Bulletin of the Technical Committee on

Data Engineering

March 1997 Vol. 20 No. 1

IEEE Computer Society

Letters

Special Issue on Supporting On-Line Analytical Processing

Conference and Journal Notices

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems.

Distribution

IEEE Computer Society
1730 Massachusetts Avenue
Washington, D.C. 20036-1992
(202) 371-1013

Letter from the Editor-in-Chief

Bulletin and Technical Committee

This begins my fifth year as Editor-in-Chief of the Data Engineering Bulletin. This has been both a rewarding experience and a time-consuming one. I have been fortunate to have worked with many fine issue editors over the years, without whose efforts, the Bulletin simply would not have happened. I want to thank them all and express the hope that my good fortune will continue in the future.

The problem of assuring funding for the Bulletin was with us from the time the Bulletin was revived in December, 1992. It continues. Fortunately, every year thus far, including this year, the IEEE Computer Society has agreed to fund us yet again. This is gratifying but also potentially undatable for the longer term. Literally, at the beginning of each calendar year, I ask whether our funding continues. So far, so good, but....

Meanwhile, the Technical Committee on Data Engineering, aside from the Bulletin, has become relatively dormant. I have not seen more than two or three messages about TC affairs over the last year. This is not a good situation. Does the TC have a role outside of the Bulletin. What about the SIGMOD model? Is there room for/do we need two database organizations- one for the ACM and one for the IEEE, or is this just an historical accident? This is more than a rhetorical question. I am prepared to devote scarce Bulletin pages to letters on this subject. I would encourage you to write with your suggestions for the future of the TCDE.

This Issue

Daniel Barbará is the issue editor for this special issue on on-line analytical processing (OLAP). OLAP has become in a few short years one of the major applications of database technology, impacting indexing, query processing, and caching strategies. This issue nicely captures some of the excitement of the OLPA field, both commercially and in the research community. Daniel has done a fine job in assembling the issue. There is much to learn in the reading of the issue.

David Lomet
Microsoft Corporation

Letter from the Special Issue Editor

On-Line Analytical Processing (OLAP), is a technology that describes applications that require multidimensional analysis of data. OLAP allows users to view aggregate data (e.g.,sales) along a set of dimensions (e.g., store, products, months) and hierarchies (e.g., day, month, year). OLAP data is frequently organized in the form of a *data cube*, which is simply a multidimensional hierarchy of aggregate values. OLAP functionality enables analysts to do calculations applied across dimensions and hierarchies, trend analysis, viewing of subsets of the data, navigation among levels of data ranging from the most summarized (roll-up) to the most detailed (drill-down), and other operations even on very large data bases. Although there are already a variety of products in the market that support OLAP, the field is rich with research opportunities and thus, has attracted the attention of researchers in both academia and industry.

This issue of IEEE data Engineering Bulletin focuses on recent efforts in developing techniques to efficiently support OLAP. In one way or another, all the papers in this issue deal with the two key issues in OLAP: performance and storage space. The first issue arises from the fact that queries in OLAP need to be interactive. The second issue is a factor since data cubes are usually very large (sometimes too large to be completely materialized).

Our first paper written by Prasad M. Deshpande, Jeffrey F. Naughton, Karthikeyan Ramasamy, Amit Shukla, Kristin Tufte and Yihong Zhao, reviews the work that the authors have done at the University of Wisconsin, Madison. Concretely , the paper addresses three areas of OLAP. First, how to efficiently compute a cube from the underlying relational database, by implementing multidimensional aggregation. Second, they review algorithms to estimate the storage that would be needed to precompute a subset of a cube. Such algorithms are useful since precomputing parts of the cube is a standard technique to improve response time, while at the same time precomputation can demand a lot of storage. Finally, they look at the tradeoffs of using multidimensional structures (MOLAP) versus tables (ROLAP) for storing multidimensional data.

The second paper by Venky Harinarayan addresses the issue of query performance in OLAP applications. Identifying the mismatch between the data cube model and the underlying relational interface and bridging this mismatch becomes the focus of the paper. The paper also discusses approximate querying on OLAP.

The third paper by Curtis Dyreson, introduces the notion of *Incomplete Data Cubes* in which regions of the cube are missing. (For instance, at some point is decided that keeping hourly sales figures is not necessary, and daily reports will suffice.) The author shows how incomplete cubes can be used as sieves of relevant summary data from an unstructured text file.

The last two papers deal with indexing methods for OLAP databases. Sunita Sarawagi presents a survey of existing indexing methods and discusses their advantages and disadvantages. The paper also proposes extensions to multidimensional indexing methods that can make the indexes more appropriate for OLAP. The paper by Ted Johnson and Dennis Shasha presents a novel data structure called *cube forests* that exploits the hierarchical nature of the OLAP data to save space.

OLAP is a fruitful area of database research with applications and needs that can be readily found in the real world. The articles in this issue show some of the most important issues that drive research in OLAP nowadays.

Daniel Barbará
Bell Communications Research

Cubing Algorithms, Storage Estimation, and Storage and Processing Alternatives for OLAP *

Prasad M. Deshpande Jeffrey F. Naughton Karthikeyan Ramasamy Amit Shukla
Kristin Tufte Yihong Zhao

Computer Sciences Department
University of Wisconsin-Madison

Abstract

"OLAP" or multi-dimensional analysis workloads present a number of interesting challenges and opportunities for database developers and researchers. While the OLAP goal of extremely fast response times is hard to meet in general, the structure of the underlying multidimensional model (whether implemented by arrays or by tables) provides a framework that can be used to approach this performance goal for this class of queries. In this note we give an overview of our research into these problems.

1 Introduction

The OLAP (On-Line Analytical Processing) phenomenon is interesting from both an academic and an industrial point of view. While historically there have been companies selling systems that would be today classified as "OLAP" systems for quite awhile, until recently this area was largely ignored by the database research community. Currently the research community is apparently making up for lost time, as we are in the midst of a tremendous burst of OLAP-related research activity. This note gives an overview of some of the research we are pursuing at the University of Wisconsin-Madison Computer Science Department.

For readers who are not familiar with "OLAP" or multi-dimensional data analysis, perhaps the best place to start is the Web, which is full of vendor white-papers describing each company's take on what these terms mean. A good jumping-off point is the OLAP council Web site — http://www.olapcouncil.org/. In this short note we will try to give enough background to make the note self-contained, but we are of necessity omitting large portions of the OLAP universe. Also, due to space limitations, we are omitting a full discussion of related work; we encourage the reader to get the full papers from the URL's listed in this paper to find references to related work.

One key demand of OLAP applications is that queries be answered quickly. Of course, this is a rather common goal; few types of systems have as their goal "slow response times." However, the OLAP marketplace has perhaps taken this to a new level, demanding that decision support queries be answered in seconds. In general this is hard to do; fortunately, the multi-dimensional data model embedded in OLAP systems is structured enough

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*This work supported by ARPA, NASA, NSF, IBM, and NCR.

to allow systems to approach this goal. The overall goal of our research is just that: to exploit the structure of the multi-dimensional model to provide extremely high performance for this class of queries.

At the heart of OLAP or multidimensional data analysis applications is the ability to simultaneously aggregate across many sets of dimensions (in SQL terms, this translates to many simultaneous group-by's.) Computing these multidimensional aggregates can be a performance bottleneck for these applications, so we have explored various schemes for implementing multidimensional aggregation. The SQL community has also taken notice of the requirement for simultaneous multidimensional aggregation; in particular, there are proposals for extending SQL to include the "cube" operator as proposed by Gray et al. [GBLP96]. This operator computes aggregates over all subsets of dimensions specified in the "cube" operation, and is equivalent to the union of a number of standard group-by operations. Initially, we focussed on efficient algorithms for computing the cube based upon the standard relational techniques of sorting and hashing as applied to data stored in tables. Section 2 gives an overview of this work. A team at IBM Almaden including Rakesh Agrawal, Ashish Gupta, and Sunita Sarawagi independently explored similar approaches to this problem; for more details please consult our joint paper in VLDB96, available as [AAD+96] or from
http://www.cs.wisc.edu/p̃md/papers/vldb96/Article.PS.

To speed up multidimensional data analysis, database systems frequently precompute aggregates on some subsets of dimensions and their corresponding hierarchies. This is related to the "computing the cube" problem in that a common tactic is to precompute some or all of the cube. This improves query response time. However, the decision of what and how much to precompute is difficult. It is further complicated by the fact that precomputation in the presence of hierarchies can result in an unintuitively large increase in the amount of storage required by the database. Hence, it is interesting and useful to estimate the storage blowup that will result from a proposed set of precomputations without actually computing them. In Section 3 we describe three strategies we have evaluated for this problem: one based on sampling, one based on mathematical approximation, and one based on probabilistic counting. Full details on this work are available in either [SDNR96] or from http://www.cs.wisc.edu/s̃amit/vldb96.ps.

Finally, it is clear from a look at the OLAP industry today that array-based storage structures are an important alternative to tables for storing multidimensional data. This has manifested in the heated "MOLAP/ROLAP" marketing wars. In our research our goal is to attempt to identify the tradeoffs between the two approaches, to quantify differences between the two approaches, and to see if object-relational technology can be employed to get the "best of both worlds." Toward this end we have implemented a multidimensional array ADT in our Paradise Object-Relational DBMS. Section 4 describes our ADT and discusses some early interesting results from experiments on the implementation and our directions for future research. A paper describing some of this research can be found in [ZDN97] or at http://www.cs.wisc.edu/zhao/cube2.ps.

2 Cubing Relational Tables

The group-by operator in SQL is typically used to compute aggregates on a set of attributes. For business data analysis, it is often necessary to aggregate data across many dimensions (attributes). For example, in a retail application, one might have a relation with attributes *(Product, Year, Customer, Sales)*. An analyst could then query the data by asking the system to display:

- *sum of sales by (product, customer)*:

  For each product, give a breakdown on how much of it was sold to each customer.

- *sum of sales by (date, customer)*:

  For each date, give a breakdown of sales by customer.

- *sum of sales by (product, date)*:

For each product, give a breakdown of sales by date.

Gray et al. [GBLP96] proposed the "cube" operator to formalize this sort of simultaneous aggregation and to express it in SQL.

The "cube" operator is the n-dimensional generalization of the group-by operator. The cube operator on n dimensions is equivalent to a collection of group-by statements, one for each subset of the n dimensions:

```
CUBE [DISTINCT | ALL] <select list> BY <aggregate list> <table expression>
```

The semantics of the cube operator is that it computes aggregates (specified in the `<aggregate list>`) by grouping on all possible subsets of the attributes in `<select list>`. Each such group-by aggregate is called a **cuboid**. The group-by aggregate over all the attributes in `<select list>` is called the **base cuboid**. Thus for a cube over n attributes, 2^n cuboids including the base cuboid would have to be computed.

Returning to our retail example, consider the relation *(Product, Year, Customer, Sales)*. The following statement will compute the sales aggregate cuboids on all 8 subsets of the set {Product, Year, Customer} (including the empty subset):

```
CUBE Product, Year, Customer By SUM(sales)
```

The cube operation on n attributes consists of computing a collection of 2^n cuboids, and a key challenge is to understand how the cuboids in this collection are related to each other, and to exploit these relationships to minimize I/O. Our approach explores the relationships between the cuboids using a hierarchical structure, and formalizes the problem of computing a CUBE in terms of this hierarchy. This provides a framework in which algorithms for computing the CUBE can be understood and evaluated.

In our work we explored a class of sorting-based methods that try to minimize the number of disk accesses by overlapping the computation of the various cuboids, thereby reducing the number of sorting steps required. Our experiments with our implementation of these methods show that they perform well even with limited amounts of memory. In particular, they always perform substantially better than the method of computing the CUBE by a sequence of group-by statements, which is the only option in many commercial relational database systems.

For more detail on this work please consult either [AAD+96] or
http://www.cs.wisc.edu/ pmd/papers/vldb96/Article.PS.

3 Storage Explosion and the Cube

As noted in the introduction, multidimensional data analysis, as supported by OLAP systems, requires the computation of several large aggregate functions over large amounts of data. To meet the performance demands imposed by these applications, virtually all OLAP products resort to some degree of precomputation of these aggregates. The more that is precomputed, the faster queries can be answered; however, it is often difficult to say *a priori* how much storage a given amount of precomputation will require. This leaves the database administrator with a difficult problem: how does one predict the amount of storage a specified set of precomputations will require without actually performing the precomputation? (Harinarayanan et al. [HRU96] considered the related problem of which aggregates are best to precompute and store if only a given number of them can be stored.)

To further clarify the problem we are considering, we begin with an example[1]. Consider a table of sales with the schema

```
Sales(ProductId, StoreId, Quantity)
```

[1] This example is from [AGS95]. We gratefully use it here because we have always wanted to write a paper incorporating the term "personal hygiene."

with the intuitive meaning that each tuple represents some quantity of some product sold in some store. Furthermore, assume that we have some information about products captured in a table

```
Products(ProductId, Type, Category)
```

capturing for each product to which Type it belongs, and for each Type to which Category it belongs. Finally, suppose we have an additional table

```
Stores(StoreId, Region)
```

which captures for each store to which region it belongs. This data set can be viewed conceptually as a two-dimensional array with hierarchies on the dimensions.

