INDEXING TECHNICAL DATA IN A MATERIALS DATABASE

F J Smith and L C Emerson

Department of Computer Science
The Queen's University of Belfast
BELFAST BT7 1NN
Northern Ireland

ABSTRACT

Much of the data used by a design engineer lack precision, particularly materials property data which are measured in inexact physical experiments and engineering tests. Indexing such data needs to suit the various types of proximity searches of interest to an engineer. It is shown that the B*-tree performance is best whenever the data are stored as specific values. A new structure, the Interval Tree, is presented which proves to be useful when much of the stored data are expressed in the form of ranges.

1 INTRODUCTION

Much of the technical data needed by the design engineer are lacking in precision; this is partly because the materials and components in a CAD/CAM database normally do not exactly match the specification of the object being designed and partly because the properties of the materials or components in the database are not precise. They are not precise because they are determined by experimental or test measurements. Conventional database systems do not provide the necessary tools to the engineer to enable him to deal with such technical data. In this paper we discuss the design of one of these tools, a tool to index numerical data on the properties of materials or components.

The problem of the inexact nature of much technical data was mentioned briefly in a previous conference paper [1]. However, the problem of indexing such data was not discussed in that paper and references to the subject in the literature are sparse. Some relevant papers are found in an extensive review of databases in Science and Engineering by Rumble et al in 1986 [2] which points out that the availability of software and database systems to handle technical data on the properties of materials or components is "generally poor". The database software available to Engineers has not improved a great deal since the data of that report, 1986, apart from the recent rapid development of software on Object Oriented Databases and Knowledge Bases [3]. However, whether Relational technology or Object Oriented technology is used in an engineering database (or knowledge base) environment, the indexing of the inexact numerical values found in the system is necessary if the system is to be efficient and give rapid responses to the design engineer.

2 INEXACT NATURE OF PROPERTIES DATA

Examples of some typical engineering data are illustrated in Tables 1 and 2, which are simplified versions of tables of values taken from typical engineering handbooks; these illustrate the type of data which would need to be stored in a good CAD/CAM database system.

The imprecise nature of the properties is clear from both tables, but particularly from Table 2 as much data in Table 2 are represented by ranges. For example, the tensile strength of an Epoxy Cast Resin lies between 26 MN m⁻² and 85 MN m⁻², i.e. varying almost by a factor of 4. So this data item is far from precise and deviates widely from the kind of data items found in conventional database systems. Note also that the data item "Epoxy Cast Resin" is itself imprecise, as there are many such resins, depending of the composition of the resin and the amount of hardener used. Information on this composition is not available in the table. The Engineer regularly faces this inexact specification of materials as well as the imprecise numerical values of material properties.

Table 2 includes an unusual number of data items in the form of ranges, e.g. 26-85 MN m⁻² or 1.25-1.67 J g⁻¹ K⁻¹. Normally, as in Table 1, only one value is given; but, even when only one value is recorded an engineer will still know that it is not precise. Take, for example, the heat capacity of acetals in Table 2, recorded as 1.46 J g⁻¹ K⁻¹. Any engineer reading this table will know that 1.46 units does not mean this exactly. He will know, for example, that there is no significant difference likely for his purposes between 1.461 and 1.46 J g⁻¹ K⁻¹. Any engineer reading this table will know that 1.46 units does not mean this exactly. He will know, for example, that there is no significant difference likely for his purposes between 1.461 and 1.46 J g⁻¹ K⁻¹. We therefore need to examine more closely what is meant by a numerical property such as 1.46 J g⁻¹ K⁻¹ or more generally the quantity 1.46 in any unit. An engineer has a view, born of experience, of its meaning: it represents a "fuzzy" quantity, one probably close to 1.46 units but with a decreasing probability of being some distance from 1.46. This fuzzy quantity has a probability distribution like that in Figure 1, rather than a single precise quantity.

