level bits are considered. In generalized Peano-order sorting, we do the same when sorting according to feature attributes. The numbers that determine the sequence are constructed from the bits of all attributes in the following way: We start with all highest-order bits of numerical attributes as well as bits that correspond to Boolean attributes. The order between these highest-order bits is chosen randomly or from domain knowledge. For binary classification tasks, it is usually beneficial to use the class label as the highest-order bit because the class label attribute is involved in many AND operations. Next in sequence are the second highest bits of numerical attributes. They are grouped together with categorical attributes that require two bits for their representation, i.e., have a domain of 3 or 4 values. We encode categorical data by randomly assigning labels. Appropriate choice of distance measures ensures that differing labels are always considered to have distance 1 irrespective of the integer value they could be seen to represent. Equal values are considered to have distance 0. Chapter 2 discussed the procedure from a distance metric perspective. The example in Table 3.1 shows the bit order for two integer-valued and two categorical attributes. For integer-value attributes, bit 0 is the highest-order bit, and for categorical attributes, it may be arbitrarily chosen. Another example that highlights the Peano-order aspects of this sorting strategy can be found in Section 5.2.1. Note that the bits of categorical attributes can all be grouped together because they are at the same level in the concept hierarchy. Different bits of categorical attributes are treated as equivalent everywhere in the data mining code. In general, the n^{th} step groups the n^{th} bit of numerical attributes together with all n bits of a categorical attribute with a domain of $[2^{n-1}, 2^n)$ values. This strategy is chosen because categorical attributes that are represented by n bits can cover as many values as numerical attributes of which only the n highest-order bits are considered. If an attribute with many values is used for sorting, sequences naturally become fragmented. Therefore, attributes with small domain should be used first, together with the higher order bits of numerical attributes. In this interpretation, the n highest-order bits of a numerical attribute can be seen as defining a numerical attribute with a correspondingly limited domain.

Table 3.2 shows an example of a data file for the attribute bits in Table 3.1. Note how c_0 show high compression despite the fact that it is not used for sorting. The reason is that the data show a correlation between the highest-order bit in age and gray hair color. The data also show a correlation between the highest-order bit in height and sex, which allows compression for bit s_0.

Table 3.1. Bit order in generalized Peano-order sorting.

<table>
<tr><td>Attribute name</td><td>Attribute type</td><td>Represented bit positions</td><td>Domain description</td></tr>
<tr><td>Age</td><td>7-bit integer</td><td>a_0 ... a_6</td><td></td></tr>
<tr><td>Height in feet</td><td>3-bit integer</td><td>h_0 ... h_2</td><td></td></tr>
<tr><td>Sex</td><td>1-bit categorical</td><td>s_0</td><td><table><tr><td>Domain</td><td>label</td></tr><tr><td>male</td><td>0</td></tr><tr><td>female</td><td>1</td></tr></table></td></tr>
<tr><td>Hair color</td><td>3-bit categorical</td><td>c_0 ... c_2</td><td><table><tr><td>Domain</td><td>label</td></tr><tr><td>red</td><td>000</td></tr><tr><td>blond</td><td>001</td></tr><tr><td>brown</td><td>010</td></tr><tr><td>black</td><td>011</td></tr><tr><td>gray</td><td>100</td></tr></table></td></tr>
<tr><td colspan="4">Bit order used for sorting:<table><tr><td>bit position</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr><tr><td>attribute bits</td><td>a_0 h_0 s_0</td><td>a_1 h_1</td><td>a_2 h_2 c_0 c_1 c_2</td><td>a_3</td><td>a_4</td><td>a_5</td><td>a_6</td></tr></table></td></tr>
</table>

The decision about whether it is efficient to include the extra sorting step depends on the expected use of the data. Figure 3.3 shows the number of nodes in a P-tree that are required without sorting, using simples sorting, and with generalized Peano sorting. The number of P-tree nodes is proportional to the storage requirements.

Table 3.2. Example of a data set that was sorted using the bit-order in Table 3.1.

a_0	h_0	s_0	a_1	h_1	a_2	h_2	c_0	c_1	c_2	a_3	a_4	a_5	a_6
0	0	1	0	1	1	1	0	1	1	1	1	1	0
0	0	1	1	0	0	1	0	0	1	0	1	0	0
0	1	0	0	1	1	0	0	1	0	1	1	1	0
0	1	0	1	1	1	1	0	0	0	1	0	1	1
1	0	0	0	1	0	0	1	0	0	1	0	0	0
1	1	0	1	0	0	0	1	0	0	0	1	1	1