There are a number of queries that can be asked of this data. For example, one may wish to know sales by product; or sales by type; or sales by product and region; or sales by store and type; and so forth. Each of these queries represents an aggregate computation. For example, sales by product in SQL is just:

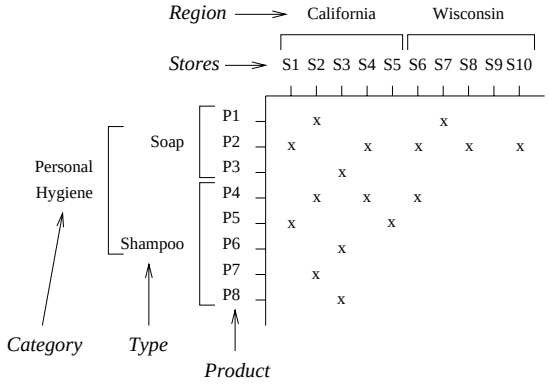
```
select ProductId, SUM(Quantity)
from sales
group by ProductId;
```

If the sales table is large, this query will be slow. However, if this aggregate is precomputed, the query (and queries derived from it) can be answered almost instantly. Therefore, the task the DBA faces is to choose a set of queries to precompute and store. In this paper, we first consider the problem of estimating how much storage will be required if all possible combinations of dimensions and their hierarchies are precomputed. Furthermore, once we have described how to estimate this full precomputation the extension to precomputation of a subset is trivial.

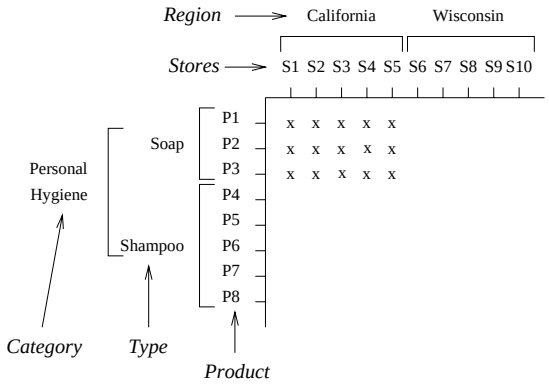
A useful way to describe the full precomputation problem is to use the cube framework discussed in the previous section. In our example, the cube consists of the group-by's: (), (ProductId), (StoreId), (ProductId, StoreId). The SQL for these four group bys (in the above order) is:

```
select SUM(Quantity)
from sales;

select ProductId, SUM(Quantity)
from sales
group by ProductId;

select StoreId, SUM(Quantity)
from sales
group by StoreId;

select ProductId, StoreId, SUM(Quantity)
from sales
group by ProductId, StoreId;
```

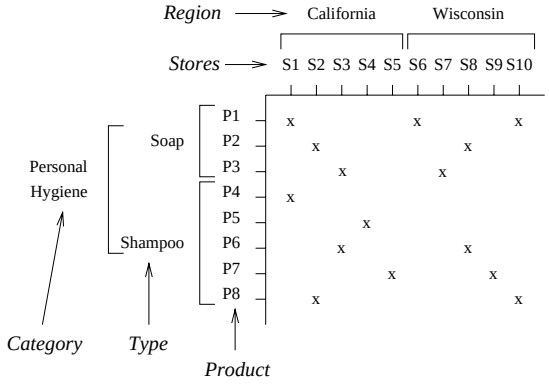
When we consider the possibility of aggregating over hierarchies, we get a generalization of the cube, which we will refer to as the cube from here on. Again returning to our example, the cube with hierarchies will compute aggregates for (), (ProductId), (StoreId), (Type), (Category), (Region), (ProductId, StoreId), (ProductId, Region), (Type, StoreId), (Type, Region), (Category, StoreId), and finally (Category, Region). It is the presence of hierarchies in the dimensions that in general make the storage requirements of cubes with hierarchies far worse than



(a)



(b) Database DB 1



(c) Database DB 2

Figure 1: Three sample multi-dimensional data sets. S_i's represent Stores and P_i's represent Products. Stores S1 - S5 are in California, and so roll up into the region California, while S6 - S10 are in Wisconsin, and roll up into the region Wisconsin. Products P1 - P3 are of type Soap, while products P4 - P8 are of type Shampoo. Soap and Shampoo are further grouped into the category Personal Hygiene. The **x**'s are sales volumes; entries that are blank correspond to (product, store) combinations for which there are no sales. (b) and (c) are sample multi-dimensional data sets which are used in an example.

Table 1: The variation in the size of the cube with the data distribution. Figures 1 (b) and (c) show DB 1 and DB 2 respectively.

Group-by	DB 1	DB 2
()	1	1
(Products)	3	8
(Type)	1	2
(Category)	1	1
(Stores)	5	10
(Regions)	1	2
(Products, Stores)	15	15
(Type, Stores)	5	15
(Category, Stores)	5	10
(Product, Region)	3	14
(Type, Region)	1	4
(Category, Region)	1	2
Size of Cube	42	84

that of cubes without hierarchies. Note that on this small example of only two dimensions the cube computed on the 16 tuples in Figure 1 (a) results in 73 tuples, while the cube without hierarchies has 34 tuples.

Furthermore, for a given database schema and a fixed number of data elements, the resulting size blowup on computing a cube can vary dramatically. Figure 1 (b) and (c) show two databases which illustrate the range of blowups that can occur. Each database has the same number of tuples (15), the same number of dimensions (2), and the same hierarchy on the dimensions. As the computation in Table 1 shows, even for a small database, and a small number of dimensions, the sizes of the cubes for the databases are very different.

We now turn to the problem of estimating the size of these blowups without computing the cube. We have considered three solutions.

For the first solution, if the data is assumed to be uniformly distributed, we can mathematically approximate the number of tuples that will appear in the result of the cube computation using the following standard result. Feller [Fel57]:

> If r elements are chosen uniformly and at random from a set of n elements, the expected number of distinct elements obtained is $n - n(1 - 1/n)^r$.

This can be used to quickly find the upper bound on the size of the cube as follows.

To apply the uniform-assumption method, we need to know the number of distinct values for each attribute of the relation. Such statistics are typically maintained in the system catalog. Using the above result, we can estimate the size of a group by on any subset of attributes. For example, consider a relation R having attributes A, B, C, D. Suppose we want to estimate the size of the group by on attributes A and B. If the number of distinct values of A is n_1 and that of B is n_2, then the number of elements in $A \times B$ is $n_1 * n_2$. Thus $n = n_1 * n_2$ in the above formula. Let r be the number of tuples in the relation. Using these values we can estimate the size of the group by. This is similar to what is done in relational group by estimation.

A cube is a collection of group bys on different subsets of attributes. If we are computing a cube on k dimen-

sions where dimension i has a hierarchy of size h_i then the total number of group bys to be computed is:

$$\prod_{i=1}^{k}(h_i + 1) \tag{1}$$

This figure is obtained by observing that in any group by atmost one of the attributes in each hierarchy should be present. We can estimate the size of each of the group bys and add them up to give the estimated size of the cube.

Any skew in the data tends to reduce the size of the group bys reducing the size of the cube. Hence the uniform assumption tends to overestimate the size of the cube, and there is of course no way to know how far off it might be, since this method does not consult the database other than to gather crude cardinalities. It also requires counts of distinct values, without which it cannot be used. However, this method has the advantage that it is simple and fast.

For the second solution we considered a simple sampling-based algorithm. The basic idea is as follows: take a random subset of the database, and compute the cube on that subset. Then scale up this estimate by the ratio of the data size to the sample size. To be more precise, we have the following. Let D and s be the database and a sample obtained from the database respectively. If $|s|$ is the sample size, $|D|$ the size of the database, and *CUBE(s)* is the size of the cube computed on the sample s, then the size of the cube on the entire database D is approximated by:

$$\textit{CUBE(s)} * \frac{|D|}{|s|}$$

This is admittedly very crude. The approach of estimating the size of an operation by computing the operation on a subset of the data and then linearly scaling produces an unbiased estimator for some common relational algebraic operations such as join and select. Unfortunately, in this case, the estimate produced is biased, as estimating the size of the cube is more akin to estimating the size of a projection than it is to estimating the size of a join. However, once again the computation is simple, and has the potential advantage over the uniform assumption estimate of examining a statistical subset of the database (instead of just using cardinalities.) This simple biased estimator produces surprisingly good estimates.

The key idea of the third solution is based on an interesting observation made from Figure 1 (a). To compute the number of tuples formed by grouping Product type by Stores, we essentially group tuples along the Product dimension (to generate Product type), and count the number of distinct stores which are generated by this operation. Hence, by estimating the number of distinct elements in a particular grouping of the data, we can estimate the number of tuples in that grouping. We use this idea to construct an algorithm that estimates the size of the cube based on the following probabilistic algorithm which counts the number of distinct elements in a multi-set.

Flajolet and Martin [FM85] propose a probabilistic algorithm that counts the number of distinct elements in a multi-set. It makes the estimate after a single pass through the database, and using only a fixed amount of additional storage. This algorithm can be used as a basis upon which to build a cube size estimation procedure.

Comparing the algorithms based on their accuracy, we found that the algorithm based on sampling overestimates the size of the cube, and the estimate is strongly dependent on the number of duplicates present in the database. The algorithm based on assuming the data is uniformly distributed works very well if the data is uniformly distributed (what a surprise!), but as the skew in the data increases, the estimate becomes inaccurate. In the experiments we carried out, the analytical estimate was more accurate than the sampling based estimate for widely varying skew in the data. The algorithm based on probabilistic counting performs very well under various degrees of skew, always giving an estimate with a bounded error. Hence it provides a more reliable, accurate and predictable estimate than the other algorithms.

Which algorithm is best depends upon the desired accuracy, the amount of time available for the estimation, and the degree of skew in the underlying data. But in most cases, the algorithm of choice for a reasonably quick

and accurate estimate of the size of the cube is the algorithm based on probabilistic counting. For more details on this work please consult either [SDNR96] or from http://www.cs.wisc.edu/šamit/vldb96.ps.

4 An Object-Relational ADT Alternative to Tables

An interesting question is what storage structure should be used to hold multidimensional data for OLAP queries. One logical option is relational tables. Suppose we wish to store a 3-D array of integers in a table. We can do so by declaring a 4 attribute table, say $t(I, J, K, D)$, where the triple (I, J, K) is the index of the array cell holding the integer D. Another option is to store the data in an array, perhaps laying out the data in row-major or column-major order, just like a programming language array.

Both of these approaches have advantages and disadvantages, and this has given rise to a vigorous ongoing debate between the MOLAP and ROLAP vendors. Some of the advantages of a MOLAP approach include:

- Dense arrays (ones with a larger fraction of their cells filled with valid data) are stored more compactly in the array format than in tables (because the array indices are implicit rather than explicit).
- Array lookups become simple arithmetic operations rather than associative lookups in tables.

Some advantages of the ROLAP approach include

- Sparse data sets may be stored more compactly in tables than in arrays (since only valid cells are stored.)
- With tables you get all of what a standard SQL database brings, including scalability to very large data sets.

The reality is of course far more complex than these simplistic facts indicate. For example, sparse arrays can be stored compactly; tables can be indexed, e.g. by bitmaps, to provide fast operations on them. Also, there are other issues we have not even touched upon here.

In our research we are investigating these tradeoffs in more detail, and to quantify the benefits of each. We are fortunate in that we have available our object-relational DBMS, Paradise, for experimentation. We have used Paradise to implement an array ADT as an example of a MOLAP-style storage system, and also to implement bit-map indices and special purpose query evaluation algorithms as an example of a high-performance ROLAP system. This allows us to greatly reduce the number of factors we vary in the experiments, since both systems run in the same code base. For example, both subsytems have the same concurrency control and recovery subsystems.

We are currently experimenting with the system and a paper is under preparation. As an early example of a problem we have investigated, consider the problem of computing the "cube" over data stored in arrays rather than in tables. We have found that for a wide variety of data sets this is surprisingly efficient; in fact, it is so efficient that in many cases it is faster to (a) start with a data set in a table, convert it to an array, "cube" the array, and store the result back to tables, than to (b) cube the table directly. For more details on this work please consult [ZDN97] or http://www.cs.wisc.edu/žhao/cube2.ps.

References

[AAD+96] S. Agarwal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, S. Sarawagi. On the Computation of Multidimensional Aggregates. *Proc. of the 22nd Int. VLDB Conf.*, 1996.

[AGS95] R. Agrawal, A. Gupta, S. Sarawagi. Modeling Multidimensional Databases. *IBM Research Report*, IBM Almaden Research Center, San Jose, California, 1995.

[Fel57] W. Feller. *An Introduction to Probability Theory and Its Applications*, Vol. I, John Wiley and Sons, pp 241, 1957.

[FM85] P. Flajolet, G.N. Martin. Probabilistic Counting Algorithms for Database Applications, *Journal of Computer and System Sciences*, 31(2): 182-209, 1985.

[GBLP96] J. Gray, A. Bosworth, A. Layman, H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals, *Proc. of the 12th Int. Conf. on Data Engg.*, pp 152-159, 1996.

[HRU96] V. Harinarayanan, A. Rajaraman, J.D. Ullman. Implementing Data Cubes Efficiently, *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 205–227, 1996.

[SDNR96] Amit Shukla, Prasad M. Deshpande, Jeffrey F. Naughton, and Karthikeyan Ramasamy, "Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies." In *Proceedings of the 22nd Conference on Very Large Databases, Mumbai (Bombay)*, 1996.

[ZDN97] Yihong Zhao, Prasad M. Deshpande, and Jeffrey F. Naughton, "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates." In *Proceedings of the 1997 SIGMOD Conference*, Tucson, Arizona, May 1997.

Issues in Interactive Aggregation

Venky Harinarayan
Junglee Corporation
1250 Oakmead Parkway, Suite 310
Sunnyvale, CA 94086
venky@junglee.com

Abstract

OLAP applications make heavy use of aggregation and yet need to be interactive. Query performance is thus crucial for OLAP applications. In this paper, we identify the impedance mismatch between the "data cube" data model exported by OLAP engines and the underlying relational data-storage interface as being key to understanding OLAP performance. Bridging this mismatch efficiently is essential and in this context we survey some of the recent results in this area. Further, as the volume of data and the number of data sources keeps increasing OLAP will have to evolve. In particular, we discuss approximate querying and the effect of the Internet on OLAP.

