A Database Architecture for Aggregate-Incomplete Data

David S. Bowers
School of Computing Studies and Accountancy, University of East Anglia, Norwich NR4 7TJ, UK

The problem is addressed of data which are not only incomplete, but also aggregate in nature. Some difficulties associated with maintaining the consistency of such data are illustrated. These problems arise in particular when it is required that both aggregate and simple data values should be stored. An architecture is proposed for a database management system to handle this class of data, which is termed *aggregate-incomplete*. A generalized materialization strategy is defined on the basis of the proposed architecture, and it is shown that the consistency problems are reduced to manageable proportions.

1. INCOMPLETE DATA AND AGGREGATE-INCOMPLETE DATA

Incomplete data in database systems present a number of problems which have been receiving attention for some time. These problems are concerned, not so much with the representation of unknown values or 'nulls', but with how they should be treated once stored.

Many designs for database management systems can cope with 'null' values at least to the level of returning a 'null' value if an attempt is made to retrieve a data item whose value is unknown, although they may not always warn the user appropriately.[1,2]

The problem is more complicated if some arithmetic operation is to be performed on the contents of a data field for which some instances are 'null'. For example, given an entity EMPLOYEE with an attribute SALARY, the calculation of an average salary for all employees is trivial if the data are complete. If, however, some of the SALARY fields are 'null', it is necessary to define a new way of calculating the average, which may not be obvious to the user.[3,4]

Further problems arise where a field in the database for which at least one occurrence is 'null' is to be used as a key for selecting data. In the previous example, it is not obvious how a search criterion of 'SALARY > 10000' would be applied within a database containing 'null' values in the SALARY field.[1,3–5]

Perhaps the problem arising from 'null' values which has attracted most attention is that where the field containing the 'null' value is used for logical navigation, and, in particular, where it participates in a relational JOIN operation. Various solutions, such as the 'MAYBE-JOIN' and the 'OUTER-JOIN', have been proposed for the problem of information-loss in such a join,[4,6,7] but the question is still to be resolved.

A final class of problems arise when consistency constraints are expressed in terms of fields which may contain 'null' values. Consider, for example, a constraint that the sum of salaries for all EMPLOYEEs must not exceed some salary budget; this is non-trivial to implement if, for some EMPLOYEEs the SALARY field is 'null'.

It is to this last class of problems that this paper is addressed. Approaches have been proposed which either assume partial knowledge of the null value,[5] or attempt to deduce, from the statistical properties of the known data, some knowledge of the null values.[8] Rather than following either of these approaches, the discussion of this paper is addressed towards the problems of storing aggregate-incomplete data, and reconciling the set of non-null data values.

In section 2, an example of such a problem is developed, based on a class of data termed *aggregate-incomplete*. Section 3 introduces a number of basic concepts and terms which may be used to describe aggregate-incomplete data. A database architecture is proposed in Section 4 which goes some way towards solving some of the problems of representing such data in a database management system. Section 5 discusses briefly the generalization of the type of consistency constraint for which the proposed architecture is appropriate. Finally, Section 6 discusses the implementation of a system based on the architecture for a real life problem.

A mathematical treatment of aggregate-incomplete data will be the subject of a forthcoming paper.[9]

2. A POPULATION DATABASE

In this section, a data model for a simple population database is developed, to illustrate the aggregate nature of the class of data which is being considered, and some of the problems which arise. Then, additional difficulties are considered which are experienced if such data are incomplete—that is, if the data are aggregate-incomplete.

2.1 An example of aggregate data

A trivial population database for Great Britain might contain four data items, namely, population figures for each of England, Scotland and Wales, and for the whole of Great Britain. The last value, being the arithmetic sum of the first three, is derived data.

If it is wished to retrieve from such a database the value of the population for Great Britain, there are two choices: to search directly for the 'derived data', 'population of Great Britain', or to retrieve the three values for the country populations and form their sum. It would seem reasonable to expect these two approaches

CCC–0010–4620/84/0027–0294$03.50

to give the same result. (Inconsistencies due to machine rounding errors are ignored for the purpose of this paper.) Thus, there is an implicit consistency constraint that the sum of the populations for England, Scotland and Wales should be equal to any value stored as the population of Great Britain. Thus far, this is a standard problem of storing derived data in a database.