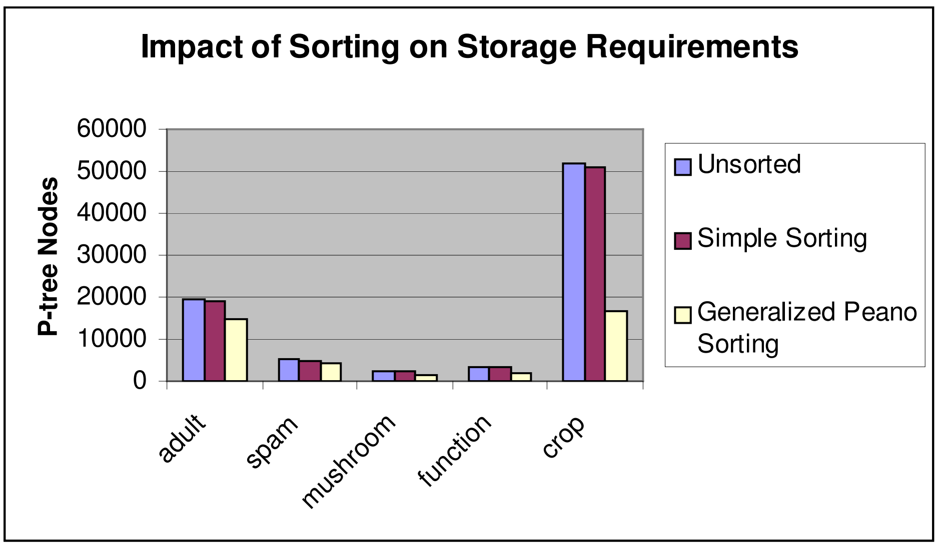

Figure 3.3. Number of P-tree nodes for different sorting schemes (data sets as explained in Chapter 4, with the crop data set restricted to 3 10^5 data points).

Good compression is not only beneficial to storage requirements. The speed of algorithms strongly depends on the number of P-tree nodes that are involved in the calculations. Efficient P-tree implementations do not require examining branches of any P-tree involved in an AND operation if at least one tree is known to be composed entirely of

0 bits, as will be explained in the next section. Figure 3.4 shows that the execution speed is significantly more affected by sorting, in particular generalized Peano sorting, than the storage requirements depicted in Figure 3.3 would have suggested. Times are based on the classification of 100 data points using rule-based P-tree classification as explained in Chapter 4.

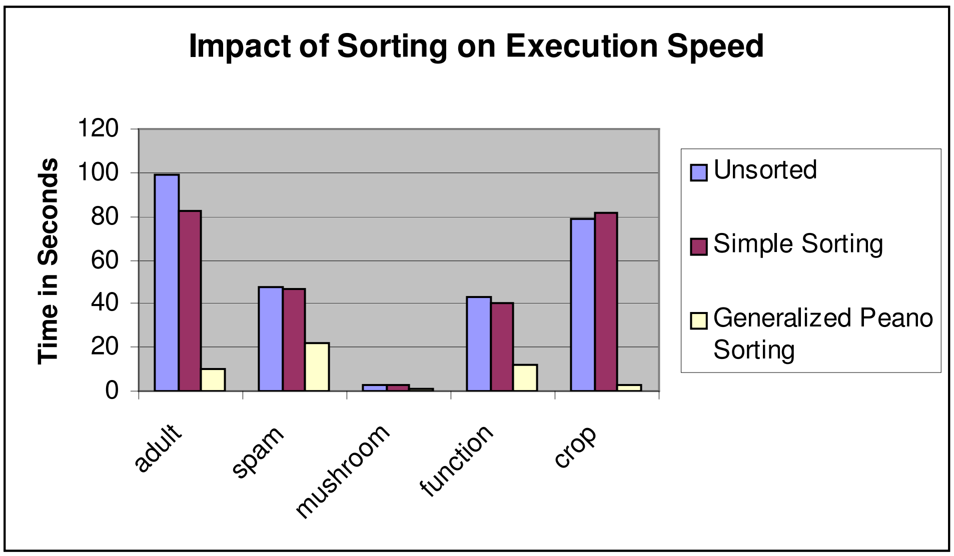

Figure 3.4. Time for the classification of 100 data points using rule-based classification as explained in Chapter 4.

3.1.3. Compression

Once an ordering has been established, the table is broken up into attributes, and attributes into bit-sequences, that are referred to as P-sequences. If optimal compression was desired, we could use run-length compression for the individual bit-sequences. The problem with such a scheme is that we routinely have to perform Boolean operations on the P-trees in response to queries and as part of data mining operations. We, therefore, choose a format that allows efficient execution of Boolean operations among P-trees while the data are compressed. To achieve this goal, it is beneficial if compression boundaries match among different bit-sequences. A tree structure is chosen that allows the hierarchical definition of boundaries. We will first describe the logical structure of a P-tree and then proceed to look at implementation choices that make P-tree operations more efficient.