1 Introduction

The increasing computerization of operations within enterprises has led to the availability of large amounts of valuable corporate data. The explosive emergence of the Internet as a commercial medium also has added to the volume of data confronting decision makers. Harvesting all this data to extract useful information is becoming critical at all levels of decision making – from the enterprise to the consumer. For example, analysts in a large retail company may determine which products to discount based on customers' buying preferences. These buying preferences may be extracted from a table that has the monthly history of individual transactions from the different retail stores. As another example, a consumer desiring to buy a car may use information gleaned from multiple data sources (web sites) on the Internet in deciding on the car and the dealer.

There are some common threads in these differing examples. First, user queries usually span multiple data sources. Further, since the decision-making process is interactive, responses to user queries must take of the order of a few seconds to execute. In many practical cases, having the query access the data sources at run time (the mediator approach [Wie92]), results in unacceptable response times. An increasingly popular solution is to collect, reconcile, and store all the required data in one unified data repository commonly referred to as a "data warehouse." It is important that decision-makers be able to navigate this data to analyze and extract information. Enabling such navigation is the goal of Online Analytic Processing (OLAP) systems. A preferred way of navigation is to view aggregate properties of the underlying data at different granularities of aggregation. Using aggregation to analyze data becomes especially important as the volume of the underlying data becomes large.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

However at the same time, the cost of aggregation increases with the volume of the data, since it can potentially involve accessing every data item. This fact coupled with user requirements for interactivity make efficient processing of aggregate queries central to the success of OLAP — enabling interactive aggregation is in some sense the goal of OLAP systems. In this paper, we survey some of the recent advances in making aggregation truly interactive.

This short paper has two sections. First, in Section 2, we examine the current state of the art in high-performance OLAP query processing. We identify the impedance mismatch between the "data cube" data model used by OLAP engines and the relational data model used by the underlying data store. Bridging this mismatch efficiently is the key to OLAP performance and it is in this context that we survey some of the recent results in this field. Next, in Section 3, we discuss some of the future directions for OLAP. In particular, we examine the impact of the Internet in making OLAP more mainstream and also the use of approximate querying in OLAP engines to handle ever-increasing volumes of data without sacrificing interactivity.

2 OLAP in the Enterprise

2.1 OLAP Architecture — The Impedance Mismatch

Enterprises are standardizing on relational database systems for storage of corporate data. Thus, enterprises typically use commercial relational database systems to house their data warehouses. As mentioned before, it is important for OLAP users to be able to navigate and visualize the underlying data for their analysis. However, presenting the data to the user in the form of many "flat," 2-D relational tables does not help much in visualizing the data. If a user is to extract information buried in the data, it is important that the data be presented to the user in an intuitive way. For many OLAP applications, it is much more intuitive to present the data to the user as a multidimensional "cube."

Example 1: Consider the example of a retail chain that sells products in various countries and is interested in their sales over time. This data is typically kept in tables in a relational database. Now, consider an analyst who has noticed that total sales in the year 1995 were higher than normal and wants to investigate why. He may wish to breakdown the sales by country to determine the countries that accounted for this increase. If the interface presented to him is one of relations, this simple operation is made unnecessarily complex by having him think of joins, selections, and so on. It is much more natural to think of the data as sales numbers populating a three-dimensional cube, whose dimensions are product, country and time. Breaking down sales by country corresponds to projecting the cube onto the country dimension. For most of the ad-hoc analysis done in OLAP, a cube is the natural interface. The relational model is much too cumbersome an interface.

This multidimensional-spreadsheet (data-cube) perspective of data is referred to as *dimensional modeling*. The data-cube data model is more restrictive than the relational data model. However it is an intuitive and natural data model for OLAP applications.

There is thus an "impedance mismatch" between the interface the users would like — the data cube, and SQL — the interface provided by the relational database that stores the data. This impedance mismatch is usually bridged by a layer of software known as the *OLAP engine*. In particular, the OLAP engine has to handle the following two shortcomings of the relational engine.

- Functionality. The relational engine may not provide the functionality required by the OLAP engine.
- Performance. The relational engine may not efficiently execute common OLAP operations.

Efficiently managing and narrowing this impedance mismatch is at the heart of improving OLAP query performance. Managing this mismatch efficiently is the charter of the OLAP engine, while narrowing this mismatch is the charter of the relational engine.

2.2 OLAP Performance

In the ensuing discussion we identify techniques to improve query performance in the OLAP engine and in the relational engine.

2.2.1 The OLAP Engine

Arguably, the best method to manage a mismatch is to avoid encountering it. For the OLAP engine, this method translates to having as much of the data available natively to the data cube interface. In effect, this means precomputing the aggregates corresponding to data cubes or parts thereof. In fact, precomputing aggregates is the single most effective tool to improve query performance [Kim96]. Picking the right set of aggregates to precompute is critical to the success of this strategy and there is a fundamental trade-off between performance and scalability: the more aggregates an OLAP engine precomputes, the better the query performance but the less the scalability with larger data cubes. Different OLAP engines choose differing points on this trade-off curve, sacrificing opportunities either because they cannot scale or because they do not have adequate performance. Arbor's Essbase system, for example, precomputes the entire data cube and stores it in its own proprietary storage system. Once the precomputation is done, there is little interaction between the OLAP engine and relational engine. Informix's MetaCube, on the other hand, only precomputes parts of the data cube and stores it in the relational engine itself. This approach while not as high-performance as Arbor's is more scalable. In [HRU96], we investigate this problem in detail and provide simple yet scalable greedy algorithms that select the set of aggregates to precompute based on the available resources. We also show that these algorithms perform provably close to optimal.

When the OLAP engine precomputes aggregates, it also has to address two important issues:

- Aggregate Navigation. It is important to make these precomputed aggregates transparent to the user of the OLAP engine. This transparency is required to make query specifications independent of the aggregates precomputed: the user of the OLAP engine should pose queries on the base tables not the precomputed aggregates. The OLAP engine should determine if the precomputed aggregates can be used to answer the query. The component of the OLAP engine that determines if the precomputed aggregates can be used to answer a given query is called the "aggregate navigator" [Kim96]. In [GHQ95], we give a general algorithm that determines when a given aggregate view can be used to answer a query. Recently, Dar et al. [DJLS96] have also provided solutions to this problem.

- Maintenance. Even though OLAP is a mostly read-only process, all the aggregates have to be recomputed when a new batch of data is loaded. Recomputing aggregates for every batch load may result in unacceptable downtimes and overhead. Quass in [Qua96] investigates and provides solutions to incrementally maintain aggregate views.

2.2.2 The Relational Engine

In terms of API-level functionality, relational engines can offer much to narrow the mismatch. By reducing the gap between SQL and the data-cube interface we move more of the processing closer to the data, reducing the transfer of data between the relational database and the OLAP engine, thus improving performance.

Since the data cube is a generalization of a spreadsheet, users expect the numerous aggregate functions popular in spreadsheets. SQL, however, has only five aggregation functions: **sum**, **count**, **avg**, **max**, and **min**. If the user specifies an aggregate function that is not expressible using these functions, the aggregate computation must be done by the OLAP engine. It is much more efficient to do this computation in the server itself. Some database systems like Red Brick, already provide an augmented SQL interface that has aggregate functions like the rank, *etc.* [Sys94]. Another extension to SQL — the data-cube operator — is proposed by Gray et al. in their seminal paper [GBLP94]. The data-cube operator enables OLAP engines to construct an entire data cube with one

query rather than having to invoke the relational engine with many groupby queries, thus improving performance. Agrawal et al. in [AAD+96] provide fast algorithms to compute the data cube in a relational engine.

In terms of performance requirements too there is a significant impedance mismatch. Most relational engines have been built with OLTP (Online Transaction Processing) applications in mind — an environment of many, simple, concurrent, read-write queries. Relational Engines frequently have poor performance for queries common in an OLAP environment.

Since OLAP is a mostly read-only query environment, complex index structures can be built to improve performance. Such indexes would be infeasible in an OLTP environment since they would incur too much overhead during updates. Indeed, index structures such as bit-mapped indexes [OG95], cube forests [JS95],*etc.*. have been suggested. Bit-mapped indexes are now available in many commercial relational systems. It should be noted here that there is a significant cost to adding a new index type, since it adds a new access method for the query optimizer to consider — substantial work on the query optimizer may be required to generate good plans.

Complex-query processing is another area in which most relational engines do not provide the performance required by OLAP applications. In OLTP systems query optimization is less important than efficient concurrency control and good resource allocation mechanisms. In OLAP systems, with complex queries and huge databases, a bad query plan can cause the query execution time to increase by orders of magnitude. Query optimization is thus extremely important. Unfortunately, while most OLAP queries are aggregate queries, aggregates are second-class citizens in commercial query optimizers: the number of plans considered is severely limited in the presence of aggregates.

Optimization of complex queries has received much attention in the research community and some of these techniques are indeed being implemented in commercial relational engines. Dayal [Day87], Chaudhuri and Shim [CS94], Yan and Larson [YL95], and Gupta et al. [GHQ95] give rules to move aggregates around in a query tree. A query optimizer can use these aggregate-reordering rules to pick the best query plan in a cost-based manner. Rather than move the aggregations around, Levy et al. [LMS94] have examined the dual problem of moving the selections around in a complex-query tree. They pull all selections to the top of the query tree and combine them to infer potentially stronger selection conditions on the the individual branches. Another technique called "magic sets" [Ull89] attempts to discard the irrelevant tuples early in the query processing. Magic sets can be used to good effect in complex queries, since the cost of determining the irrelevant tuples is more than offset by the benefit of reducing the size of intermediate relations. Mumick et al. first showed that magic sets could be used for query optimization in nonrecursive queries [MFPR90]. Seshadri et al. [SHR+96] recently show how to use the magic-sets rewriting in the context of a new relational operator they call "filter join" and incorporate the magic-set rewriting process into conventional cost-based optimizers.

We believe that relational engines will continue to extend their OLAP functionality using plug-in modules like Informix's datablades for example, to provide more of the required interfaces and functions. Thus, in the future, it is quite possible that relational engines will completely support the OLAP interfaces thereby obviating the need for OLAP engines.

3 OLAP: the Future

3.1 Approximate Queries

OLAP queries are typically aggregate queries that ask for statistical properties of the data. It is only appropriate that the results of an aggregate query not be exact but be statistically characterized. Consider the following example.

Example 2: An analyst in a retail chain wishes to know the average sales per day of widgets and issues this query over the sales-history table. Let us assume that five widgets, on average, were sold each day. In most scenarios,

an analyst is looking for trends not exact values, so a result of the form: the average number of widgets sold per day is between 4.999 and 5.001, with a confidence of 95%, is sufficient.

The advantage of answering a query approximately is that it may be answered very quickly. For example, sampling techniques can be used to reduce the size of the data sets. The biggest reasons given for not adopting such "statistical querying" in OLAP are:

1. Cultural. In many cases, mostly for psychological reasons, users want exact answers.

2. Software Engineering. It is not easy to develop and maintain a query processor which may return a different answer for the same query each time.

As the amount of data in data warehouses keeps increasing, techniques like statistical querying will have to be used in OLAP, if the querying process is to remain interactive. This need is already being recognized in industry: recently, Informix announced its MetaCube 3.0 product which has some of this capability. In another interesting direction, Hellerstein [Hel96] explores *online* aggregation, an extension to statistical querying. With online aggregation, the aggregation operators provide ongoing feedback, including statistical parameters like confidence. The aggregation operators are also controllable, so the user can terminate processing whenever desired. It should be noted however that such statistical OLAP engines increase the impedance mismatch with the underlying relational engine.

3.2 The Internet

The widespread adoption of Internet technology will profoundly affect OLAP. Most vendors of OLAP engines have thus far focused on Internet-enabling their offerings. Arbor's Essbase Web Gateway, Microstrategy's DSS-Web and Oracle's Express Web Agent are examples of solutions that enable access to OLAP engines through popular web browsers.

The true promise of the Internet however is in making OLAP a mainstream technology — moving OLAP from the domain of analysts to consumers. Already, one of the larger applications of the Internet is in decision-support — in providing job-seekers, computer-buyers, and such consumers with the information to make their decisions. The basic concepts of data warehousing and aggregation have thus naturally made their way onto the web. In fact, some of the most popular web sites on the Internet are basically data warehouses. Examples are search engines such as Alta Vista (www.alta-vista.com) and Lycos (www.lycos.com) which attempt to warehouse the entire web. Aggregation as a means to navigate and comprehend the vast amounts of data on the Internet, has also been recognized. Directory services such as Yahoo! (www.yahoo.com) and Excite (www.excite.com) attempt to aggregate the entire web into a category hierarchy and give users the ability to navigate this hierarchy.

However, inspite of the magnitude of this application and the accompanying growth of Internet commerce, the infrastructure present for decision support is extremely rudimentary and limited. Even popular web data warehouses like Alta Vista offer very limited capability for decision-support and this is often cited as a serious failing. The reason for this failing is that such data warehouses are unstructured and not subject-oriented. Hence complex queries like those needed for decision-support cannot be posed. There are significant technical challenges in building structured and subject-oriented warehouses that enable decision-support. Even if the underlying data source is structured, since the language of choice on the web, HTML, is unstructured, the underlying structure is lost. HTML thus needs to be enhanced to allow marking-up content structure. Another approach is to build sophisticated domain-specific extraction rules that derive structure. Also in many cases for a variety of reasons including issues such as copyright, volume of data, staleness and so on the data from some sources cannot be collected into a central data warehouse. For such sources, only the metadata can be stored and the data is presented at query time by accessing the source. How to answer queries in such a mixed mediator-warehouse environment is an interesting research problem.

The benefits of such subject-oriented, structured warehouses are immense. They will enable consumers to have comprehensive market knowledge. For example, job seekers will know how jobs are distributed by location, category and salaries. Home buyers will know how home prices in a given location vary by time of the year. As anecdotal evidence to the utility of OLAP on the web, consider the example of a warehouse that pertains to jobs that was built by Junglee (www.junglee.com) for a leading newspaper. The popularity of the "JobView" pages that gives the user a fixed "OLAP view" of the distribution of all jobs by location, company, and category, has been steadily increasing and now accounts for a significant fraction of all page views.