The range of this distribution is itself fuzzy. Mathematically the value 1.46 should mean that the "exact" value of the property should lie between 1.455 and 1.465, but such mathematical exactitude is not the norm in the Physical Sciences or Engineering. Sometimes the range of possible values has been estimated by the originator of the datum 1.46 and given error bars, say 1.46 ± 0.02 which the engineer is expected to interpret as a half width in a distribution as in Figure 1. Sometimes the error bars are given a confidence measure of 90% or 95% (also illustrated in Figure 1); but although the Experimental Physicist or Test
### TABLE 1: EXAMPLE OF NUMERICAL DATA ON MATERIALS PROPERTIES

<table>
<thead>
<tr>
<th>alloy</th>
<th>thermal conductivity</th>
<th>resistivity</th>
<th>modulus of elasticity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nickel silver 18% alloy A</td>
<td>0.14</td>
<td>29</td>
<td>18</td>
</tr>
<tr>
<td>Cupronickel</td>
<td>0.13</td>
<td>35</td>
<td>22</td>
</tr>
<tr>
<td>Red brass (cast)</td>
<td>0.32</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>Chemical lead</td>
<td>0.15</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>Antimonial lead (hard lead)</td>
<td>0.13</td>
<td>23</td>
<td>3</td>
</tr>
<tr>
<td>Solder 50-50</td>
<td>0.20</td>
<td>15</td>
<td>-</td>
</tr>
<tr>
<td>Magnesium alloy AZ31B</td>
<td>0.34</td>
<td>9</td>
<td>6.5</td>
</tr>
<tr>
<td>K Monel</td>
<td>0.08</td>
<td>58</td>
<td>26</td>
</tr>
<tr>
<td>Nickel</td>
<td>0.26</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>Cupronickel 55-45 (Constantan)</td>
<td>0.10</td>
<td>49</td>
<td>24</td>
</tr>
<tr>
<td>Commercial titanium</td>
<td>0.08</td>
<td>80</td>
<td>16.5</td>
</tr>
</tbody>
</table>

### TABLE 2: PROPERTIES OF POLYMERS

<table>
<thead>
<tr>
<th>polymer</th>
<th>p kg m⁻³</th>
<th>tensile strength MN m⁻²</th>
<th>10⁶ coeff heat capacity Jg⁻¹ K⁻¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acetals</td>
<td>1420</td>
<td>65⁺</td>
<td>8</td>
</tr>
<tr>
<td>Cellulose</td>
<td>1480-1530</td>
<td>80-240</td>
<td>1.46</td>
</tr>
<tr>
<td>Cellulose acetate</td>
<td>1220-1340</td>
<td>12-58</td>
<td>8-18</td>
</tr>
<tr>
<td>Chlorinated polyether</td>
<td>1400</td>
<td>39</td>
<td>1.26-1.28</td>
</tr>
<tr>
<td>Epoxy cast resin</td>
<td>1110-1400</td>
<td>26-85</td>
<td>4.5-6.5</td>
</tr>
<tr>
<td>Phenolic cast resins</td>
<td>1240-1320</td>
<td>35-59</td>
<td>6-8</td>
</tr>
</tbody>
</table>

*⁺this strength decreases rapidly in the presence of acids

*based on unreliable measurements which overestimate the coefficient

Engineer who produced the datum, 1.46 ± 0.02, may have given a confidence of 90% on the error bar, the user Engineer will know from experience to treat this with some skepticism and widen the limits and confidence level considerably in his view of what this means. So a value 1.46 ± 0.02 with 90% confidence will be taken by an experienced engineer to mean that the "true" value probably lies in a wider range such as

1.46 ± 0.06

with a reasonable confidence level (say 80%). When no error bars are given, but only a single value 1.46 as in Table 2, this might be interpreted to mean a range of about

1.46 ± 0.10

This assumed range in the fuzzy view of the data item obviously depends on the nature of the data and on the source. A range specified by two values will be treated similarly. Thus the range mentioned earlier

1.25 - 1.67 Jg⁻¹ K⁻¹

will not be interpreted to mean that 1.25 and 1.67 are strict bounds to the range of possibilities, but rather a probable range, similar to that shown in Figure 2, i.e. with a fuzzy boundary.

In conclusion, we can note that numerical properties of materials or components are usually represented in a database by a single real number, or by a range of real numbers. Both of these represent a fuzzy range of values to an engineer, as in Figure 1 or Figure 2 and our database or knowledge base system needs to take this into account when indexing and using the data.