Logically, a P-tree can be seen as a tree in which each level-0 node (lowest level) represents one bit of the data. Each level-1 node in the tree has f level-0 nodes as children, where f is the fan-out of the tree. The fan-out is chosen to be a power of 2, or a power of 2^d for d-dimensional data. For the purpose of this thesis, the fan-out is chosen to be constant for the entire tree. In principle, a different fan-out could be chosen for different levels. If all f children of the node are 0, the node is called "pure 0"; if all are 1, the node is called "pure 1"; in all other cases, the node is called "mixed." This statement is generalized at higher levels in the tree. If all f children of the node are "pure 0," the node is called "pure 0"; if all are "pure 1," the node is called "pure 1"; in all other cases, the node is called "mixed." A level-0 node that represents the bit 0 (1) can, therefore, be seen as a special case of a "pure 0" ("pure 1") node. Note that level-0 nodes cannot be "mixed." Children of "pure 0" ("pure 1") nodes do not have to be stored since they are guaranteed to be "pure 0" ("pure 1") at any level. P-trees achieve compression through nodes that do not have to be stored. Figure 3.5 shows the structure of a P-tree.

A node can be have three states, "mixed," "pure 0," and "pure 1," that can be represented using two Boolean variables. If the tree structure can be inferred from separate information, such as node addresses or the existence of child pointers, one bit is sufficient. The existence of child node information then automatically identifies a node as "mixed."

For computational reasons, it may nevertheless be useful to represent two bits at each node. In the following section, we will look at different implementation options.

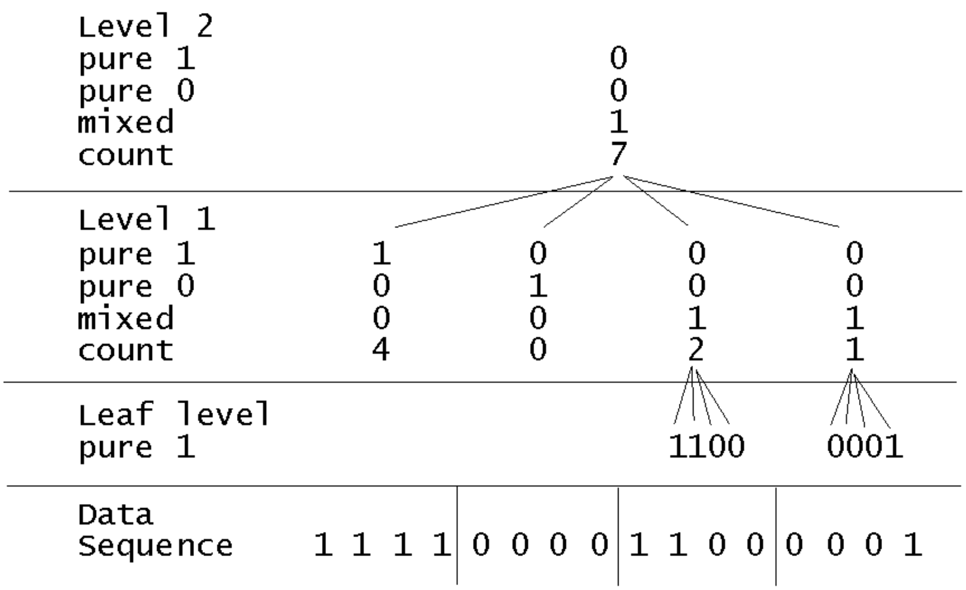

Figure 3.5. Structure of a P-tree.

3.2. Implementation

The logical definition of a P-tree does not uniquely specify its representation within a computer program. The tree structure itself can be maintained through pointers, node addresses, or as a sequence. Some implementations will, furthermore, represent child or even grandchild information within each node to improve efficiency. Four main types of implementations have been used in the past: quadrant-ID-based; tree-based; sequence-based; and array-converted, tree-based implementations. We will limit our discussion to those representations that can be seen as precursors to the array-converted, tree-implementation that has been implemented and used for this thesis. Before discussing differences among implementations, we will review some commonalities.

3.2.1. AND Operation

We frequently have to perform AND operations on P-trees. Appendix A gives a formal definition of an AND operation. The basic strategy of most implementations of the AND consists of determining those nodes that are guaranteed to be "pure 1" (nodes that are "pure 1" for all trees that are being ANDed) and nodes that are guaranteed to be "pure 0" (nodes that are "pure 0" for at least one tree that is being ANDed). The remaining nodes can be either "pure 0" or "mixed," and sub-trees have to be examined. Two main criteria for fast ANDing can be extracted from this description. Taking the AND of "pure 1" information and the "OR" of "pure 0" information must be fast. To this end, we make use of the parallelism of bit vector representations. We also have to be able to find children quickly. This goal can be achieved through pointers (See Section 3.2.3.) or storage of array indices (See Section 3.2.5.). Storing the pre-order sequence of a tree allows fast retrieval of the first child. Retrieving later children requires parsing all previous ones, which decreases performance when one or more children can be eliminated entirely because they are being ANDed with a pure 0 node.