In summary, in today's society, as the amount of data being made available increases, information is becoming the biggest differentiator, and time the biggest resource. Decision-support techniques such as OLAP hold a key to improving efficiency in the society of the future.

Acknowledgements I'd like to thank Ashish Gupta for his comments on this paper.

References

[AAD+96] S. Agarwal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, 1996.

[CS94] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 354–366, 1994.

[Day87] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates and quantifiers. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, pages 197–208, 1987.

[DJLS96] S. Dar, H. V. Jagadish, A. Y. Levy, and D. Srivastava. Answering queries with aggregation using views. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, 1996.

[GBLP94] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. Microsoft Technical Report MSR-TR-95-22, 1994.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data-warehousing environments. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 358–369, 1995.

[Hel96] J. M. Hellerstein. The case for online aggregation. UC Berkeley Technical Report UCB//CSD-96-908, July 1996.

[HRU96] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proceedings of the ACM SIGMOD International Conference of Mangement of Data*, 1996.

[JS95] T. Johnson and D. Shasha. Hierarchically split cube forests for decision support: description and tuned design. Unpublished Memorandum, New York University, 1995.

[Kim96] R. Kimball. *The Data Warehouse Toolkit*. John Wiley and Sons, Inc., New York, NY, 1996.

[LMS94] A.Y. Levy, I.S. Mumick, and Y. Sagiv. Query optimization by predicate move-around. In *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994.

[MFPR90] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *Proceedings of the ACM SIGMOD International Conference of Mangement of Data*, pages 247–258, 1990.

[OG95] P. O'Neill and G. Graefe. Multi-table joins through bitmapped join indexes. In *SIGMOD Record*, pages 8–11, 1995.

[Qua96] D. Quass. Maintenance expressions for views with aggregation. In *Proceedings of the Views 96 Workshop*, 1996.

[SHR+96] P. Seshadri, J. M. Hellerstein, R. Ramakrishnan, T. Y. C. Leung, H. Pirahesh, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *Proceedings of the ACM SIGMOD International Conference of Mangement of Data*, 1996.

[Sys94] Red Brick Systems. *RISQL Reference Guide, Red Brick Warehouse VPT Version 3*. Part Number:401530, Los Gatos, CA, 1994.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II: The New Technologies*. Computer Science Press, Rockville, MD, 1989.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.

[YL95] W. P. Yan and P.-A. Larson. Eager aggregation and lazy aggregation. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 345–357, 1995.

Using an Incomplete Data Cube as a Summary Data Sieve

Curtis Dyreson
Department of Computer Science
James Cook University
Townsville, QLD 4811
AUSTRALIA
curtis@cs.jcu.edu.au
http://www.cs.jcu.edu.au/~curtis

Abstract

An incomplete data cube is a multidimensional hierarchy of aggregate values in which regions of the hierarchy, and the source data from which those regions are derived, are missing. We model an incomplete cube as a collection of complete sub-cubes called cubettes. Each cubette is defined using a precise, simple specification. The set of cubette specifications concisely characterises the information content of the cube. We discuss how these simple specifications can be used to generate a parser to sieve data from an underlying data source and populate an incomplete cube.

1 Introduction

Data cubes are a relatively recent, and popular phenomenon. A brief description of a data cube is that it is a multidimensional hierarchy of aggregate values. Values higher in the hierarchy are further aggregations of those lower in the hierarchy. The utility of the hierarchical organisation is that the user can easily navigate between high and low precision views of the same aggregate data. The hierarchical organisation supports *drill-down*, an operation that increases the precision of the aggregate data being viewed, and *roll-up*, which decreases that precision. For instance, suppose that a store manager is using a data cube to look at monthly sales for shoes and notices that sales in January were low. To analyse the poor sales the manager might drill-down to look at monthly sales by type of shoe or she might roll-up to look at sales for all product types combined. Several vendors already have cube products on the market, either as add-ons to existing databases or as stand-alone tools, and a "cube" operator has been proposed for inclusion in future SQL standards [3].

A cube can be implemented using a *lazy*, *eager*, or *semi-eager* strategy [6]. The eager strategy materialises every aggregate value in the cube hierarchy. The advantage of the eager strategy is that values can be quickly fetched from the cube during a query. The primary disadvantage is high storage cost (Shukla et al. present algorithms for estimating cube size [5]). For many applications eager cubes are just too big. A lazy implementation strategy does not materialise values. Instead, the values in the cube are computed from the underlying relations

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

during a query. The disadvantage of the lazy strategy is that it slows query evaluation (faster evaluation algorithms are being researched [1]). The semi-eager strategy eagerly materialises only some regions in the cube, and lazily computes others when needed during a query [4].

An *incomplete* data cube is also a multidimensional hierarchy of aggregate values. But in an incomplete data cube regions of the hierarchy, and the source data from which those regions are derived, are missing. For example, a data cube administrator may decide that hourly sales data from two years ago is no longer needed, daily sales data will suffice. The administrator can remove the aged, hourly data from the cube. The missing region makes the data cube incomplete and some queries (e.g., what are the hourly sales figures over the lifetime of the enterprise) can no longer be satisfied. Incomplete cubes have mechanisms for handling queries in the missing regions, such as suggesting alternative, complete queries and computing partial results.

In terms of storage, an incomplete cube has the same desirable behaviour as lazy and semi-eager cubes. Each materialises only part of what would be stored in an eager cube; the incomplete or unmaterialised portions incur no storage cost. For example, assume that a regional sales officer wants aggregate data for sales at stores in her region for every hour in 1995, but for stores in other regions, aggregate data for each day will suffice. In an eager cube an aggregate value for every combination of store and hour must be stored resulting in a much larger cube than needed. In contrast, an incomplete cube only stores the relatively small amount of data specified as needed, the hourly data for the other stores forms an incomplete region. Incomplete, lazy, and semi-eager cubes also scale well, new dimensions can be added to the cube and existing dimensions can increase in size (i.e., a more precise measure can be added to the dimension) with no adjustment to the existing cube storage. The resulting cube is merely incomplete in the new dimension, and can be populated as needed later.

But in one important respect an incomplete data cube is like an eager data cube, and unlike a lazy or semi-eager cube. Eager and incomplete cubes do not need the source data from which aggregate values in the cube are derived. Both lazy and semi-eager cubes presume that the source data is still available, so that an aggregate value which is not stored in the cube can be computed when needed. Both strategies *tightly couple* the cube to a data source. Eager and incomplete cubes, on the other hand, *uncouple* the cube from the source data.

In general, an incomplete cube is useful in situations where a complete, eager cube would be unnecessarily large, but where a lazy or semi-eager cube cannot be used because the source data is not available or expensive to query. We conjecture that an incomplete data cube would be useful in the following scenarios, among others.

- One reason that data cubes are popular is that many data collections are characterised by the property that as data in the collection ages, each datum individually becomes less relevant, but remains relevant in aggregate. For such data collections, a data cube can be used to store the aggregated historical data, allowing the original data to be archived or deleted and resulting in considerable savings in space.

- A data cube is used to summarise data from a log file or flat file. For example, suppose that a data cube is used to store aggregate data from a log file of sales transactions rather than a sales relation. To search a large log file and retrieve data during query evaluation imposes a heavy burden on system resources, so the data cube's administrator decides to use an incomplete data cube and package requests for more data in an overnight cron job.

- Aggregate data is broadcast on a network by various sites. The aggregate data from external sites is collected and inserted into a cube at each site, but the source data is not shipped across the network for a number of reasons (privacy, cost of broadcasting and duplicating the source data at each site, etc.).

- The cube contains regions of secret data and the authorisation to view the secret data varies from user to user, that is, some users can see all of the data, others only a portion, still others a different portion, etc. In an incomplete cube, it is easy to create a different, incomplete view of the same complete cube for each class of authorised user. The data can be kept secret by hiding it in an incomplete region.

This paper describes how an incomplete data cube can be used to sieve relevant summary data from an unstructured text file which is too large to store and query, such as a rapidly-growing log file. The next section briefly describes a real-world problem that could benefit from the application of incomplete data cube technology. We will use this example problem in the remainder of the paper. We then discuss an incomplete data cube in more detail. An incomplete cube is a federation of complete sub-cubes, which we call *cubettes*. Each cubette is defined using a precise, simple specification. The set of cubette specifications concisely characterises the information content of the cube. We then discuss how these simple specifications can be used to generate a parser to sieve data from an underlying data source and populate an incomplete cube.

More information on incomplete data cubes as well as a prototype implementation in the Java programming language is available at `http://www.cs.jcu.edu.au/~curtis/IncompleteCube.html`.

2 Motivating example - analysis of the World-Wide Web access log

A very rough description of the World-Wide Web (WWW) is that it is a network of information servers. A server responds to a client (browser) request by supplying the appropriate information, typically a page in a hypertext document. The authors of these documents usually want to know how their documents are being used. In particular, they would like to ascertain how frequently their pages are being requested and what kinds of users are requesting them. Of course there will likely be many different authors at a site and each will want a very detailed analysis of their materials. In addition, others at the site will want other kinds of summary information, for instance the server administrator might want to monitor the overall daily traffic from each country. So in general both detailed and abstract summaries must be collected and maintained.

The starting place for any quantitative analysis of WWW page usage is the server *access log*. The access log is a history of server requests. Each record in the log has a timestamp, the requesting machine IP name or number, and the resource requested. For example, in the following access log record:

```
crab.jcu.edu.au - - [01/Jan/1997:17:55:24] "GET /web/home.html HTTP/1.0" 200 4748
```

the timestamp is `[01/Jan/1997:17:55:24]`, the requesting machine is `crab.jcu.edu.au`, and the resource is `/web/home.html`. Although each record in the access log also has other information, such as the response code, without loss of generality, we ignore that information in this paper.

A single access log, however, will never contain the entire request history for pages on a server. Many clients will use a proxy cache server. A proxy cache server keeps copies of frequently requested pages in its cache. A page request for a cached page will be handled entirely by the proxy, and so will be logged in its access log. So the complete history of requests for a page is commonly distributed over many log files. In order to more accurately assess the use of WWW pages, the data in these distributed log files needs to be integrated.

Few, if any, of these log files are kept entirely in situ. Logically, the access log is an append-only file since server history only accumulates. The access log for the Department of Computer Science server at James Cook University grows by about a megabyte a day. Physically, some technique must be used to limit the file's growth, otherwise, at many sites, it will quickly exceed storage capacity. Typically, as the access log grows, it is periodically moved to a different storage medium, e.g., from disk to tape, or from an uncompressed to a compressed version, and the log is restarted.

Even if the entire access log is kept, access to the log file at many sites must be restricted to protect the privacy of both clients and authors. So it would be beneficial if the relevant access log data could be hidden or made available on an individual basis.

In summary, a data analysis tool is needed that can generate a parser to sieve data from a text file prior to the file being archived, allows the piecemeal specification and retention of relevant summaries, is able to integrate several data sources, and supports data hiding. While a complete data cube can satisfy some of these requirements, the resulting data cube will likely be too large. An incomplete data cube can satisfy all the requirements

in a minimal amount of space since only the data specifically requested is put into the cube (and just like in a complete cube, that data can still be compressed and compacted).

3 Measures and units

The three kinds of information in an access log record come from separate domains. The time is a value from the temporal domain, the source machine is from the domain of machine names (a spatial domain), and the file name is from the domain of file names (a second spatial domain). Each kind of information is in fact a unit of measurement, and given to an implied system of measurement. For example, the temporal information in an access log record is measured in *seconds*. Individual units in a measurement by seconds are `01/Jan/1997:17:55:24`, `01/Jan/1997:17:55:25`, etc. In the remainder of this paper we will refer to a system of measurement as a *measure*, e.g., seconds, and a unit of measurement as a *unit*, e.g., a particular second.

There can be many different measures in a single domain. For instance, in the temporal domain, a page request might be measured by the second, hour, day, week, month, or year in which it occurs. Most of these measures are related insofar as some are strictly more precise than others. For example, a temporal measurement given in days is more precise than one given in weeks since a measurement in days pinpoints the week but not vice-versa (since there are seven days in every week). In the next section we discuss how units, measures, and the relationships among them are used in an incomplete data cube.

4 Data cubes

In this section we describe an incomplete data cube in more detail.

A useful way of understanding a data cube is to conceive of it as a dataflow graph. An edge in the graph represents a data dependency; the data at the "to" node is derived from the data at the "from" node. In the graph there are two kinds of nodes: *source* nodes and *derived* nodes.

A source node represents a group of facts drawn from some underlying data source, usually a database relation. Facts are precisely measured in several dimensions and those with the same measurements are grouped at the same source node. The most common measurement dimensions are space and time. For example, facts could be grouped based on the temporal measure of days and the spatial measure of countries. In such a grouping there would be one source node for every combination of country and day; facts having the same country and day measurements would be placed in the same source node.

A derived node, on the other hand, holds the result of an aggregate operation applied to the data at the incoming nodes. A derived node on an edge from a source node holds the result of computing an aggregate, e.g., **count**, on the group of facts at the source node. Other derived nodes hold the result of an aggregate operation, e.g., **sum**, applied to the aggregate values on the incoming edges.

An example graph is shown in Figure 1. The graph assumes that there is only one domain of measurement: time. The lone source node depicted is the group `1 jan 1997`. Facts that share a temporal measurement of 1/Jan/1997 are in the group at this source node. Above the source node is a derived node, the unit `1 jan 1997`. The aggregate value at this node is a count of all the page requests for the first of January. An edge connects the source node to this node because the aggregate value is computed using the data at the source node. Above that derived node are other derived nodes, with the data dependencies indicated by the edges.

The dependencies in the graph are given by the relationships among the measures in each dimension. In general, units in less precise measures are dependent on those in a more precise measure. For example, the measure of months is less precise than that of days. The graph in Figure 1 shows that the unit `jan 1997` (which is in the measure of months) is dependent on the units `1 jan 1997` through `31 jan 1997`.