The trivial database is complicated slightly if the population figures are broken down into 'male' and 'female'. Now, three values must be stored for each of the three countries and also for Great Britain—i.e. male, female and total in each case. The set of derived data now comprises total population for each of the three countries, and male, female and total population for Great Britain.

Derived values such as 'total population of England' or 'male population of Great Britain' are analogous to the single derived value in the trivial case above. There are just two ways of materializing such values, and they should be equivalent.

However, the derived value 'total population of Great Britain' in the enlarged database may be realized as:

(i) a single stored value
(ii) the sum of 'male population of Great Britain' and 'female population of Great Britain'
(iii) the sum of the total populations of the three countries
(iv) the sum of male and female populations for each of the three countries
(v) any combination of the above which avoids double counting, e.g. total population of England + total population of Wales + male population of Scotland + female population of Scotland. (There are eight permissible combinations.)

Thus, the database management system must choose between 12 strategies for materializing the value of 'total population of Great Britain'. It must apply consistency constraints to ensure that all 12 are equivalent, as far as the user is concerned. Further, it must be aware that:

(a) female population of Great Britain + male population of England + male population of Wales + total population of Scotland, and
(b) male population of Great Britain + female population of England + female population of Wales

are both invalid, as are a large number of other combinations. Summation (a) involves double counting of the female population of Scotland, whereas (b) simply ignores that part of the total population.

If a further level of disaggregation were to be introduced into the database, for example, by dividing the population figures between adults and children, the problem would become even more complicated. The basic data would then comprise 12 items such as, 'female child population of England', with all the other data items mentioned thus far being derived data. The derived data would comprise some 14 data items with 2 data materialization strategies, 3 with 8 strategies, 4 with 12, and 1 ('total population of Great Britain') with well over 500 logically equivalent materialization strategies.

2.2 Aggregate-incomplete data

Clearly, if one were dealing with complete data, allowing such a multiplicity of materialization strategies would not be considered. Probably, only basic (non-derived) values would be stored, requiring actual summations to be performed to retrieve derived data. Derived data might conceivably be stored once calculated, but it would be foolish to permit them to take part in arbitrary summations for other derived data.

This is not to suggest that it would be impractical to have a set of static mappings between data items, so that a particular derived data item would always be formed from a given set of data values, some of which might themselves be derived. This is no different from basing views of the underlying data on other views. However, the foregoing discussion has illustrated that there is a consistency problem of exponential proportions if materialization strategies for derived data may be based on arbitrary sets of derived or atomic data, with the strategy selected effectively at random.

Consider now the problem where data are incomplete. To return to the trivial example containing just three basic values and one derived value, it would seem that there would be essentially two choices if the values of 'population of Great Britain' and 'population of England' were known, and both the other values were 'null': not to store the derived value, 'population of Great Britain', on the grounds that it would make things difficult for the database management system, or to insist on storing it, on the grounds that it is a useful number.

The discussion in this paper assumes that the latter choice is preferred. (The possibility of restructuring the data to introduce, for example, a new data item, 'population of Wales and Scotland', is ignored, since such a solution does not readily generalize.)

If the two values 'population of Great Britain' and 'population of England' are stored, then it is necessary to return a 'null' value if either the 'population of Scotland' or the 'population of Wales' is requested. This is an example of the first kind of incomplete data problem outlined in the introduction to this paper.

If, at some future date, a value becomes available for 'population of Scotland', then it is necessary to check that the value of 'population of Scotland' is currently 'null', and then insert the new value.

When a value for 'population of Wales' subsequently becomes available, things are more difficult. The implicit consistency constraint is that the sum of 'population of England', 'population of Scotland' and 'population of Wales' must be equal to the stored value of 'population of Great Britain'. Hence, before the value for 'population of Wales' is entered, not only must the value currently stored be 'null', but the new value must also be consistent with those already stored. Should it not be consistent, it is necessary to reject either the new value or one of the values already stored.

In the simple case outlined above, the value being considered is part of just one aggregate value—'population of Great Britain'. As it has been shown, the number of ways of materializing any aggregate quantity increases rapidly as the number of levels of aggregation increases. Unfortunately, this applies also in reverse—as the number of levels of aggregation increases, so too does the number of materialization strategies in which any given value may partake.