3.2.2. Bit Vector Operations

Many implementations, to some extent, use the concept of bit vectors. To do so, the purity information of the children of a given node is collected into one or more bit vectors, called a child-purity vector. Internally, bit vectors are represented through integer types or arrays thereof. The size of each child-purity vector is given by the fan-out of the tree, i.e. the maximum number of children per node. A key factor in optimizing P-tree operations lies in the efficient use of the inherent parallelism that comes with the use of bit vectors. Bit-wise Boolean operations on integers can be done in one machine cycle and correspond to the parallel execution of 32 or 64 Boolean operations depending on system architecture. Bit vectors also allow an efficient implementation of bit counting. Most data mining algorithms rely on determining the number of data points that satisfy a particular condition which, in the context of P-trees, is the number of 1 bits that result from Boolean operations on P-trees. One possible counting algorithm would evaluate the number of occurrences of 1 in an integer by shifting the number one bit at a time. A faster implementation takes several bits and determines the number of 1 bits through table look-up. Bit sequence 0110, for example, has two bits set to 1. In a look-up table, we would store the value 2 as the number of bits for index 6, the number that 0110 represents. The same strategy can be used to determine the position of the first 1 bit in a bit vector. This strategy for counting and finding the first 1 bit works well for 8 bits with 256 entries in a look-up table. It is not efficient beyond 8 bits because the look-up table would become too large.

A general consideration is how to choose the size of bit vectors. In most representations, that size is equal to the maximum number of children, or the fan-out of the tree. For a structural dimension of 2, it is natural to choose a fan-out of 4. Each node in the tree then has four children, each of which represents a quarter, or quadrant, of the parent node range. Choosing a larger fan-out increases parallelism and can significantly improve the ANDing speed of P-trees. The current implementation was optimized for 32-bit registers. Thirty-two-bit vectors naturally represent a P-tree in 5 structural dimensions and cannot well be justified for 2-dimensional spatial data. We, therefore, used a fan-out of 16 that corresponds to collapsing two levels into one for spatial data.

3.2.3. Tree-based Implementations

In tree-based implementations, the tree structure is maintained through pointers. These pointers can either be provided by the programming language or can be logical pointers, such as array indices. Using language-provided pointers leads to problems, referred to as pointer swizzling, if sub-trees are to be distributed over a network. A further disadvantage of standard pointers is that storage requirements for each pointer cannot be adapted to the actual address space that the P-tree requires. Using logical pointers such array indices allows matching the data type to the address space requirements based on the actual P-tree-size and thereby reducing storage. Using array indices has the further benefit that arrays are commonly stored contiguously in memory. Iterating through an array is, therefore, likely to be faster than following pointers to unrelated positions in memory.

A common criticism of tree-based implementations is that the storage requirements of pointers could easily exceed the storage of nodes. It is correct that a naive implementation could show this behavior. Figure 3.6 gives a graphical view of different tree-based representations, each one giving the "pure 1" information. Note that, theoretically, no "mixed" bit has to be stored because the existence of a child-pointer is equivalent to "mixed" information. In practice, most tree-based implementations will still maintain the full child purity information to allow efficient bit-vector-based computations as well as allowing compressed storage of child-pointers.

It can be seen that the number of pointers is equal to one less than the total number of nodes. Intuitively, it may seem as if the number of pointers had to scale as the number of nodes multiplied by the fan-out. The scaling is, however, better since no pointers have to be maintained at the lowest level. A naive implementation in which nodes represent their own purity information is nevertheless very inefficient. If the address space is assumed to be 1 million nodes, corresponding to 20 bits, pointers require 20 times the storage required by nodes. A representation that maintains the child purity at each node will improve the ratio, especially for a large fan-out. If we assume the fan-out to be 16, the storage space for pointers will be comparable to that of the data. We can repeat the process of representing child information within the parent, leading to a representation of grandchild purity within each node. A child purity representation has storage requirements for pointers of approximately $1/f_e$ that of the node representation, where f_e is the effective fan-out, i.e., the number of children that have to be stored. We will discuss this improvement for the sequence-tree-hybrid implementation. Appendix A systematically carries through the corresponding transformations.