As an aside, we note that units, measures, and the relationships among them are specified in internal cube tables (we assume finite, bounded domains). Figure 2 presents a simplified view of the internal tables which

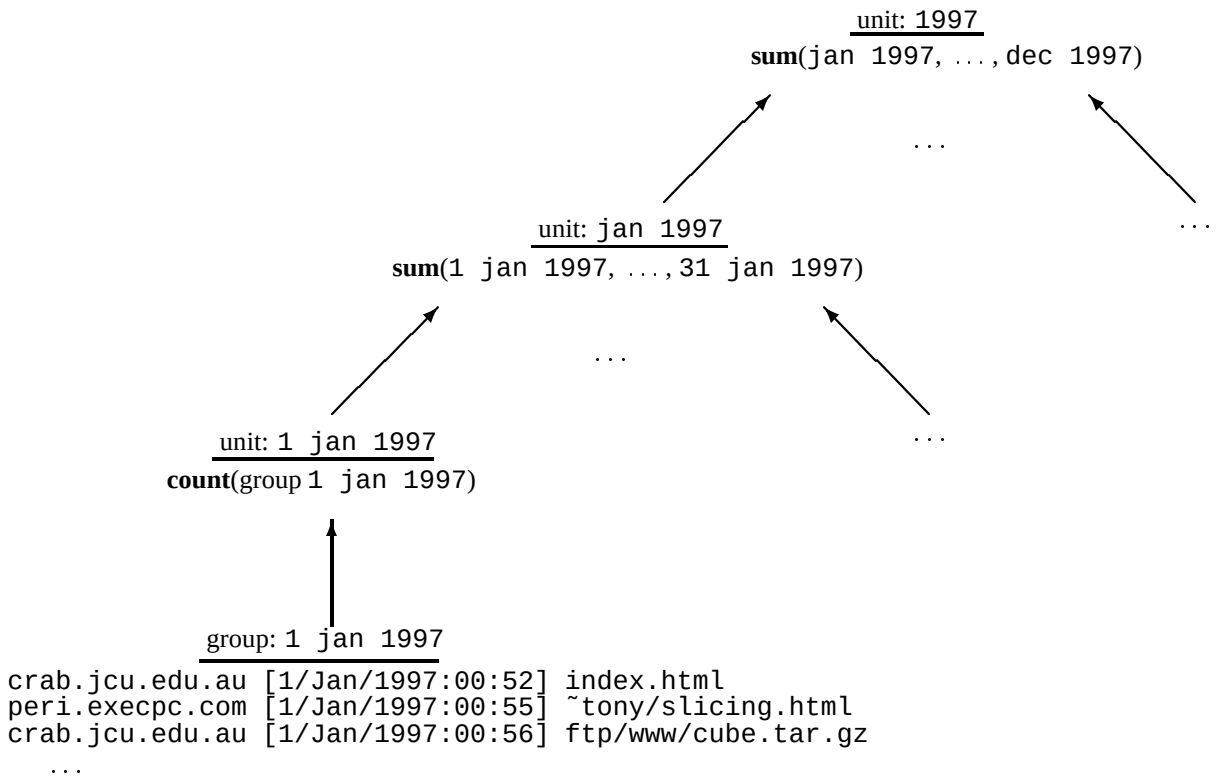


Figure 1: A graph depicting the hierarchy of dependencies in a data cube

describe the dependencies in the graph for Figure 1. The *Edges* table is the set of edges between nodes (the edge from a source node to a derived node is implicit). The *Groups* table is the set of groups. The membership pattern is a regular expression which is used to identify members of the group when scanning the access log. To reconfigure the cube for a text file with a different date format, the *Groups* table must be edited. Although the cube is a multidimensional space, the tables for each dimension are specified independently.

In an incomplete data cube the values of some nodes in the graph are *incomplete*. An incomplete value is not stored and can not be computed from the values that it depends on (at least one of them must also be incomplete, otherwise the value could be computed).

4.1 Cubettes

Integral to an incomplete data cube is a concise, high-level description of the complete regions in the cube (and by omission, which regions are incomplete). We will call a complete region a *cubette*, to signify that it is a diminutive, complete cube within the incomplete cube hierarchy. A cubette is concisely specified as a combination of a unit, u, and a measure m, and is written $u@m$ (literally u at m). The cubette specification describes a subgraph that extends from an apex at u to those units in the measure of m that are on a path from a source node to u. The units at measure m in the subgraph form the *base* of the cubette. For example, consider the cubette specification `jan 1997@days`. The base of the cubette is all the days that are on a path from a source node to `jan 1997`, that is, all the days in the month of January 1997. All the nodes in the cubette from the base (the days in January 1997) to the apex (`jan 1997`) are complete.

The core of an incomplete cube is the set of cubette specifications. The set represents the information content of the incomplete cube. We anticipate that the set will be moderately large, between a thousand and a hundred thousand cubettes. Each cubette is an independent, complete cube within the incomplete cube and can be implemented using a lazy, semi-eager, or eager strategy. Elsewhere we give algorithms for inserting new cubettes

(which is more involved than simply adding a new specification to the set) and deleting cubettes [2].

4.2 Queries

A data cube query, such as drill-down, is an operation that retrieves the value at a unit (or set of units). For instance, the user could query for the value of the unit `jan 1997`. This unit may be within a cubette and the query can be satisfied (elsewhere we give an algorithm for quickly determining whether the query can be satisfied [2]), or the unit may be in an incomplete region.

When the user queries for information in an incomplete region, the incomplete cube may have enough information to partially satisfy the query. For example, a query for the value of `jan 1997` can be partially satisfied since most of the values upon which it depends are complete (all save `1 jan 1997`). Various notions of query completeness can be defined and supported in an incomplete cube [2].

Queries in a cube will often fall into an incomplete region since users typically will not know a priori the extent of the complete information in a cube. However, a query in an incomplete region can be redirected to the "nearest" complete region. For example, a query for the incomplete value at `jan 1997` could be redirected to the complete value at `jan 1997` (or to one of the days in January). Elsewhere we give an algorithm for redirecting queries to complete regions[2].

5 Populating the cube

The cubette specifications describe which regions of the graph are complete. These regions can be populated from an underlying text file by parsing the file and matching only records that belong to source nodes that the region (transitively) depends on. In this section we sketch how to build the grammar for the parser.

Initially, the grammar just consists of productions for recognising units in a dimension. These productions are built from the tables described in Section 3. The terminals in the grammar are the groups (they are the tokens recognised by the lexical analyser). Groups have an associated regular expression that defines their members as shown in Figure 2. The nonterminals are the derived nodes. There is one production in the grammar for each derived node in the graph. The body of the production consists of all the nodes on incoming edges. An example grammar in BNF for the units depicted in the graph in Figure 1 is given below. The terminals are the day groups.

1997 ::= *Jan_1997* | *Feb_1997* | . . . | Dec_1997;
Jan_1997 ::= *1_Jan_1997* | *2_Jan_1997* | . . . | *31_Jan_1997*;
. . .
Dec_1997 ::= *1_Dec_1997* | *2_Dec_1997* | . . . | *31_Dec_1997*;
1_Jan_1997 ::= `1_Jan_1997`;
. . .
31_Dec_1997 ::= `31_Dec_1997`;

So the token `1_jan_1997` would be recognised as *1_Jan_1997*, *Jan_1997*, and *1997*.

To complete the grammar, we need to add a production that identifies which records will be considered legal sentences. We only want records that provide information for a cubette. So we add a production that defines a record as the sequence of units in each cubette specification. For example, suppose that we have the following cubette specifications: `jan 1997@days`, `nov 1997@months`, and `1 dec 1997@days`. Then we would add the following production to the grammar.

Record ::= *Jan_1997* | *Nov_1997* | *1_Dec_1997*;

Record is the start symbol in the grammar. A parser for this grammar will only accept records pertinent to January, November, or December 1. Only these records will be needed to compute the values in the complete regions of the cube.

Edges

to	from
1997	jan 1997
1997	feb 1997
...	...
1997	dec 1997
jan 1997	1 jan 1997
...	...
jan 1997	31 jan 1997
...	

Groups

group	membership pattern
1 jan 1997	`"[1/Jan/1997"[^\]]+`
2 jan 1997	`"[2/Jan/1997"[^\]]+`
3 jan 1997	`"[3/Jan/1997"[^\]]+`
4 jan 1997	`"[4/Jan/1997"[^\]]+`
5 jan 1997	`"[5/Jan/1997"[^\]]+`
...	...
31 dec 1997	`"[31/Dec/1997"[^\]]+`
	...

Figure 2: An example of the tables that store the dependencies in a cube

This grammar can be fed directly into a parser generator (e.g., yacc) to construct a parser for populating an incomplete data cube. The grammar itself can be automatically generated from the tables and set of cubette specifications. Although we have not discussed the actions that need to be associated with each production, these actions can also be automatically generated.

There are, however, several problems with this solution that all have to do with the fact that available parser generation packages tend to be limited. The first is that for an N-dimensional cube, the resulting grammar will require N lookahead symbols. Most parser generators can only generate single lookahead parsers. The second problem is that top-down parsers will be much too slow, they might explore the entire cube graph for every record, so some bottom-up parsing technique has to be utilised. But bottom-up parsers usually generate tables that are proportional in size to the product of the number of terminals and the number of parsing states. There will be a large number of terminals (one for every unit in a dimension). Finally, if overlapping cubette specifications are allowed the generated grammar will be ambiguous (since some records might satisfy two or more cubettes). To handle this kind of ambiguous grammar the constructed parser should allow "multiple passes" over the token stream for some productions.

6 Conclusions

An incomplete data cube is a data cube that allows incomplete regions to exist in the cube hierarchy. Such cubes are useful when the analytic power of the cube organisation is needed, a complete, eager cube is too large, and the underlying source data for the incomplete regions is unavailable. We briefly described a real-world problem that fits these criteria. To analyse the WWW access log, a site administrator can specify which summary data to put in the cube by giving a set of cubette specifications, e.g., to monitor the use of lecture notes for an Introduction to Database course from machines on campus week-by-week during the first semester, the site administrator would add the following cubette specification (assuming the relevant units and measures exist).

```
jcu@machines, first_semester_1997@weeks, database_lecture_notes@pages
```

The set of cubette specifications is a high-level description of the information content of the cube. It is essential to querying the cube, and can also be used to automatically generate a data sieve for populating the cube.

References

[1] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proceedings of the 22st Conference on Very Large Databases*, pages 506–521, Mumbai, India, September 1996.

[2] C. Dyreson. Information retrieval from an incomplete data cube. In *Proceedings of the 22st Conference on Very Large Databases*, pages 532–543, Mumbai, India, September 1996.

[3] J. Gray, Bosworth A., A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proceedings of the 12th International Conference on Data Engineering*, pages 152–159, New Orleans, LA, 1996.

[4] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 311–322, Montreal, Canada, June 1996.

[5] A. Shukla, P. M. Deshpande, J. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. In *Proceedings of the 22st Conference on Very Large Databases*, pages 522–531, Mumbai, India, September 1996.

[6] J. Widom. Research problems in data warehousing. In *Proceedings of the 4th Int'l Conference on Information and Knowledge Management (CIKM)*, November 1995.

Some Approaches to Index Design for Cube Forests *

Theordore Johnson
University of Florida, Gainsville
ted@squall.cis.ufl.edu
and AT&T Laboratories
johnsont@research.att.com

Dennis Shasha
New York University
shasha@cs.nyu.edu

Abstract

The paradigmatic view of data in decision support consists of a set of dimensions (e.g., location, product, time period, ...), each encoding a hierarchy (e.g., location has hemisphere, country, state/province, ..., block) or nearly a hierarchy (there are exceptions: weeks don't fit into months and sizes don't fit into colors). Typical queries consist of aggregates over a quantifiable attribute (e.g., sales) as a function of at most one attribute in each dimension of this "data cube." For example, find the sum of all sales of blue polo shirts in Palm Beach during the last quarter. We discuss a few different alternatives to solve this problem including a data structure called cube forests *and its elaboration* hierarchically split cube forests *that exploit the hierarchical nature of the data to save space. We also discuss some design considerations.*

1 Introduction

Corporate and government executives must gather and present data before making decisions about the future of their enterprises. The data at their disposal is too vast to understand in its raw form, so they must consider it in summarized form, e.g. the trend of sales of such and such a brand over the last few time periods[3]. "Decision support" software to help them is often optimized for read-only complex queries. Companies such as Red Brick Systems, Teradata, Tandem, Masspar, Sybase, Praxis, and smaller companies such as Arbor Software all have products that cater to this market often using proprietary data structures. Other decision support products in the PC market are CA-Comptel, and Lotus Improv.

2 Approaches

We know of four main approaches to decision support:

1. Virtual memory data structures made up of two levels in which the bottom level is a one or two dimensional data structure, e.g., a time series array or a spreadsheet-style two dimensional matrix (time against

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*Partial support for this work came from the U.S. National Science Foundation grants IRI-9224601 and IRI-9531554 and by the Office of Naval Research under grant N00014–92-J-1719.

accounts) (see Figure 1. Arbor has a patent [2] that generalizes this so that the bottom level index can have many dimensions. These are the dense dimensions (the ones for which all possible combinations exist). The sparse dimensions are at a higher level in the form of a sparse matrix or a tree. Queries on this structure that specify values for all the sparse dimensions work quite well. Others work less well.

2. Bit map-based approaches in which a selection on any attribute results in a bit vector. Multiple conjunctive selections (e.g., on product, date, and location) result in multiple bit vectors which are bitwise Anded (see Figure 2. The resulting vector is used to select out values to be aggregated. Praxis [8] and the Sybase index accelerator use this approach. Bit vector ideas can be extended across tables by using join indexes.