### 3 THE REQUIREMENT OF THE DESIGN ENGINEER

Before beginning to look at methods of indexing these inexact property data, we need first to look at the kind of questions which an Engineer is likely to ask about the properties of materials.

13
1.46 Property Value

A property with a value recorded in a database as 1.46 is viewed as a fuzzy quantity by an Engineer, with a probability distribution similar to that in the figure.

Figure 1:
A property with a value recorded in a database as 1.46 is viewed as a fuzzy quantity by an Engineer, with a probability distribution similar to that in the figure.

1.46 is viewed as a fuzzy quantity by an Engineer, with a probability distribution similar to that in the figure.

Direct search
The simplest query is to find the properties of a given material. If the required material can be specified clearly and if it matches a material in the database, (very often this is not true, but that would be the subject of another paper), then the property data (say 1.46 units) can be obtained and used for some purpose such as deciding on the diameter of an electrical cable to carry a certain current or on the thickness of a strut to carry a certain load. The engineer will treat the value 1.46 with a healthy skepticism and use the upper or lower limit of the range as appropriate to carry out a conservation calculation on the design.

Inverse search
The next simplest search is the inverse search, i.e. for materials with a specified property, e.g. having a numerical value 1.46 units. The search engineer will normally use a search quantity like 1.46 units as a "fuzzy" quantity in exactly the same way as we described in Figure 1. So, for example, he will certainly expect to retrieve a material with a property value of 1.46! So we see that normally a search for a single value is really a search for a narrow range, such as

1.46 ± 0.01

This is similar to the next type of search called a proximity search and differs only in degree of proximity, that is on the range of values to be searched.

Proximity search
In a proximity search the engineer is looking for materials with properties close to the specified value, i.e. close to 1.46 units, say within 10% of this value. It may be specified in this way, or alternatively it may be specified as the n-nearest materials, for example, the 10 materials with properties closest to 1.46 units. Even if no materials are found close to 1.46 the engineer will still want the materials with the nearest properties, e.g. material A with value 1.73 and material B with value 1.09, even if these can hardly be called "close".

Materials with property values expressed as ranges which intersect with the above range would also be found. So a material which we call Material C with property 1.05 - 1.45 (or 1.25 ± 0.20) would be retrieved. If material C is a composite material, for example a metal alloy with two components, metal A and metal B, and the range 1.05 to 1.45 is due to varying composition, the system should be able to compute the proportion of A and B to most closely match the required property, 1.46 units.

Range search
Similar comments can be made for a range search, where the engineer wants materials with properties within a range, for example within the range

1.20 - 1.50 units.

If few or no materials are found within the range, then the engineer may wish to retrieve materials with values just outside the range, e.g. material B with property 1.09 might be acceptable and would be retrieved.

Bounded search
When one of the limits to a range search is infinity (or zero for a positive valued property) then we have a

1.46

Figure 3:
An engineer's view of a range expressed as a data item "> 1.46".
bounded search. This is probably the most common form of range search, for example, a search for a material with a melting point above 2,000°C or resistivity below 0.001 ohms m⁻¹. Since these may retrieve a large number of materials (sometimes all but a few are excluded by the condition) it is almost always used along with several other conditions or ranges for other properties, e.g. a query may be for a material with a melting point above 2000°C, a resistivity less than 0.001 ohms m⁻¹ and a density less than 2.0 \times 10⁻⁹ kg m⁻³.

A material meeting two of the above conditions but not quite meeting the third would still be presented to the engineer if the number of hits was not high. So a material with melting point 3500°C, resistivity 10⁸ ohms m⁻¹ and density 2.8 \times 10⁻³ kg m⁻³ might still provide a solution, if not an ideal one, to meet the engineer’s needs.

4 INDEXING

From the previous two sections it is clear that both the physical properties of materials and the expression of searches for materials with specified properties are in the form of ranges of values, either narrow ranges for (apparently) specific values, slightly wider ranges for proximity searching and still wider ranges for range or bounded searches. For these purposes, in summary, 1.46 and 1.461 are effectively the same data.