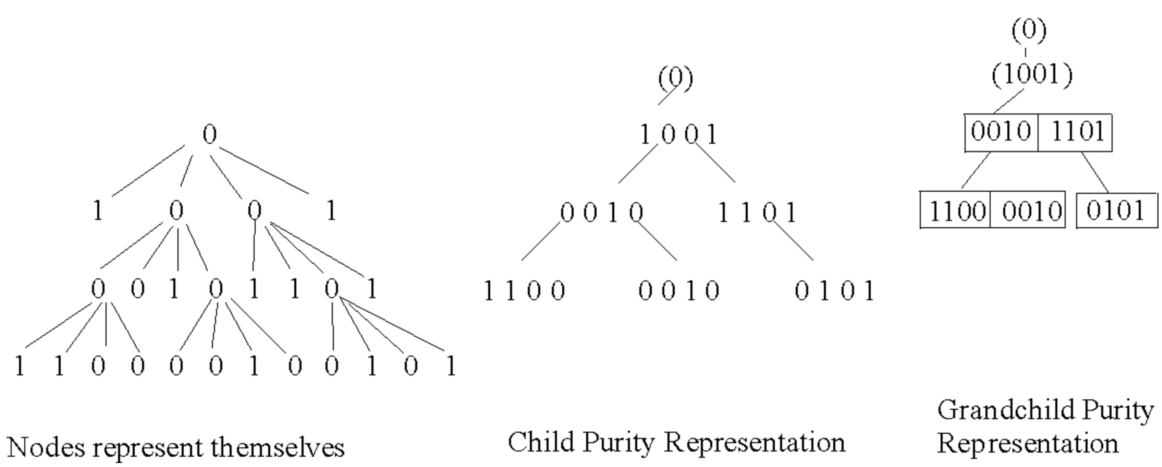

Figure 3.6. Representations of P-tree structure (pure 1 information displayed).

3.2.4. Sequence-based Implementations

Sequence-based implementations rely on the storage sequence for reconstruction of the original data. Sequence-based representations can be constructed for any of the three tree variants discussed previously: nodes that contain their own purity, child purity, or grandchild purity. The storage sequence alone can only be loss-less if purity information is allowed to cover the three values of "pure 1," "pure 0," and "mixed." Note that, for tree-based implementations, "pure 1" (or "pure 0") information alone is sufficient to distinguish "pure 0" from "pure 1" nodes, with mixed nodes being identified by the existence of a child. The values in three-value logic are mapped to the computer-supported binary logic by representing two of the three possible states, such as "pure 1" and "mixed." The third value ("pure 0") can be inferred from the other two as "pure 0" = $\neg$ ("pure 1" $\vee$ "mixed"). An alternative way of describing this implementation is to say that the "mixed" information represents the tree-structure in a way that is equivalent to, albeit different from, pointers or node addresses, called quadrant IDs, in other representations.

The storage sequence can be defined according to any of the common tree-walk strategies, such as depth-first or breadth-first, where a depth-first tree-walk allows further choices regarding the positioning of node values with respect to each child tree-walk (pre-order or post-order sequence). In a pre-order sequence, the value of a node is stored before sequences that are defined by the child nodes. In the next section, ideas from pointer- and sequence-based representations will be combined for maximum storage and ANDing speed efficiency.

3.2.5. Array-converted Tree-based Representation

The benefits of tree-based representations, namely the fast access to all child nodes, can be achieved without the main drawbacks of pointers. If node information is stored in array form, array indices can easily serve the purpose of pointers. The conversion is especially easy and efficient for the grandchild purity representation discussed in Section 3.2.3. Grandchild purity information can be grouped by child node. For each mixed child, an address has to be stored as well as the child's child-purity vectors; i.e., exactly one array index has to be stored for each bit vector pair of child-purity information, allowing for a straightforward array-based storage organization. Figure 3.7 shows an example of an array-converted, tree-based representation as it was used in the code that was written for this thesis.

"Pure 1" information together with "mixed" information are used to perform the bit vector operations necessary in P-tree ANDing. The address sequence, *a*, maintains the array indices that act as pointers to child-nodes. Each node contains the full grandchild purity information, i.e., a bit vector (child-purity vector) for every mixed child. The example in Figure 3.7 depicts the "pure 1" information above the "mixed" information for each node. The lowest-level node requires neither mixed information (Level-0 nodes are pure by definition.) nor addresses (Level-0 nodes have no children.) The count of bits is furthermore maintained to increase ANDing speed. The key benefit of this representation with respect to simple sequence-based representations lies in the fact that the branch on the right side of Figure 3.7 can be located without iterating through the branch on the left side. Without this property, ANDing speed would not gain serious benefit from compression, and the improvements in execution speed that were depicted in Figure 4 would not be possible.