3. RedBrick has implemented specially encoded multiway join indexes meant to support star schemas (http://www.redbrick.com/rbs/whitepapers/star_wp.html). These schemas have a single large table (e.g., the sales table in our running example) joined to many other tables through foreign key join, (to location, time, product type and so on in our example). Red Brick makes heavy use of such STARindexes to identify the rows of the atomic table (sales table) applicable to a query. Aggregates can then be calculated by retrieving the pages with the found rows and scanning to produce the aggregates. Our structure is also aimed at star schemas, except that our structure holds the aggregates directly making queries on aggregates significantly faster. Redbrick claims that STARindexes work well on non-star schemas too. Our basic structure does not support such a generalization.

4. Materialized views for star schemas. Gupta, Harinarayan, Rajaraman and Ullman[6, 5] present a framework for choosing good aggregate views and indices to materialize. While the problem is NP-Complete, the authors give heuristics which approximate the optimal solution extremely closely.

5. Massive parallelism on specialized processors and/or networks — Teradata and Tandem use this approach, making use of techniques for parallelizing the various database operators such as select, joins, and aggregates over horizontally partitioned data. Masspar[1] by contrast uses a SIMD model and specialized query pr

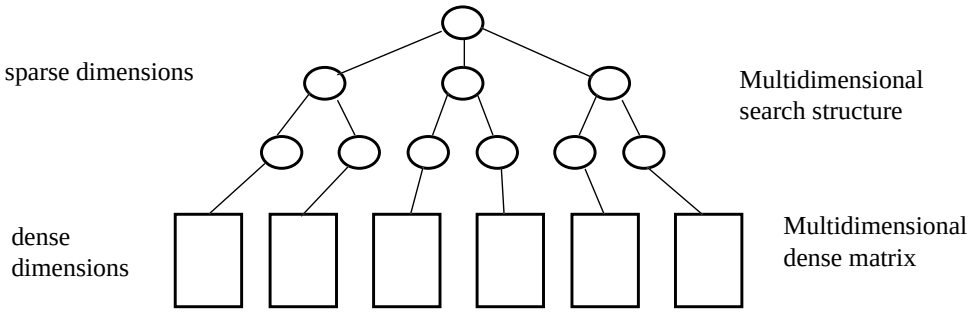


Figure 1: A two-level sparse/dense index.

We introduce a new data structure for this problem, which we call *cube forests*. Like bit vectors and two dimensional indexes, cube forests are oriented towards a batch-load-then-read-intensively system. As we will see, cube forests improve upon two level indexes for large data sets because cube forests treat all dimensions symmetrically and use standard disk-oriented indexes (there may be other reasons; the internal data structures of the PC vendors are wrapped in a veil of secrecy). Cube forests improve upon bit vectors for at least some applications because each bit vector query is linear in the number of rows in the table. Cube queries of the kind introduced in the abstract, by contrast, can be answered using a single index search in a cube forest, regardless of selectivities.

Attributes				Bit vectors				
A	B	C	V	1<A<10	B=5	C=10	B>9	0<C<9
0	5	7	3	0	1	0	0	1
12	4	15	2	0	0	0	0	0
6	3	8	1	1	0	0	0	1
12	5	10	7	0	1	1	0	0
8	10	3	4	1	0	0	1	1

Figure 2: Bit vectors defined on a relation.

The price of fast response to queries is duplication of information. As a result, cube forests have higher update costs and higher storage requirements than bit vectors. In many applications, this is the right tradeoff. Hierarchically split cube forests provide a method for efficiently duplicating information, and can be optimized to reduce update and storage costs. In summary, cube forests are most appropriate for read-intensive, update-rarely-and-in-large-batches multidimensional applications in an off-the-shelf (low cost) hardware environment.

3 Cube Forests

3.1 Preliminary Notions

To simplify the discussion, suppose our data is a single denormalized table whose attributes come from d dimensions denoting orthogonal properties (e.g., as above, product, location, time, organization, and so on). Each dimension c is organized hierarchically into n_c *dimensional attributes* $A_{c1}, A_{c2}, ..., A_{cn_c}$ where A_{c1} is the most general attribute (e.g., continent) and A_{cn_c} is the most specific (e.g., school district). Thus, the denormalized relation looks like this: $R(A_{11}, A_{12}, ..., A_{1n_1}, A_{21}, A_{22}, ..., A_{2n_2}, \ldots, A_{d1}, A_{d2}, ..., A_{dn_d}, value\_attributes)$. Here the value attributes are those to be aggregated, e.g., sale price, cost, value added, or whatever. This relation is denormalized because $A_{ij} \rightarrow A_{ik}$ when $j > k$, thus violating third normal form. (The key is $A_{1n_1}A_{2n_2}...A_{dn_d}$.)

We first present *cube forests*, in which every dimension consists of a single attribute, and then present the *hierarchically split cube forest*, in which dimensions are hierarchies of attributes. (Our techniques can also handle lattice-structured dimensions, though we don't show that here.) While our presentation uses a single aggregate value, our structure applies unchanged to multiple aggregates over multiple values.

3.2 The Basic Structure

The *instantiation* of a *cube tree* is a tree whose nodes are search structures (e.g. B-trees or multidimensional structures). Each node represents an index on one attribute (or a collection of attributes). Parent nodes store aggregate values over the values stored in their children[1]. A cube tree is specified by its *template,* which shows the (partial) order in which the attributes are indexed. Let us consider the simple example illustrated in Figure 3. Suppose that we index a table R first on A, then on B, then on C. The template for the cube tree is the list A-B-C, and we call this type of cube tree a *linear* cube tree. The *instantiation* is shown on the right side of Figure 3. The quantity inside each leaf node is the sum of the V attribute for a specific A, B, C value. At the next level up the quantity inside a node is the sum of the V attribute for a specific A, B combination. At the very top, we have the sum of V over the entire relation.

[1] Larry Laing of Data Fusion Technologies points out that such a structure can also be useful in the detection of phantom concurrency control conflicts.

We note that the index in our example is "cooked" to simplify its presentation. Since the relation is small, we can represent an attribute in the template as one level in the index. For a large relation, we need to define a strategy for implementing an index structure based on the template. For most of our discussion, we will assume that each attribute in the template corresponds to a separate index (for example, a B-tree). Our experiments with cube forest implementations have shown that a tight integration between the index structure and the cube forest algorithm is required for good performance. Cube forest *template* design and optimization can be performed ind

A	B	C	V
1	3	1	1
1	3	1	2
1	3	7	3
12	2	3	4
12	2	3	5
12	2	7	6
12	3	7	7
12	8	9	8

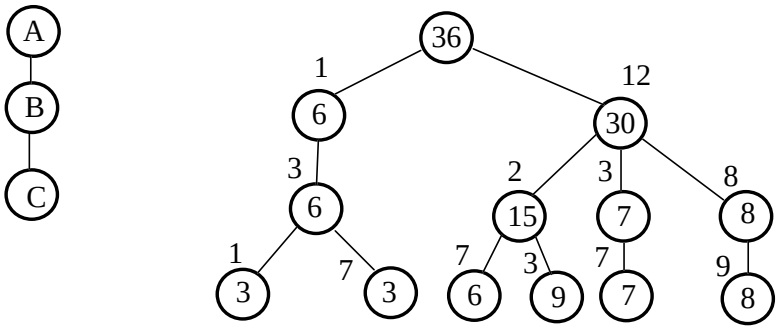


Figure 3: A cube tree template and its instantiation. Circled numbers represent the sum of V (i) over the entire relation (at the root, depth 0), (ii) for particular A values (depth 1), (iii) for particular AB combinations (depth 2), (iv) for particular ABC combinations (depth 3 or leaf level). Note that the instantiation is a tree even though the template is linear.

An interior node may have several children. In this case, each entry in a leaf of an index in the instantiation corresponding to the node has a pointer to a subindex for each template child of the node. This second feature leads to a tree topology for the template as shown in the left side of Figure 4. (The idea of using sequential combinations of attribute indexes as in our linear cube trees, though without aggregates, first appeared in Lum's seminal work[7].)

Let n be a node in a cube tree template. We define $attrib(n)$ to be the attributes indexed by n. Next, we define $pathattrib(n)$ to be the union of $attrib(m)$ for every m on the path from the root of the cube tree to n (inclusive). The aggregates in the instantiation of n are sums over particular combinations of values of the attributes in $pathattrib(n)$. Finally, we define $ancestattrib(n)$ to be $pathattrib(m)$, where m is the parent of n in the cube tree, or $\emptyset$ if n is a root. A cube tree template T is *well-formed* if the following two conditions hold:

1. For every n in T, $attrib(n)$ consists only of dimensional attributes (as opposed to value attributes such as sales).

2. For every n in T, $attrib(n) \cap ancestattrib(n) = \emptyset$.

That is, a cube tree template is well-formed if it contains neither unnecessary nor redundant attributes in any of its nodes. The definition of a well-formed *cube forest* template extends this definition, but requires the elimination of redundancies between trees. So, a cube forest template F is *well-formed* if the following two conditions hold:

1. Every cube tree template $T \in F$ is well formed.

2. Let n be a node in T and m be a node in T', where $T, T' \in F$. Then $pathattrib(n) = pathattrib(m)$ implies that $n = m$. If n were unequal to m in this case, the two nodes could be combined and their children combined.

For example, consider the cube forest shown in Figure 4 for the relation R of Figure 3. There are two trees, one described as A-(B,C) and the other (a linear tree) as B-C. The root of the first tree (A) has two children (B and C). Therefore, in the instantiation of the tree with data from R, each A value has two pointers, one to a subtree indexing, effectively, sums over the V attribute for AB combinations and one to a subtree providing sums for AC combinations. Note that B appears in both trees. But, in the first tree the path to B is A-B, while in the second tree the path is B. Since the paths are different, the forest is well-formed. This should be intuitive, since a node in the instantiation of B in the first tree corresponds to a sum over an AB pair whereas a node in the instantiation of B in the second tree is a sum over a B value.

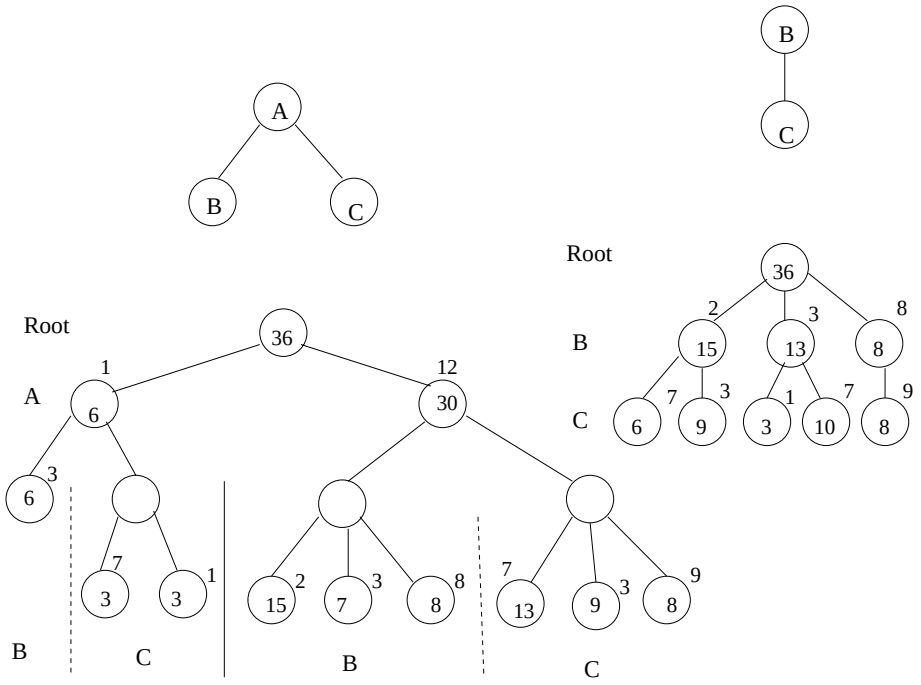


Figure 4: A cube forest template and its instantiation. Note that each A value in the instantiation of the left template has both B children and C children.

Given query q, let $point(q)$ be the set of attributes that the qeury specifies in either the where clause or the group-by clause. A cube forest F is *complete with respect to* q if there is a rooted path in F that contains all attributes in $point(q) \cup range(q)$. A cube forest is *complete* if it is complete with respect to q for every $q \in Q$.

Completeness ensures that a query will not have to scan the base relation. However, even complete cube forests can lead to bad query execution times. If there is an attribute that is in the rooted path but that is not in $point(q)$, then the execution will be forced to scan the corresponding level of the cube forest. Consider for example a query of the form (B : All, A : Point, C : Point) to be executed on the linear cube tree of Figure 3: every B value corresponding to the specified AC combination will be summed over. This observation motivates a stronger condition.

Forest F is *compatible* with query q if there is a rooted path P in F such that $attributes(P)$ equals $point(q) \cup range(q)$. Compatibility reduces the time needed to calculate a point query to a single index search.

4 Full Cube Forests

We would like to be able to construct a well-formed cube forest in which any point query can be answered by searching for a single node. That is, the structure should be compatible with every point query. We can construct

such a *full cube forest* F_i on i attributes, $A_1, \ldots, A_i$ recursively:

1. F_1 consists of a single node n labeled A_1.
2. To construct F_i,
   (a) Create a node n labeled A_i.
   (b) Create a copy of F_{i-1}. Make each tree in F_{i-1} a subtree of n.
   (c) Create another copy of F_{i-1}. F_i is the union of F_{i-1} and the tree rooted at n.

An example of a full cube forest is shown in Figure 5.

Theorem 1: The full cube forest F_n
(i) contains $2^n - 1$ template nodes, 2^{n-1} of which are leaves.
(ii) is compatible with any query on its n dimensional attributes.

Proof Note: (i) follows by solving the recurrence implicit in the construction: F_n consists of two instances of the template tree of F_{n-1} plus one new node.