Indexing such data is going to be different from indexing the kind of discrete data found in conventional database systems. Two forms of indexing are most commonly used for these conventional data items:

1. B-trees
2. Hashing

We look at both of these and show how each has to be adopted to suit numerical properties data.

4.1 B-Tree

The indexing of numerical properties data is straightforward if none of the data is expressed as a range (as in Table 1) or if the ranges are small enough that they do not overlap and can therefore be represented by a single real number. Then the real values of the properties can be stored in sequence and indexed in a B-tree in a traditional manner [4,5].

Because searches will almost always be for a range of values, either a short range or a wide range, it is important that the blocks holding the ordered sequence of quantities (or of the data records which store the quantities) should include pointers to the next block in the sequence, as illustrated in Figure 4; the blocks may also be linked in the reverse direction. Also, it is important that all the sequence of quantities (or the data records which store the quantities) should appear on the same level of the tree. With these additions a B-tree is normally called a B⁺-tree, as shown in Figure 4 [6]. This is more efficient as it saves the system from navigating up and down through a tree to bring together values at the end of one block with values at the beginning of the next, when a search range overlaps 2 or more blocks of the sequence set.

So a B-tree satisfactorily indexes a set of quantities, provided none is expressed as a range. An example of the kind of difficulty which occurs is illustrated in Figure 5. If all of the ranges being indexed were short enough, then one value, at the centre of the range, could be used to build the B-tree. However, when the ranges are varied in width, as in Figure 5 or as in Table 2, then the wider ranges are likely to span several blocks of data in the sequence set. A search for a value at the end of a range will not find it, as the centre of the range will be stored in another block, possibly 2 or more blocks away.

One way round this problem is to store 2 values for each range, the minimum and maximum values and then produce an index for all minimum values and a second index for all maximum values. Single values properties can be recorded twice, with the minimum value being equal to the maximum value.

This certainly permits the retrieval of every value and range in response to a search, but at a cost, the cost of a large amount of processing. For example, to find all values close to 1.46 units and ranges overlapping with 1.46 units it would be necessary to first find all records with minimum values less than 1.46 and then match these with a list of all records with maximum values greater than 1.46. This would involve the merging of two long lists and the total length of the two together would equal the total size of the file being searched, since every record would be in one or both lists.

This would not make sense, as it would be quicker to search through a random index file with a linear search, without using a B⁺-tree.

It follows that if the stored data includes wide ranges as data items, a B⁺-tree will not be suitable for indexing. Fortunately, this is not often the case, and the property values can be expressed as single values or short ranges only, like 1.46 ± 0.02. Then a B⁺-tree is effective and efficient.

4.2 Hashing

The second common method of indexing in databases is by generating a computed hash address of the appropriate block from a key value. We find immediately, however, that the most common methods of hashing used for discrete values (such as division hash coding) cannot be applied to the real numbers or ranges which specify properties data. Take for example the three property values

1.459, 1.46, 1.4607 units.

These data items in almost all cases will represent effectively the same value to the Engineer, but even when it is important to recognise the differences it is clear that they should be stored together and retrieved together. However, if we used a Division Hash index on these values, indexing them as separate "words" or codes, they are all likely to generate different addresses and therefore to be stored in different and separated blocks on a disk store.

In recent years many new ideas have evolved on hashing. New forms, such as Linear Hashing [7, 8] and Extendible Hashing [9] overcome the traditional problem of long lists of overflow blocks when the number of blocks N has been under-estimated. However, the data records are still spread randomly across the file and loose their natural order.

They can still be stored together in a hashed file if the hashing algorithm is chosen in an appropriate form. The division hash algorithm mentioned above normally uses the remainder as the address of the block on disk holding a record. This can be written as

\[ R = K \mod N \]
where \( K \) is the key and \( N \) the number of blocks in the hashed file. For properties data, to keep contiguous blocks together we can use the quotient rather than the remainder

\[
Q = K \div N
\]

and generate addresses directly from the quotients \( Q \). This is ideal if we want all intervals to have the same size. Often we do not. More generally we specify intervals of values such as

\[
<0.1, 0.1 \text{ to } 0.2, 0.2 \text{ to } 0.4 \text{ etc}
\]

and then store as a list all the properties data which fall into each interval, as in Figure 6. The intervals may be chosen at convenient values, chosen as in the example above to store roughly the same number of data items in each interval. They may be chosen for physical reasons; the simplest example might be values of temperature broken into 3 intervals:

\[
<0^\circ \text{C}, 0 \text{ to } 100^\circ \text{C} \text{ and } >100^\circ \text{C}
\]

to represent the 3 phases of water.