Thus, in order to enter a new value into an arbitrarily complex database containing aggregate-incomplete data, every possible way must be considered in which the new value must be consistent with those already present. This means considering *every* materialization strategy in which the new value could participate, and, if the data are sufficiently complete for the materialization to be performed, comparing the materialized value with any stored aggregate value. Further, if the new value is itself an aggregate, all possible materializations of that value itself must be considered, and checked against the new value. Finally, all materializations in which the new value can participate, and for which an aggregate value is *not* stored, must be considered, to check that the aggregate value obtained using the new value is consistent with any alternative materialization of that aggregate value. This is no trivial task.

It would appear, therefore, that there is a choice when designing a database for aggregate-incomplete data. It can be decided not to store any aggregate values, thus inconveniencing the user, for whom an aggregate value may be just as important as 'atomic' data, or alternatively to store all aggregate values, in which case horrendous consistency problems may arise. It is not practical even to restrict the system to materializing aggregates only from atomic values to reduce the consistency problem, because it cannot be guaranteed that all the atomic values will be present if the data are incomplete.

In the sections that follow, a third strategy is explored, which is essentially a compromise between these two approaches. The strategy will allow the system to store some, but not all, aggregate values. By defining a generalized materialization strategy which is always unique, the design reduces to a minimum the consistency problem.

3. HIERARCHIC ATTRIBUTE SETS

Before discussing the architecture proposed for aggregate-incomplete data, it is useful to introduce the concept of a *hierarchic attribute set*. Examples of such sets, drawn from the population database discussed in the previous section, would include {'Great Britain', 'England', 'Scotland', 'Wales'} and {'total', 'male', 'female'}.

The members of these sets are *attribute* values which describe particular numbers in the population database. The sets are *hierarchic* in the sense that each element is either a superset or a subset of another member of the set. Hence, the term *hierarchic attribute set*.

Hierarchic attribute sets may display several levels of hierarchy; for example, the attributes 'England', 'Scotland' and 'Wales' could be subdivided into their appropriate counties.

Any two elements of a hierarchic attribute set must be disjoint, unless one is a strict subset of the other. For example, 'England' and 'Scotland' are disjoint, whereas 'England and Wales' and 'Scotland and Wales' would be overlapping, and hence invalid as members of the same hierarchic attribute set.

Further, if any element of a hierarchic attribute set is a subset of a second element, then there must be other elements which, together with the first, are equivalent to the second. For example, if the hierarchic attribute set contains 'Great Britain', 'England' and 'Scotland', it must contain also 'Wales' (or some collection of disjoint elements whose union is equal to 'Wales').

Finally, the example discussed in Section 2 is described by two independent hierarchic attribute sets, which are regarded as *orthogonal*, in the sense that any element of the first such set may appear in conjunction with any element of the second set. Thus, valid combinations could include 'England male', 'England total', 'Scotland female' and 'Great Britain male'.

4. A DATABASE ARCHITECTURE FOR AGGREGATE-INCOMPLETE DATA

Having dealt with the potential difficulty of the problem of aggregate-incomplete data, and introduced a few basic terms, it is appropriate to consider a database architecture which can go some way towards offering a solution to the problem.

4.1 The proposed architecture

The architecture which is described below is a compromise between the conflicting requirements imposed by aggregate-incomplete data. Any such compromise is bound to fall short of the mark in at least some respects. Nevertheless, it is proposed as a design which is at least useful, and which may be amenable to improvement given further study.

The proposed architecture is developed in terms of data described by a single hierarchic attribute set, and then extended for data described by multiple hierarchic attribute sets.

Consider the single-level hierarchic attribute set, {Great Britain = England + Scotland + Wales}. It is proposed that data described by such an attribute set should be stored in a logical tree. That is, that there should be a (root) node corresponding to 'Great Britain', and a set of (leaf) nodes corresponding to 'England', 'Scotland' and 'Wales'.

It is proposed that the logical tree should be *pruned* to the minimum configuration required to store the known data. That is, if there is no value to be stored for *any* descendant of a given node, then that node should appear as a leaf in the logical tree. If values should subsequently become available for any descendant node, then it is proposed that the tree should be extended as required.

It is proposed that there should be a consistency constraint within the tree so that, should values be known for all the daughter nodes of any node in the tree, then they may be stored *only* if their arithmetic sum is equal to the value for the parent node.

With this architecture, a value may be retrieved for any node within the logical tree, either by reading the value stored at that node, or by summation of the values of daughter nodes, provided that the data are sufficiently complete.

4.2 The population example

Applying these proposals to the population example, if just a single value for 'Great Britain' is known, then a logical structure of a single node would be stored—with no daughters—containing that value.