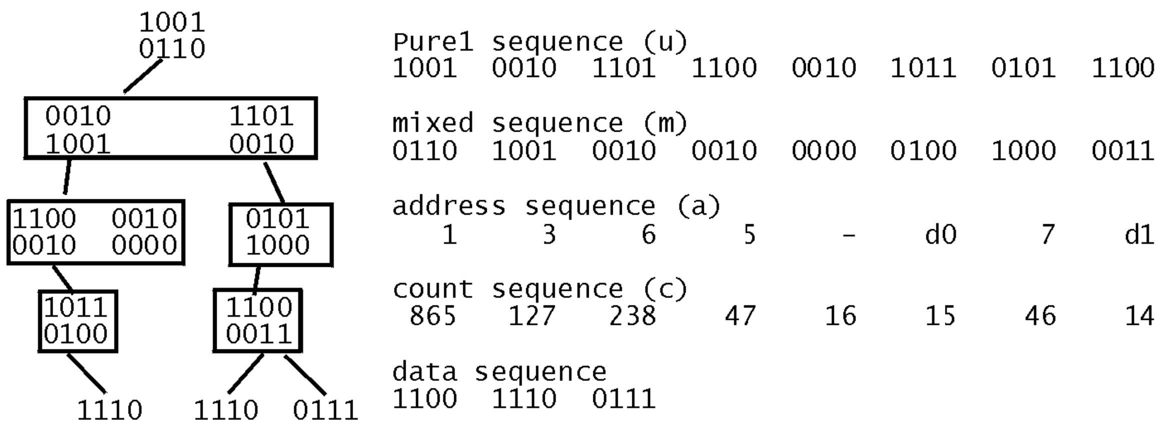

Figure 3.7. Large example that represents the implementation for this thesis.

3.3. Application Programming Interface (API)

Many people use and contribute to P-tree-based data mining code. It is, therefore, important to make collaboration as easy as possible. Providing a well-defined application programming interface, API, is central to enabling collaborative programming. The design of the API was guided by the wish to allow a flexible combination of different P-tree implementations with a variety of data mining algorithms on a wide choice of data sets. We, therefore, structured the API into a data mining interface, DMI, that defines how P-tree code is called from data mining applications and a data capturing interface, DCI, that specifies the format in which data are read into a P-tree. Figure 3.8 shows the relationships between the most important classes of the API, using universal modeling language, UML, notation. The classes will now be explained. Please refer to [4] for a complete UML class diagram.

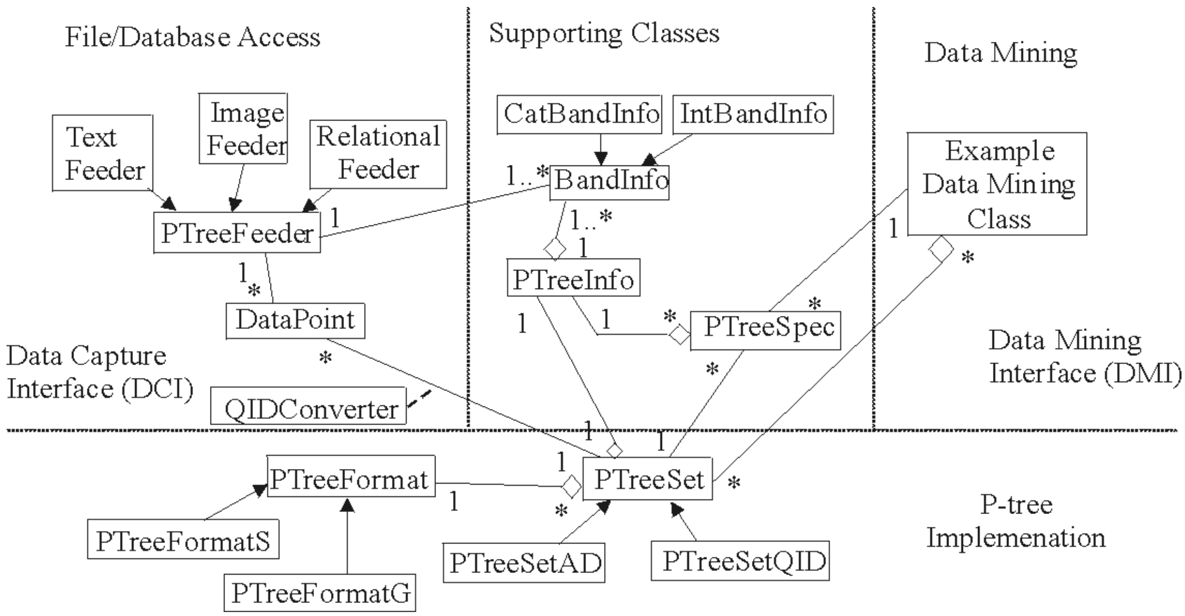

Figure 3.8. Relationships among the most important classes in the P-tree API.