The resulting structure would consist of a set of predetermined intervals of values and associated with each interval, a list of values which might be short or long. This is similar to the structure of an inverted file in the Inverted File Model [10] of a database; it is also similar to the inverted file structure in a Document Database (Information Retrieval) [11] system where an interval of values is replaced by a word in a dictionary. So the structure is well known and can be very efficient.

If the addresses of the intervals are computed from an algorithm, such as a division algorithm, the intervals will have a regular spacing. However, if the users of a database want the intervals to vary in size and position to suit some need and a computed algorithm is not appropriate, the highest (or lowest) value in each interval can be indexed using a \( B^+ \)-tree. This is efficient and flexible. We call it an "Interval Tree".

This type of index can be used effectively for discrete values. It is equally effective in storing range items. If a data item consists of a wide range, covering 2 or more intervals, then all we have to do is to add an address of the appropriate record to all of the intervals. Since the number of data items per interval can be as large as we need, there is no difficulty in repeating a data item several times in the interval lists.

5 EXPERIMENTS

We have described above two methods of indexing numerical properties data: a \( B^+ \)-tree and an Interval Tree. Although the Interval tree is prepared for data items expressed as ranges of values, both appear to be appropriate for data which are expressed as single values and most data are recorded in this form. We therefore decided to compare the two methods by simulating a real database system storing materials properties.

Briefly we used a random number generator to generate a set of \( N \) properties in a fixed range, each assumed to be associated with a single record. The records were then indexed using the two methods and the methods compared to find the time and, more important, the number of disk accesses required.

Figure 5:
Example of overlapping ranges. Each dot or line in the graph represents a data item in the database. A search for the value \( x \) should find 4 data items, but some may be ranges with centres far distant from \( x \).
TABLE 3: COMPARISON OF DISK ACCESSES FOR DIFFERENT RANGE SEARCHES IN A B*-TREE AND IN INTERVAL TREES OF 3,000 AND 30,000 RECORDS WITH KEYS UNIFORMLY DISTRIBUTED IN [0,1]. THE INTERVAL TREES CORRESPOND TO 30 AND 300 EQUISPACED INTERVALS

<table>
<thead>
<tr>
<th>File Size</th>
<th>Range B*-tree</th>
<th>Ordered Interval Trees</th>
<th>Unordered Interval Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Ordered</td>
<td>Unordered</td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td>0.001 2.12</td>
<td>2.06</td>
<td>2.39</td>
</tr>
<tr>
<td></td>
<td>0.01 2.91</td>
<td>2.88</td>
<td>5.11</td>
</tr>
<tr>
<td></td>
<td>0.1 10.74</td>
<td>10.56</td>
<td>32.10</td>
</tr>
<tr>
<td>30000</td>
<td>0.001 2.88</td>
<td>15.38</td>
<td>3.79</td>
</tr>
<tr>
<td></td>
<td>0.01 10.62</td>
<td>23.80</td>
<td>11.86</td>
</tr>
<tr>
<td></td>
<td>0.1 88.03</td>
<td>102.25</td>
<td>89.83</td>
</tr>
</tbody>
</table>

TABLE 4: COMPARISON OF DISK ACCESSES FOR PROXIMITY SEARCHES IN A B*-TREE AND IN INTERVAL TREES FOR FILES OF 3000 AND 30000 RECORDS WITH KEYS UNIFORMLY DISTRIBUTED IN [0,1]. THE INTERVAL TREES CORRESPOND TO 30 AND 300 EQUISPACED INTERVALS