If a value for 'England' were subsequently to become known, then three daughter nodes would be added to the original node, the 'England' value placed in one, and 'nulls' in the other two.

Adding a value for 'Scotland' would pose no problems, as the null stored for the 'Scotland' node would simply be replaced by the new value.

The addition of a value for 'Wales' would proceed as follows:

```
If the sum of values for 'England', 'Scotland' and
    'Wales' is equal to that for 'Great Britain'
then insert the new value for 'Wales';
else reject the new value (i.e. give an error message).
```

If at some time the logical tree were to contain values for 'England', 'Scotland' and 'Wales', but none for 'Great Britain', a value input for 'Great Britain' could still be checked for consistency. The action taken would be to compare the input value with the sum of the stored values, and confirm them to be correct if they were equal, or to be erroneous if they were not.

If the hierarchic attribute set were later extended to include the Welsh counties ({Wales = Clwydd + ⋯}) as subsets of 'Wales', and values input for those Welsh counties, then daughter nodes would be added to the 'Wales' node, and the same consistency constraint procedure applied as described above. If no values were stored for 'Wales' or 'Great Britain', the former value would be materialized as the sum of the values for the Welsh counties, and the latter as the sum of those for 'England', 'Scotland' and the Welsh counties.

4.3 Multiple hierarchic attributes

The next stage is to consider how the proposed architecture might be extended to accommodate data described by multiple hierarchic attribute sets. The problem, of course, is that, if the attribute sets are truly independent, then they ought to be represented in the logical tree by orthogonal divisions. Such a network structure, apart from being hopelessly complicated, would not overcome any of the combinatorial problems discussed earlier.

The solution which is proposed is that the trees representing the attribute sets should be joined together into a single tree. Essentially, any node in the logical tree will correspond to a *set* of attribute values, exactly one being selected from each of the hierarchic attribute sets. The presence of daughter nodes at the node in question may represent the division of any *one* of those attribute values into subsets. Subdivision of two or more attribute values must be effected in sequence, so that the same type of split may be replicated at a number of distinct nodes in the tree. For example, a 'male'–'female' split may occur at nodes corresponding to each of the 'England', 'Wales' and 'Scotland' nodes.

With such a structure, the same rules may be applied for data materialization and consistency constraints as were applied above to a single-attribute data structure. The problem of materialization will, however, be slightly more complicated for quantities which do not correspond to nodes in the actual configuration of the tree, but could have done in an alternative configuration. For such values, a special selective summation operation must be defined, which is more complicated than a mere summation over the node of a tree.

4.4 The materialization strategy

The strategy is developed in terms of a multiple attribute tree derived from the three hierarchic attribute sets introduced in Section 2, shown in Fig. 1. A possible configuration for the tree is shown in Fig. 2, with the labels at each node showing the three attribute values for the node, and an asterisk indicating the presence of a stored value.

```
{Great Britain =
    England
    Scotland
    Wales}
{total =
    male
    female}
{people =
    adult
    child}
```

Figure 1. Three hierarchic attribute sets.

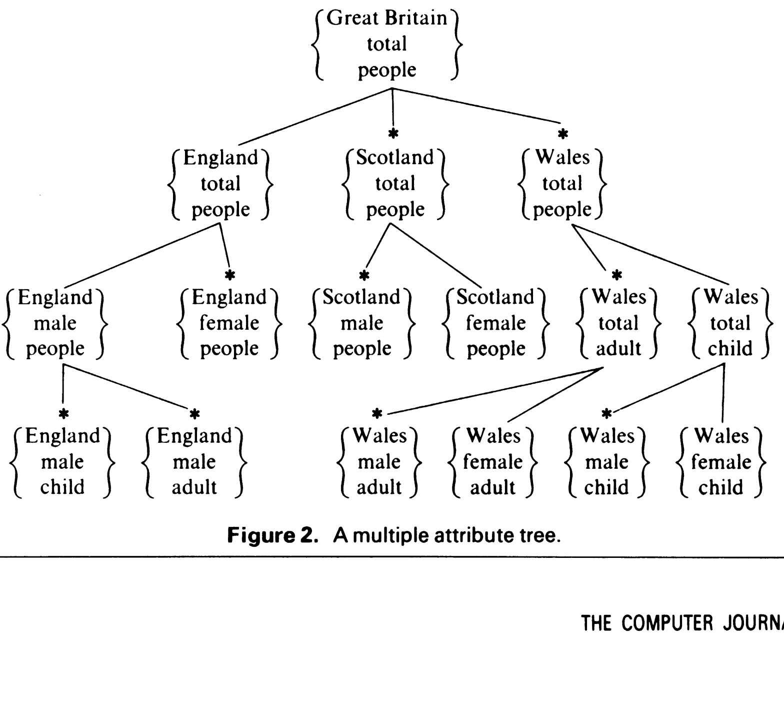

Figure 2. A multiple attribute tree.