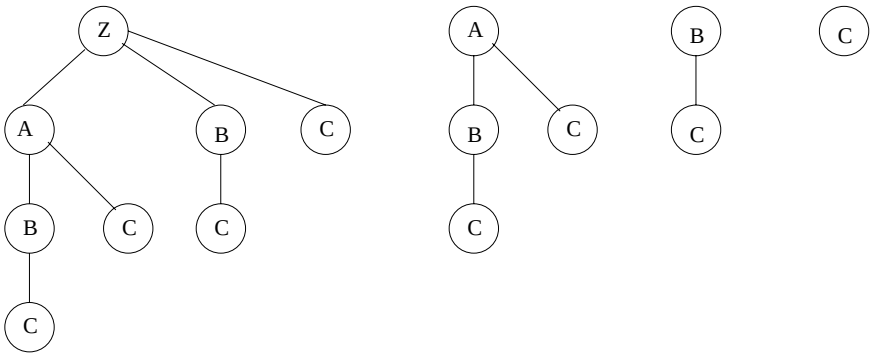


Figure 5: A full cube forest on the four attributes Z, A, B, C.

5 Hierarchically Split Cube Forests

From now on, we allow a dimension to contain multiple attributes that are related in a hierarchical (or, if you prefer, a one-to-many fashion). For example, the date dimension can be specified by year, month, or day with a one-to-many relationship from year to month and month to day (we defer the question of weeks to a longer paper). The attributes in a single dimension are *co-dimensional*. The template for a *hierarchically split cube forest* on dimensions $D_1, \ldots, D_n$ is a full cube forest on $D_1, \ldots, D_n$. (We call this full cube forest made up only of dimensions a *dimensional cube forest*.)

Each dimension D_i consists of attributes $a_{i,1}, \ldots, a_{i,k_i}$, where $a_{i,1}$ is coarsest (e.g., year) and a_{i,k_i} is finest (e.g., day). So, $a_{i,k+1}$ is the child of and functionally determines $a_{k,i}$. Each $a_{i,j}$ has as children (i) a full copy of the h-split forests corresponding to the forests of which D_i is the ancestor in the dimensional cube forest; and (ii) if $j < k_i$, a *co-dimensional child* $a_{i,j+1}$.

A full *h-split forest* (our abbreviation for hierarchically split cube forest) on three dimensions is shown in Figure 6. Dimensions A and C each have three attributes, and B has two attributes. The three rightmost trees in Figure 5 constitute its underlying full dimensional cube forest.

In Figure 6, the tree for dimension B and the tree for dimension C are attached to each of the attributes of dimension A (i.e., A1, A2, A3). This is called *splitting a dimension*. This may seem to be inefficient, but the

full h-split forest has far fewer nodes and takes much less space than a full cube forest that treats all attributes as orthogonal. Recall that a full cube forest on k attributes contains 2^k nodes in the template. Suppose that the h-split forest contains H dimensions, each of which contains A_i attributes, $i = 1, \ldots, H$. Let N_i be the number of nodes in the h-split forest template for i dimensions. Then,

Theorem 2: Including a node that gives the total value of the aggregate over all data instances, $N_H = \prod_{i=1}^{H}(A_i + 1)$

The theorem holds because of the recurrence $N_i = N_{i-1} + A_i * N_{i-1}$. Note that the number of root to leaf paths is $\prod_{i=1}^{H-1}(A_i + 1)$. Consider the example in Figure 6. The h-split forest has 48 nodes, 12 of which are leaves. A full cube forest on the same set of attributes would have 255 nodes, 128 of which would be leaves.

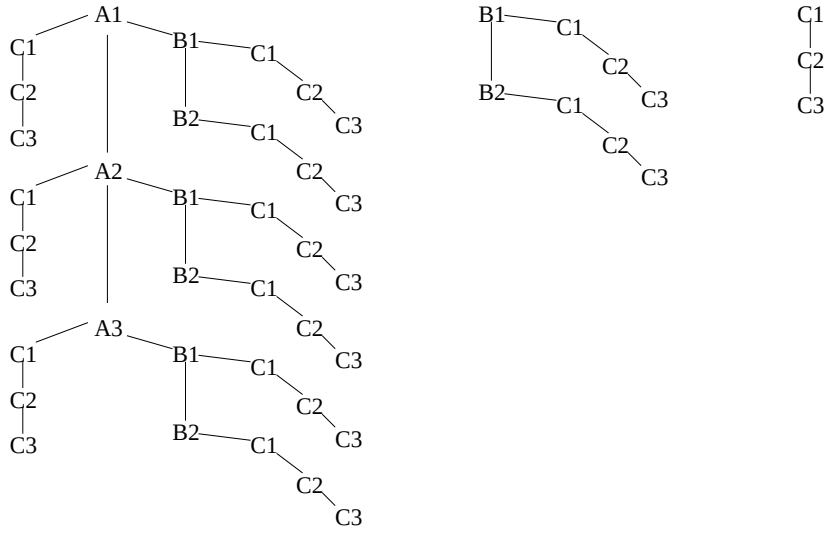


Figure 6: Template for a hierarchically split cube forest on 3 dimensions (A, B, and C).

How good is a full hierarchically split cube forest? Consider the style of queries we discussed at the beginning. Ostensibly a full h-split forest is not compatible with most queries, since any query referencing $a_{i,j}$ must also reference $a_{i,k}$ for $1 <= k < j$. This does not force a scan, however, because $a_{i,j}$ has a many-to-one relationship with all such $a_{i,k}$, as opposed to being an orthogonal attribute.

We formalize this intuition as follows.

A *point cube query* is a cube query restricted to the form: (A_1 : point, A_2 : point, ... , A_n : point, D_1 : all, D_2 : all, ... , D_k : all) such that A_i is in a different dimension from A_j when i and j are different. Also, neither A_i nor A_j is in dimension D_m for $1 <= m <= k$. In other words, either no attribute in a dimension D is specified or exactly one attribute in D has a point constraint. A rooted path P through an h-split forest is *hierarchically compatible* with a point cube query q if the following holds: (i) P touches every attribute A in point-attribute(q) but touches no co-dimensional child of A. (ii) P touches no attributes from dimensions outside point-dimension(q).

To gain intuition about hierarchical compatibility, consider the point cube query (A_2 : point, C_1 : point, B : all) on the forest in Figure 6. Suppose, for concreteness, the query specifies A_2 to be 313 and C_1 to be 1711. A search descends the instantiation of the leftmost template tree of the figure until the search reaches a node n containing the A_2 value of 313. The search continues down the C subtree of n until it reaches the node m in the instantiation of template node C_1 containing 1711. Node m contains, therefore, the aggregate, say sum of V, where $A_2 = 313$ and $C_1 = 1711$.

We can reduce the number of nodes (and the attendant storage and update costs) still further, if we are willing to increase the query costs. To bound this increase, we exploit the fact that many of the attributes summarize only

a few values from the attribute immediately lower in the hierarchy – only 12 months in a year, only a few states in a region. It might be acceptable to leave out all children of, say, the year node other than its co-dimensional child month. The reason is that we can compute a per-year aggregate by summing twelve per-month aggregates. We call this operation pruning.

Pruning attribute $A_{i,j}$ means eliminating all children of $A_{i,j}$ (and subtrees rooted at those children) other than its co-dimensional child $A_{i,j+1}$. An attribute may be pruned only if it has co-dimensional children, because that would cause an unrecoverable loss of information (one can calculate the sum of sales over years from the sum of sales over months if years are pruned but not vice versa).

Pruning a node high in a tree eliminates many template nodes, because all descendant trees will be pruned. However, the cost imposed on the worst case queries multiplies whenever two pruned attributes are related by ancestor-descendant relationships within the same dimension or come from different dimensions. For example, if we prune months alone, then each year calculation is slowed down by a factor of 12 compared to an unpruned h-split forest; if we prune month and state, however, and we want sales from a specific month and state, then the calculation is slowed down by a factor of 12 times the number of counties per state.

5.1 Tuning Considerations

The designer of a cube forest has many options – how to "stack" the dimensions and how to prune the resulting cube forest. We are designing algorithms to compute optimal cube forest templates. However, many people prefer to use a rule of thumb. Below, we give some guidelines to cube forest design:

1. Frequently issued queries should have compatible paths at their disposal.
2. Put the dimension with the largest number of attributes at the "bottom" dimension in the forest, to minimize the number of root-to-leaf paths in the template.
3. Put dimensions whose attributes have only a few unique values in a batch "high" in the forest (i.e., the Time dimension for daily feeds.)
4. Dimensions that are frequently involved in range queries should be "low" in the forest.
5. It is better to prune attributes with a small number of unique values, and preferably an attribute that is low in a dimension but high in a given tree.
6. Each tree in a hierarchically split cube forest can be pruned independently.

6 Conclusions

Hierarchically split cube forests constitute a new structure for multi-dimensional hierarchical decision support. In a full h-split forest, point cube queries of the form "Find the sales of all Mustangs in New England in the first quarter of last year" can be answered in a single index lookup yielding sub-second response time. Other general purpose structures such as bit vectors would require at least linear time searches for such queries.

Our experiments suggest both that the hierarchically cube forest structure gives good performance on large data. Batch updates show good use of main memory.

7 Acknowledgments

An early version of the data cube paper of Gray, Bosworth, Layman and Pirahesh [4] turned us on to this problem. Jim Gray also made helpful comments on early drafts of this idea.

References

[1] Latha S. Colby, Nancy L. Martin and Robert M. Wehrmeister. “Query Processing for Decision Support: The SQLmpp Solution”. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, 1994, Austin, Texas, pp. 121-130.

[2] “Method and Apparatus for Storing and Retrieving Multi-dimensional data in Computer Memory”, 05359724, Robert J. Earle.

[3] Maurice Frank, “A Drill-down Analysis of Multidimensional Databases ” DBMS, vol. 7, no. 8, pp. 60ff July 1994.

[4] Jim Gray, Adam Bosworth, Andrew Layman and Hamid Pirahesh “Data Cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals” Proc. of the 12th International Conference on Data Engineering pp. 152-159, 1996

[5] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, Jeffrey D. Ullman “Index Selection for OLAP” Stanford University Computer Science Technical Note 1996.

[6] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman “Implementing Data Cubes Efficiently” in *Proceedings of ACM SIGMOD 96* Montreal, Canada, pp. 205-216.

[7] V.Y. Lum, “Multi-attribute Retrieval with Combined Indexes” Communications of the ACM, vol. 13m no 11 Nov. 1970 pg. 660-665.

[8] Patrick O’Neil, “Model 204: Architecture and Performance” *High Performance Transaction Systems* Lecture Notes in Computer Science Number 359, September, 1987. Springer-Verlag, eds. D. Gawlick, M. Haynie, A. Reuters

Indexing OLAP data

Sunita Sarawagi
IBM Almaden Research Center
sunita@almaden.ibm.com

Abstract

In this paper we discuss indexing methods for On-Line Analytical Processing (OLAP) databases. We start with a survey of existing indexing methods and discuss their advantages and shortcomings. We then propose extensions to conventional multidimensional indexing methods to make them more suitable for indexing OLAP data. We compare and contrast R-trees with bit-mapped indices which is the most popular choice for indexing OLAP data today.

1 Introduction

Decision support applications are increasingly relying on data warehouses to understand and analyze their businesses. These applications often require fast interactive response time to a wide variety of large aggregate queries on huge amounts of data. Current relational database systems have been designed and tuned for On-Line Transaction Processing (OLTP) and are inadequate for these applications. On-Line Analytical Processing (OLAP) [CCS93] databases have been designed to close this gap and meet the needs of decision support applications. As a result, several OLAP products have appeared on the market (see [Rad95], [Gri96] for a survey). These systems provide fast response time by pre-computing a large number of anticipated aggregate queries [GBLP96, AAD+96, HRU96] and making extensive use of specialized indexing methods on multiple attributes of the data. In this paper, we will discuss the problem of indexing OLAP data.

Contents We first consider an example OLAP database and review relevant terminologies in Section 1.1. We then discuss the desired features of a good OLAP indexing method and highlight why indexing OLAP data is different from indexing OLTP data in Section 1.2. We then briefly discuss in Section 2 some of the popular indexing methods in use today. One alternative that has not been explored is using conventional multidimensional indexing methods like R-trees for indexing OLAP data. In Section 3 we discuss how R-trees can be modified to take advantage of the special characteristics of OLAP data and thus be more useful for indexing OLAP data. We argue how, in many cases, R-trees could out-perform the popularly used bit-mapped indexing methods.

1.1 Terminology

Consider a database that contains point of sale data about the sales price of `products`, the date of sale and the store which made the sale. Conceptually, this cube can be viewed as a multidimensional cube. In this cube, at-

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

tributes like `product, date, store` which together form a key are referred to as *dimensions*, while the attributes like `sales` are referred to as *measures*. There can be multiple measures like, `quantity`, `projected sales`, `profit` associated with a given record. Dimensions usually have associated with them *hierarchies* that specify aggregation levels and hence granularity of viewing data. Thus, $day \rightarrow month \rightarrow quarter \rightarrow year$ is a hierarchy on `date`. Similarly, $product\ name \rightarrow type \rightarrow category$ is a hierarchy on the `product` dimension. In addition to hierarchies, dimensions can have other attributes (called *non-dimension attributes*) like "`color of product`" and "`owner of store`" associated with them. There are two implementation approaches for OLAP data: MOLAP (multidimensional OLAP) where data is stored as a multidimensional *data cube* with the dimensions forming the axis of the cube and ROLAP (relational OLAP) where data is stored in tables. ROLAP systems often organize data using the *star-schema* where a central *fact table* stores the encoded dimension values (like 4-byte `product`-id, store-id, data-id) and all the measures and individual *dimension tables* store hierarchies and other associated non-dimension attributes for each dimension. See [CD96] for a detailed survey.

1.2 Requirements on an indexing method

Example 1: We give below some queries to provide a flavor of multidimensional queries. These queries use the cube from the previous section.