At the time of designing the interface, several P-tree implementations were already in existence. We, therefore, had to be sure that each one of them would fit into our model. One way of ensuring compliance with existing code was to use two significantly different implementations, one of which is presented in this thesis, as benchmarks for the feasibility of any suggestion. A result of this strategy was that we decided to combine P-tree creation and ANDing into one class, PTreeSet, that holds those basic P-trees that are to be used in AND operations. PTreeSet may hold complement P-trees as well as basic P-trees if the implementation requires it. Alternatively, the implementation may opt to construct complements on the fly. For this and other reasons, it would be limiting to define a class PTree and insist on how P-trees are to be combined into PTreeSets. Two types of parameters are used to define the logical structure of a P-tree: the fan-out and the number of levels. We combined these parameters into a class, PTreeFormat. Some implementations may allow different fan-out at different levels, whereas others will use one fan-out for the entire tree. These distinctions were handled by creating sub-classes to PTreeFormat.

3.3.1. Data Mining Interface (DMI)

The main operation of the DMI consists of requesting a count as a result of an AND operation on a particular combination of P-trees (andCnt(PTreeSpec)). The central construct that allows defining the combination of the P-trees that are to be ANDed is the P-tree specification, PTreeSpec. The P-tree specification consists of a bit-pattern, "pattern," that is 1 if a basic P-tree is to be included in the AND and 0 for a complement P-tree. A second bit vector, "mask," specifies those P-trees that are to be included in the AND. In principle, it is possible to set the bits in both of those bit vectors individually. In practice, especially for a large number of P-trees, it is not advisable to do so.

Much of the work on the DMI was guided by the need that arises from the complexity of dealing with several hundred P-trees that belong to dozens of different attributes, representing many different data types. A main decision that was taken was to allow access to P-trees based on attributes, or bands, as well as relative indexes within those attributes. Bands can be identified by their name. In practice, access by a sequential number was determined to be at least as important. P-trees that belong to one band can be distinguished by an index within the band. At a still higher level, one may wish to use methods that increase or decrease intervals in a type-independent fashion rather than explicitly dealing with indexes within a band. Such methods were included into sub-classes of PTreeSpec that were used for the programs described in this thesis. The high-level methods were intentionally not included into the DMI with the intent of maintaining simplicity for programmers who may not need such generality. The possible need to identify band types did, however, motivate a set of classes that preserves meta-data information from the data file. In an initial design of the API, we underestimated the need of making meta-data information available to data mining code. The programs written for this thesis demonstrated the need to improve the design and formally allow the transfer of meta-data information from a data file to data mining code through a class, BandInfo.

The BandInfo class maintains information regarding the type of band, such as whether it can have unknown values as well as type-related information. A band with unknown values requires an additional P-tree that identifies those data points for which the particular band information has been provided. BandInfo also maintains the position of the particular band within the PTreeSet. Each BandInfo object may, therefore, only be part of one PTreeInfo object that goes with one PTreeSet. Different types of bands, such as integer, bit vector, and categorical bands, differ in the way they represent distances and intervals. Categorical attributes only allow two distances, distance 0 if values are identical and distance 1 if they differ, with no other distances defined. A single-valued categorical attribute may be represented by a label, such as red = 0, green = 1, blue = 2, provided distances are guaranteed to be evaluated correctly. Label-encoded categorical attributes are represented by class CatBandInfo. Multi-valued categorical attributes are commonly represented by bit vectors where each domain value is represented by one bit. Distance 1 now corresponds to one matching bit with multiple bits combined through OR. Requiring all bits to match (AND), as in the case of label-encoded integers, would correspond to requiring each of the multiple values to match, which clearly does not represent the common understanding of matching values. Multi-valued categorical attributes are not yet integrated into the API.

The BandInfo sub-classes, such as IntBandInfo and CatBandInfo, offer specialized implementations of methods such as getDataMeaning(bit_vector) and getRepresentation(string) which allow translating back and forth between the conventional representation of the data and the bit vector representation used within the P-tree code. BandInfo objects are collected into a central PTreeInfo class that maintains all information related to a particular PTreeSet. Each PTreeSet holds a PTreeInfo object that is updated whenever a band is added to the PTreeSet.

3.3.2. Data Capture Interface (DCI)

The data capture interface was designed to make file reading independent of the P-tree implementation. Independence is achieved by supplying a PTreeFeeder class for each file-format that is to be read. The PTreeFeeder class offers a method getPoint that returns the data for one data point as an object of type DataPoint. Each object of type DataPoint consists of a key (retrievable by method getLocation()) as well as a bit vector that contains the bit values for all basic P-trees (retrievable by getData()). It is important to note that information is passed one data point at a time; i.e., no separate data structure has to be held in memory to supply the data that are used to construct P-trees. Most PTreeFeeder classes are implemented to read data from a stream, such as a file, when the getPoint() method is called. Note that PTreeFeeders do not have to be implemented this way. Data can also be the result of a database query or may be read into an array first and read from the array for each call of getPoint(). The latter options are important if data are to be sorted according to one or many of the feature attributes.