Materialization of some values, such as 'England total people', consists merely of a summation over the specified node, if no value is actually stored at the node. The algorithm can be expressed as the following recursive procedure:

```
procedure sum_over_node (current_node)
  if a value is stored at current_node then
    sum_over_node = stored value
  elseif current_node has daughter nodes then
    sum = 0;
    for each daughter_node do
      sum = sum + sum_over_node (daughter_node)
    enddo
    sum_over_node = sum
  elseif no value stored and no daughter nodes then
    error
  endif
endproc
```

An error exit must be followed if any node required in the summation is a 'leaf' node (i.e. it has no daughter nodes) with no value stored in it. In other words, if there is insufficient data to materialize the value, then the algorithm must fail.

A value such as 'Great Britain male', however, is more difficult. This requires a selective summation of the values corresponding to the nodes 'England male people', 'Scotland male people', 'Wales male adult' and 'Wales male child'.

The materialization strategy is unique, given the configuration of the logical tree, and can be defined precisely in terms of the action to be taken at each node while scanning the tree. There is no question of having to make an arbitrary choice between a (large) number of strategies.

If the set of attribute values for the required value is denoted $\{A\}$, and the set of attribute values for a given ('current') node is denoted $\{B\}$, then the materialization strategy may be expressed as a procedure to be followed at any node, starting with the root node. In the following expression of the algorithm, it is assumed that the 'current' node represents a division of the ith attribute value.

```
procedure selective_sum (current_node, $\{A\}$)
  /* required value described by $\{A\}$
     current node described by $\{B\}$
     current node is division of ith attribute */
  if $\forall j, Aj \supseteq Bj$ then
    selective_sum = sum_over_node (current_node)
  elseif current_node has no daughters then
    error
  elseif $Ai \subset Bi$ then
    select appropriate daughter node;
    selective_sum = selective_sum (daughter_node, $\{A\}$)
  else
    sum = 0;
    for each daughter_node do
      sum = sum + selective_sum (daughter_node, $\{A\}$)
    enddo
    selective_sum = sum
  endif
endproc
```

Unless the value corresponding to the 'current node' is either the required value or a subset of it, then a further selective summation is effected. Essentially, the first branch from the node is taken, and whatever is required from that branch is retrieved, and the selective summation node is returned to. As long as there are more branches, the next in sequence is selected, and the required contents retrieved, and so on. When there are no more branches, the summation is complete, and either the materialization is complete, or an earlier selective summation node can be reconsidered.

As with the procedure *sum_over_node*, the algorithm fails if the data are insufficiently complete to materialize the required value. This could be either because a node under consideration has at least one attribute value which is a superset of that for the required value, and yet it is a 'leaf' node, or because a simple summation over a node fails. In either case, the data are insufficient to construct the required value, and no other materialization strategy could succeed.

The algorithm requires at most a complete traversal of the logical tree, after which, either the value will have been formed, or it will be known not to be available. The total size of the logical tree grows only as the product of the sizes of the single attribute trees, rather than showing the exponential behaviour illustrated in the previous sections.

Figure 3 shows the sequence of nodes visited, and the action taken at each, in order to materialize values corresponding to 'England total people' and 'Great Britain male people' from the tree configuration of Fig. 2.

(a)	{England total people}	
	{Great Britain total people}	select daughter node
	{England total people}	sum over node
(b)	{Great Britain male people}	
	{Great Britain total people}	sum over daughter nodes (1), (2) and (3)
(1)	{England total people}	select daughter node
	{England male people}	sum over node; add to sum
(2)	{Scotland total people}	select daughter node
	{Scotland male people}	sum over node; add to sum
(3)	{Wales total people}	sum over daughter nodes: (i) and (ii)
(i)	{Wales total adult}	select daughter node
	{Wales male adult}	sum over node; add to subtotal
(ii)	{Wales total child}	select daughter node
	{Wales male child}	sum over node; add to subtotal
		add subtotal to sum.