<table>
<thead>
<tr>
<th>File Size</th>
<th>Nearest Neighbours* B*-tree</th>
<th>Ordered Interval Trees</th>
<th>Unordered Interval Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Nearest neighbours*</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ordered</td>
<td>Unordered</td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td>±3 2.16</td>
<td>2.06</td>
<td>2.69</td>
</tr>
<tr>
<td></td>
<td>±8 2.45</td>
<td>2.35</td>
<td>3.66</td>
</tr>
<tr>
<td>30000</td>
<td>±3 2.16</td>
<td>15.04</td>
<td>3.05</td>
</tr>
<tr>
<td></td>
<td>±8 2.49</td>
<td>15.20</td>
<td>3.36</td>
</tr>
</tbody>
</table>

*Note: the term ±3 means that one search for 3 values above and 3 values below the search value.

accesses needed to find a record in a large number of different searches. It is the number of disk accesses which usually determines the response time of a database system.

The size of the file was varied from 1,000 to 100,000 records and the distribution of records over the interval was varied between a uniform distribution and a Zipfian (K/Rank) distribution (the distribution of words or names in a language) [12].

We give results for files of 3,000 and 30,000 records. For the Interval Trees, the results are for trees of 30 and 300 equispaced intervals.

In the B*-tree the data records are ordered and stored in data blocks and indexed by a set of short index records, which include the property value and a pointer to the data blocks. In the Interval Trees the data records in each interval are stored on a linked list of data blocks. The first block in each interval is indexed in a similar manner to the data blocks in the B*-tree. A maximum of 30 records can be stored in a data block.

On a computer with a large solid state memory and relatively small data file of materials all of the B*-tree or Interval tree would be stored in the central store and no disk accesses would be necessary to traverse the tree, apart from the disk accesses and processing time needed to read the tree into the store. In our experiments we have assumed that we are dealing with a large data file and that all of the B*-tree or Interval tree must be held on disk.

We have also defined two forms of Interval tree: ordered and unordered. In the Unordered Interval Tree the storage space utilization was kept at a maximum since each record inserted into a list was placed in the next available space in the list and a new data block was only added to the end of the list when the current last block was full. For the Ordered Interval Tree, however, when one of the blocks of a list was full, it was split in two and the records divided among the two blocks.

Each of the results given in Tables 3-4 are the average of 1,000 searches. Each size of database was built with 5 different data sets and 200 searches were performed in each. The data for the searches was chosen by a random number generator. Proximity Searches for 3 (and 8) nearest
neighbours above and below the search value were performed along with various Range Searches. Results are not given for point searches, because clearly the B*-tree is a better and more flexible structure than the Interval Tree for these searches.

Table 3 is a comparison of disk accesses for various range searches in a B*-tree and in the two Interval Trees, with keys uniformly distributed. It shows that the performance of the Interval Trees is dependent on the number of intervals chosen. In most cases the B*-tree is the most efficient of the structures. For small ranges the Ordered Interval Tree is better than the Unordered Interval Tree. However, as the range size increases the Unordered Tree performs better and even outperforms the B*-tree.

Table 4 is a comparison for answering proximity searches in a B*-tree and in the two Interval Trees. The B*-tree is generally still better.

**Multidimensional Indexing**

Much of the searching performed in a materials database involves more than one attribute, i.e. associative searching, as mentioned in Section 3 with regard to bounded searches. Traditionally the most widely used structure for indexing multidimensional keys is a series of Inverted Files [10]. Clearly we require a better structure for associative range, proximity, bounded searches etc., a structure that preserves as much as possible the order defined on each attribute domain. The Grid File [13] is a promising alternative and this is currently being investigated with the possibility of using a similar Multidimensional Interval Tree.

**CONCLUSION**

We have shown how data used by engineers is different from that stored in a conventional database, i.e. imprecise nature of data gathered from physical experiments and engineering tests. As a result, searches in general will retrieve records with a range of key values. The B*-tree is a very efficient structure if all the data is stored as specific values. We have also introduced the Interval Tree to enable us to store data in the form of ranges. The tests have shown that the performance of the Interval Tree can be as good as that of the B*-tree, if the number of intervals used is chosen well. Therefore if much of the data is in the form of ranges we recommend the use of the Interval Tree.

**REFERENCES**