The DataPoint and PTreeFeeder classes need to know nothing about P-tree format other than that it is a bit-wise representation. Since a DataPoint provides only one bit for any one P-tree, it is unaffected by the actual P-tree storage or compression, or by Peano ordering. Peano ordering can be seen as separate from both the file reading and the P-tree implementation. The conversion from location information into quadrant identifier information (qid) was, therefore, moved into a separate class QIDConverter. An important goal of both the DCI and the DMI was to keep those classes that have multiple implementations as small as possible, e.g., the PTreeFeeder that requires a separate implementation for every file format. The PTreeSet class also has many implementations that are beneficial for different types of data. Any responsibility that can be transferred to supporting classes reduces the effort of implementing any of the classes of which multiple variants are necessary or desired.

The PTreeFeeder class does have to construct the BandInfo objects that hold meta-data and offer methods for use by data mining algorithms. Meta-data can come from the data file itself or may be even be determined by the fact that a particular file format is used. Tiff color images, for example, will always contain integer-valued information, and bands will be required; i.e., there will be no pixels that have information on red and green intensities but no value for blue. For more general data formats such as data from the University of California at Irvine, UCI, machine learning repository, meta-data have to be read from a separate file.

Additional supporting classes can be and have been implemented, e.g., to clean data that come from particular data files or to assist in common data mining tasks such as the calculation of averages, use of HOBbit-based Gaussian weight functions, etc. Most of these supporting classes are not considered part of the API but may be included if many people use them.

3.4. P-tree API as an Example of a Column-based Design

We will now look at the P-tree API in the light of column-based data organization as was discussed in Section 2.6. The entity that represents the main data mining table, PTreeSet, is considered as one class, as is the case normally when records are treated as an object. Bit-columns are represented in P-tree format using a special class, PTree, to handle the compression and hierarchical organization. Class PTree is not part of the API since its interface is implementation dependent. The implementation that was done for this thesis does, however, have a distinct class PTree, as do most other implementations.

A generic implementation of operations among P-trees is not easy due to their hierarchical structure and was not attempted in the context of this thesis, although plans for such an implementation are currently being developed. The main operation on P-trees is the AND operation that determines (the number of) those rows that match a sample in a specified sub-set of its attribute bits. This operation requires the ability of specifying a row, which is done using class PTreeSpec. The interesting aspect of this class is that it represents a complete row that has to match all attribute definitions of the data mining table, PTreeSet. Operations on the row specification class, PTreeSpec, rely on some knowledge of the attributes in the data mining table, PTreeSet. We, therefore, need a class to maintain header information, PTreeInfo. Since the header information has to match the attributes in PTreeSet, a PTreeInfo object is contained within the class that represents the PTreeSet object. A new copy has to be retrieved whenever a row specification (object of class PTreeSpec) is constructed. Header information is, furthermore, broken up into attribute headers, BandInfo. Attribute header objects represent type information as well as maintaining methods that can be used in the manipulation of row specification, PTreeSpec, objects. PTreeSpec does not maintain methods that are to be used by the data mining table, PTreeSet, itself; otherwise, the performance issues of using method calls on a large number of rows would recur.

Our design, therefore, requires a minimum of five classes, PTreeSet, PTree, PTreeSpec, PTreeInfo, and BandInfo, to represent a single column-based table, with additional classes used to handle compression specific issues such as PTreeFormat and QIDConverter that were discussed in the previous sections. A row-based implementation would require no more than two classes, one that represents a row and a container class to allow access to all rows. This difference shows that an object-oriented implementation of a column-based data structure does indeed use more classes than a row-based implementation. It should not, however, discourage the use of an object-oriented implementation since it was performance that guided our design. The benefit of using an object-oriented design can be seen from previous sections that demonstrated how the table implementation becomes an integral part of a complete object-oriented design with all its benefits.

3.5. References

[1] Q. Ding, W. Perrizo, and Q. Ding, “On Mining Satellite and Other Remotely Sensed Images,” Workshop on Data Mining and Knowledge Discovery (DMKD-2001), Santa Barbara, CA, 2001.

[2] W. Perrizo, W. Jockheck, A. Perera, D. Ren, W. Wu, and Y. Zhang, "Multimedia Data Mining Using P-trees," Multimedia Data Mining Workshop of the Conference on Knowledge Discovery and Data Mining (KDD-2002), Edmonton, Canada, June 2002.

[3] A. Perera, A. Denton, P. Kotala, W. Jockheck, W. Valdivia Granda, and W. Perrizo, "P-tree Classification of Yeast Gene Deletion Data", SIGKDD Explorations, Vol. 4, No 2, pp. 108-109, Dec. 2002.