Figure 3. Materialization of aggregate values.

4.5 Consequences of the architecture

A few comments must be made about the effect of the configuration of the logical tree, or the order in which the attribute values are subdivided. The main restriction is in the set of aggregate values which may be stored.

Clearly, a value can be stored only if there exists a corresponding node in the tree. Thus, for example, no value could be stored in the multiple attribute tree considered above for 'Great Britain male'. Hence, the architecture fails to allow the user to store *all* possible aggregate values. However, it does not prevent him from materializing such values—if the data are sufficiently complete—nor from checking them for consistency, since the materialized value may be checked against an input value.

Nevertheless, it does allow him to store aggregate values in many cases. Indeed, the database management system discussed in Section 6 selects the configuration of the logical tree on the basis of which aggregate values are present in the initial input data. Although the architecture may not allow all aggregate values to be stored, it does allow sufficient to be stored to be useful.

Further, the architecture as proposed does not require the configuration of the tree to be homogeneous. For example, the tree shown in Fig. 2 has 'Wales total people' subdivided between 'adult' and 'child', rather than the division between 'male' and 'female' which appears at the corresponding nodes for 'England' and 'Scotland'. The algorithm given for materializing values is quite capable of dealing with such 'inhomogeneous disaggregation'.

Since the materialization algorithm is well defined, given the configuration of the logical tree, the problem of selecting from an immense variety of materialization strategies is avoided. Further, the materialization strategy can terminate immediately it is found to be impossible to materialize the required value—there is no possibility that an alternative strategy might be successful. Finally, if the materialization strategy succeeds, then the value obtained must be complete—there is no chance that some component has been omitted.

5. GENERALIZATION

The consistency constraints associated with aggregate values, as discussed in the development of the proposed architecture, are purely arithmetic. In essence, certain values which could be stored in the database must be (arithmetically) equal to values formed from sets of data items; the aggregate value is thus a single value corresponding to a set, as well as being a value in its own right.

The proposed architecture requires only this essence of the consistency requirement—that some set property matches the corresponding aggregate value derived from the set elements. Arithmetic summation is just one example of this class of requirements, but the architecture could, in principle, support any other consistency requirement of the same class, requiring only minor modifications to the implementation.

Various types of aggregation are possible. These include *combinatorial* operations, such as summation or product, *selective* operations, such as maximum or minimum, and *containment* operations, such as 'all elements are factors of the aggregate'. Similar operations on textual data would also be possible.

Provided that the definition of the hierarchic attribute sets is unique, then the proposed architecture should be suitable for aggregate-incomplete data subject to consistency requirements corresponding to any of these aggregation operations. Further, it should be possible to cope with multiple set values derived from a variety of aggregation operations.

More complex aggregation operations, in which elements of particular sets are treated in an inhomogeneous manner to generate the aggregate, or where two or more sets of values take part in the formation of the aggregate, could perhaps be accommodated, given some extensions to the architecture; however, this is an area for further study.

6. IMPLEMENTATION

A database management system based on the proposed architecture has been implemented for the Energy Research Group of the Cavendish Laboratory, Cambridge. The system has been in operation for some two years, serving a number of research projects in energy policy and related areas.

The design and implementation of the database management system for aggregate-incomplete data will be discussed more fully in a forthcoming paper.[10]

6.1 The data

The data stored by the system consist of energy production and consumption figures, population data and a number of economic indicators for each of 186 countries, over a 30 year time period. The raw data were derived from published statistical series, and were aggregate-incomplete in nature. Many of the aggregates were derived independently, and needed to be checked for consistency. Out of a possible 300,000 data values, about half were unknown when the data were loaded initially. This proportion has been reduced significantly during the period that the system has been in use, but by no means eliminated.

The data are described by three hierarchic attribute sets, corresponding, respectively, to geographical regions, economic sectors and class of fuel. A fourth hierarchic attribute set is used to represent to which year a particular value refers. This hierarchic attribute set corresponds to aggregation over time, to produce cumulative values, although the raw data did not contain any such aggregates.

6.2 Modifications to the architecture

Since the data were real values, rather than integers, and subject to measurement and reporting errors, the requirement for equality in the consistency constraint had to be relaxed to 'equality within a specified tolerance'.

Two minor extensions to the architecture were required to accommodate the data, but neither altered the character of the system fundamentally.

The first concerned data, such as gross national product, measured in national currency units, which, although they could be described by combinations of elements of the hierarchical attribute sets, could not be aggregated directly. (Adding dollars to pounds is meaningless.) Automatic aggregation is suppressed for such data.

The second concerned data for which certain combinations of the attribute values were not possible. Rather than recording the corresponding data value as unknown, which would make aggregation impossible, or as zero, which would imply the possibility of a non-zero value, a 'dummy' value can be stored, which allows correct aggregation while being invisible to the user.

6.3 User experience

Energy policy studies often require aggregate values, being sums over geographical regions, economic sectors, fuels, or some combination of these. All such aggregates are readily available using the system, whether or not they were present in the original data, provided that there is sufficient information to form them. Such derivations would previously have required exhaustive manual searches of the data, as suggested in Section 2; the implemented system, with its well-defined materialization algorithm, has been found to be a great asset.

Restructuring of the logical trees has not been a problem. This would be necessary if the configuration of the tree prevented the storage of some aggregate values, whereas an alternative configuration could accommodate them without preventing the storage of aggregate values present in the original configuration. If the aggregate values are input before the 'atomic' data, the tree is automatically configured to accommodate the aggregates, provided that the different aggregates do not require incompatible configurations, in which case a conscious decision is needed as to which are to be stored. Since the types of aggregate present in the data were known when the data was loaded, a need to restructure the tree has not arisen.

Restructuring would also be necessary if the definition of the hierarchic attribute sets were modified fundamentally. Minor extensions to the hierarchic attribute sets, such as the subdivision of a particular (atomic) element, can be accommodated readily, since the system is concerned with logical rather than physical trees. Although the system did need to be reloaded once, when the size of the temporal hierarchic attribute set was increased, it was found to be a trivial operation.

To date, the system has been found to be more than adequate for its task supporting a research programme.

7. CONCLUSION

A class of data termed *aggregate-incomplete* has been defined. Such data are described by *hierarchic attribute sets*. It has been shown that, if aggregate values are stored for such data, then the problem of materializing some aggregate values is complicated by an immense variety of materialization strategies.

Having asserted that it is desirable, if not essential, to store aggregate values if data are incomplete, an architecture based on a tree structure has been proposed for a database to hold aggregate-incomplete data. A materialization algorithm has been defined which is unique, and which will always succeed if sufficient data are stored. At most, the algorithm requires a complete traversal of the data tree, a task which will often be insignificant in comparison with attempting all possible materialization strategies. It has been suggested that the architecture could be appropriate for data subject to a number of types of consistency requirements. It has been asserted that, although the architecture fails to meet all the requirements of aggregate-incomplete data in full, it does satisfy all of them sufficiently for it to be useful as the basis for a database management system. Finally, the implementation of a system based on the architecture in a real environment has been outlined.

Acknowledgements

I am grateful to Professor P. M. Stocker and to Dr P. A. Dearnley for their advice and comments during the preparation of this paper. I would like also to thank the referees for their helpful criticisms of the content.

REFERENCES

1. CODASYL. *Data Base Task Group Report* (1971).
2. D. D. Chamberlain and R. F. Boyce, SEQUEL—a structured English query language. *Proceedings ACM-SIGFIDET Workshop*, Ann Arbor, Michigan, May (1974).
3. D. D. Chamberlain *et al.* SEQUEL-2: a unified approach to data definition, manipulation and control. *IBM Journal of Research and Development* **20**, 560–575, November (1976).
4. C. J. Date, Null values in database management systems. *Proceedings of 2nd National Conference on Databases*, Bristol, July (1982).
5. W. Lipski, On semantic issues connected with incomplete information in databases. *ACM Transactions on Database Systems* **4** (3), 262–296 (1979).
6. M. Lacroix and A. Pirotte, Generalised joins. *ACM SIGMOD Record* **8** (3), 14–15 (1976).
7. E. F. Codd, Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems* **4** (3), 397–434 (1979).
8. E. Wong, A statistical approach to incomplete information in database systems. *ACM Transactions on Database Systems* **7** (3), 470–488 (1982).
9. D. S. Bowers, A mathematical treatment of aggregate incomplete data. (To appear.)
10. D. S. Bowers, The design and implementation of a database management system for aggregate-incomplete data. (To appear.)

Received May 1983