- Give the total sales for each `product` in each quarter of 1995.
- In 1995, for each store give the `products` with the top 5 sales.
- For store "Ace" and for each `product`, give difference in sales between Jan. 1995 and Jan. 1994.
- Select top 5 stores for each `product category` for last year, based on total sales.
- For each `product category`, select total sales this month of the `product` that had highest sales in that `category last month`.
- Select stores that currently sell the highest selling `product` of last month.
- Select stores for which the total sale of every `product` increased in each of last 5 years.

Based on these queries, we can enlist the following requirements on a good OLAP indexing method.

Symmetric partial match queries Most of the OLAP queries can be expressed conceptually as a partial range query where associated with one or more dimensions of the cube is a union of range of values and we need to efficiently retrieve data corresponding to this range. The extreme case is where the size of the range is one for all dimensions giving us a *point query*. The range could be continuous, for instance, "`time between Jan '94 to July '94`" or discontinuous, for instance, "`first month of every year`" and "`product IN {soap, shirts, shoes}`". It is desirable to extend the index traversal techniques to allow a collection of key values to be searched simultaneously instead of doing the search one value at a time. Typically, the cardinality of the range is one for most dimensions except a few. Also, there is no fixed set of dimensions on which the predicates are applied: therefore, ideally we would like to *symmetrically* index all dimensions of the cube.

Sometimes, it might be necessary to index the non-dimension attributes of a dimension too. The measure attributes can also be treated as dimensions in some cases and it is useful to index them. For instance, an analyst might be interested only in `products` whose total sales is greater than some amount. Queries of the form: "top-5 selling `products` in each `category`" are also common in OLAP and could benefit from a combined index on `product category` and sales.

OLTP queries differ from the OLAP queries discussed above in that most OLTP queries typically access small amounts of data. In OLTP databases, point queries are more common. Also, multiple predicates on many attributes at the same time is less common than in OLAP application.

Indexing at multiple levels of aggregation Most OLAP databases pre-compute multiple group-bys corresponding to different levels of aggregations of the base cube. For instance, groupbys could be computed at the <product-store> level, <product-time> and <time> level for a base cube with dimensions product, store and time. It is equally important to index the summarized data. An issue that arises here is whether to build separate index trees for different levels of aggregation or whether to add special values to the dimension and index precomputed summaries along with the base level data. For instance, if we index sales for <product-year>, then we can store total-sales at the <product> level by simply adding an additional value for the year dimension corresponding to "ALL" years and storing total-sales for each product there. Similarly values along hierarchies can be handled by extending the domain of the dimensions.

Multiple traversal orders B-trees are commonly used in OLTP systems to retrieve data sorted on the indexed attribute. This is often a cheaper alternative to doing external sorts. OLAP databases, because of the large number of group-bys that they perform, can also benefit from using indices to sort data fast. The challenge is in allowing such a feature over multiple attributes instead of a single one and also for any permutation of any subset of the attributes.

Efficient batch update OLAP databases have the advantage that frequent point updates as in OLTP data is uncommon. However, the update problem cannot be totally ignored. Making batch updates efficient is absolutely necessary. It is not uncommon for multinational organizations to update data as high as four times a day since daily data from different locations of the world appear at different times. On the other hand, these updates are clustered by region and time. This property can be exploited in localizing changes and making updates faster.

Handle sparse data Colliat in [Col96] states that typically 20% of the data in the logical OLAP cube are non-zero. However, as the OLAP model is finding newer applications, it is desirable to have an indexing method that is not tied to any fixed notions of sparsity. Therefore, the ideal method should scale well with increasing sparsity.

2 Existing methods

We classify existing indexing methods into four classes. The first class consists of methods that are based on using multidimensional arrays. The methods of the second class are based on bit-mapped indices. The third class consists of hierarchical methods and finally the fourth class includes conventional multidimensional indices originally designed for spatial data.

2.1 Multidimensional array-based methods

Logically, the OLAP data cube can be viewed as a multidimensional array with the key attributes forming the axis of the array. The ideal indexing scheme for this logical view of the data would have been a multidimensional array if the data cube were dense. Any exact or range query on any combination of attributes could have been easily answered by algebraically computing the right offsets and fetching the required data. But since most OLAP data is not dense, several alternatives have been proposed that attempt to handle sparsity while staying as close as possible to the array model. A good example of this is Essbase's proprietory indexing scheme [Ear94] that we discuss next.

In Essbase [Ear94], the user identifies a set of dimensions of the cube that are dense meaning that each combination of values formed out of the dense dimensions has high likelihood of data associated with it. These are the dense dimensions D, the remaining dimensions belong to the sparse set S. A index tree is constructed on the combination of values of the sparse dimensions. Each entry in the leaf of the index tree points to a multidimensional array formed by the dense dimensions D. This array, called a *block*, stores in its cells all the measure

values. The dimension fields (which are often arbitrary strings) are mapped to continuous integers which determine the contiguous positions of the multidimensional arrays. The arrays of dense dimensions may not all be dense. Therefore, they are further compressed where necessary.

Consider an example of a four dimensional cube where `product` and `store` are identified as sparse dimensions and `time` and `scenarios` belong to the dense set. A B-tree index is built on the `product-store` pair and for each pair of values where data is present, a 2-dimensional array of `time` and `scenarios` is stored. When a query arrives with a restriction on one or more sparse dimensions the index tree is searched on the sparse dimension, and for each matching block, the restrictions if any, on the dense dimensions are used to get the right offsets in the block. Before searching the index tree the mapping tables are used to convert required values to their integer maps.

We will now evaluate how this method meets the requirements of Section 1.2. Assume that a B-tree is used to index a concatenation of fields from the sparse dimensions. A point query is fast since we first search the B-tree on the sparse dimensions and then calculate the array offsets to reach the *data* in the dense dimension. The hope is that the sparse index is small and can fit in memory [Col96]. Thus, we go to the disk only for data. When the sparse index does not fit in memory, the performance of a query that retrieves a range of values on a dimension that is not one of the outermost dimension will result in multiple searches. In the above example, if we built a B-tree on the concatenation of `product` and `store`, then a query of the form, "`store = Ace`" will require us to make multiple searches for different values of `products`. Note that other methods like R-trees will be better in this regard. If there are predicates only on the dense dimensions, one can directly calculate the right offsets in the blocks and visit the linked array blocks in turn. It is not clear, how non-dimension and summary attributes are indexed. Batch updates with this method is efficient because the data can be first sorted in the index order (the sparse dimensions first in the same order as the compound key, the dense dimensions later in the same order as the array storage order) and then the index structure can be updated as a batch. Precomputed summaries are stored in the same index as the base cube. The success of this method depends on the ability to find enough dense dimensions, failing which, this reduces to B-tree on multiple attributes and inherits are its disadvantages [LS90].

2.2 Bit-mapped indices and variations

When data is sparse, a good option is not to index the multidimensional data space but index each of the dimension space separately as in bit-mapped indices. This is a popular method used by several vendors. Different vendors have different variants of the basic method [OG95] that we discuss next.

Each dimension of the cube has associated with it a bit-mapped index. In the simplest form, a bit-mapped index is a B-tree where instead of storing RIDs for each key-value at the leaf, we store a bit-map. The bit-map for value "v" of attribute "A" is an array of bits where each bit corresponds to a row of the fact table (or a non-empty cell of the data cube). The bit is a "1" only at those positions where the corresponding row has value "v" for attribute A.

Exact match queries on one or more dimension can be answered by intersecting the bit maps from multiple dimensions. Limited kinds of range queries can also be answered by ORing the bit-maps for different values of the same dimension and finally ANDing them with the bit-map of the other dimension as suggested in [OG95].

Major advantages of this method is that: (1) for low cardinality data, bit maps are both space and retrieval efficient. Bit-operations like AND/OR/NOT/COUNT are more efficient than doing the same operations on RID lists. (2) Access to data is clustered since the bit-map order corresponds to the data storage order. (3) All dimensions are treated symmetrically and sparse data can be handled the same way as dense data. (4) If required, data can also be retrieved in any arbitrary sorting order of dimensions by traversing the bit-maps in a certain order. However, using the indices to retrieve data in a particular order, can result in a loss of the clustered data access property discussed in item (2).

The major disadvantages of bit-mapped indices is: (1) ORing bit maps for range queries might be expensive. The number AND operations is limited to the number of dimensions and is therefore small in most cases. How-

ever, the number of OR operations can be large for many queries since each value in a range of a dimension will incur a OR operation. (2) the increased space overhead of storing the bit-maps especially for high cardinality data. (3) Batch updates can also be expensive since all bit-map indices will have to modified for even a single new row insertion.

In short, this approach is only viable when the domain of each attribute is small. Otherwise, the space overhead and the bit processing overhead could be probatively large. We next discuss some of the techniques used by vendors to deal with these disadvantages.

Compression Bit-maps are often compressed to reduce space overhead. But, this implies that the overhead of decompression has to be incurred during retrieval or one has to rely on methods of doing "AND" and "OR" operations on compressed bit-maps [GDCG91]. For simple compression schemes like run-length encoding it is easy to design AND/OR algorithms that work on compressed data. However, it is necessary to keep the cost of these operations low since one of the main reasons for using bit-mapped indices is that it enable fast bit operations.

Hybrid methods Since bit-maps are not appropriate for high cardinality data, some products [Ede95] follow a hybrid approach where a plain B-tree is used when the list of qualifying RIDs per entry is small, otherwise a bit-mapped index is used.

Dynamic bit-maps Another approach (used by some vendors) to handle high cardinality data and large range queries is to construct the bit-maps dynamically from vertically partitioned fact table as follows. Each column stores a compressed representation of the values in the column attribute. For instance, if there are n different values of a particular attribute, we map the values to continuous integers and represent each value in the column by only $\log n$ bits which represents its integer map. When a predicate requires a subset of values in that column, the required values are converted to their integer maps and represented in an in-memory array or hash-table. Now, the column partition is scanned and for each value, the in-memory array is probed. Depending on whether a match is found or not, a 1 or a 0 is stored at the row position of a bit-map that is constructed dynamically. This process is repeated for predicates on other columns. At the end of scanning all queried columns we have a bit-map with a 1 at the row positions that satisfied all predicates. This bit-map can be further AND-ed with a bit-map obtained from a bit-mapped B-tree index on a low-cardinality column.

2.3 Hierarchical indexing methods

Both of the above schemes index data aggregated at different levels of detail the same way. Thus, measures summarized at the `product` level are indexed the same way, in the same indexing structure as measures at the `product-store` level. A different approach is followed by hierarchical indexing methods as proposed by Dimensional Insight [Pow93] and Johnson and Shasha [JS96]. In these schemes, we first build an index tree on the `product` dimension and store summaries at the `product` level. Each `product` value, contains a separate index at the store level and stores summaries at the `product-store` level and so on. Summaries at the `store` level are kept in a separate index tree on `store`. In general, the number of such index trees can grow exponentially, [JS96] discusses how to cut down the number of trees based on commonly asked queries.

The main advantage of the hierarchical indexing schemes is that data at higher levels of aggregations that is typically accessed more frequently can be retrieved faster than the larger detailed data. Also, dimensions are symmetrically handled and data can be retrieved in a sorted order for several permutation of dimensions. The main disadvantage is the widely increased index storage overhead and thus a decrease in update efficiency. The average retrieval efficiency can also suffer because large indexing structures often implies poor caching and disk performance.

2.4 Multidimensional indices

Another alternative for indexing OLAP data is to apply one of the many existing multi-dimensional indexing methods designed for spatial data (see [Gut94] for an overview). This alternative is not well explored in the commercial arena. The widely cited reasons being that these schemes do not scale well with increasing dimensionality and that for predicates on multiple categorical attributes a cartesian product of the keys will need to be searched. However, some of these indexing methods offer certain advantages which should be deployed in indexing OLAP data, albeit with some modifications. One of the key features of several indexing schemes like R-trees and Grid files is symmetric treatment of all dimensions without incurring the space overhead of the hierarchical indexing methods or processing overhead of doing bit operations as in bit-mapped indices. We discuss in Section 3 how some OLAP-specific optimizations can be applied to such indexing methods. This is a summary of the results being reported in [Sar97].

3 Optimizing R-trees for efficient access to OLAP data

The OLAP matrix is sparse but not uniformly so. There are typically rectangular shaped regions of density. For instance, a supplier might be selling to stores only in a particular region. Hence, for that supplier all other regions will have NULL values. Ideally, we do not want to explicitly index dense regions since we can directly compute the array offset. We propose extending the indexing method to allow nodes of two types: 1. rectangular dense regions that contain more than a threshold number of points in that region and 2. points in sparse regions. For the dense clusters we only store the boundaries. It is like doing run-length encoding on points in multidimensional space. The dense cluster itself is stored as a multi-dimensional sub-array elsewhere. For instance, if we can find a 10 by 10 rectangular dense cluster of 100 points in an otherwise sparse two-dimensional array, we can index all 100 points in that sub-array using a single rectangle instead of inserting 100 individual points in the index. The index entry for the rectangle would point to the 100-point sub-array stored elsewhere. Searching the index for a point in the sub-array would first return the boundaries of the rectangle and then we use array offset calculations to reach to the exact point in the sub-array. This idea of indexing dense regions generalizes the approach used by Arbor software for indexing OLAP data. Their approach is to manually identify dense and sparse dimensions. We believe that, one is more likely to find dense regions rather than dense dimensions. For instance, if we have a 2-D data cube with the bottom-right quarter dense and the rest of the region sparse, the Arbor approach will not be able to extract any dense dimensions.

3.1 Storing Dense clusters

The issues that arise in storing the dense clusters are similar to the ones used for storing duplicates in a B-tree access method as discussed in [GR94]. Each dense-cluster entry in the R-tree contains the boundary of the dense cluster and a pointer to a variable length array. The array itself can be organized in one of two ways. Each entry of the array can either be (1) a TID (tuple identifier) or (2) the tuple itself. In either case, the entries of all sub-arrays can be concatenated one after another and stored as a single tuple stream as discussed in [GR94]. The advantage of the second approach is lower storage overhead and fewer I/Os during retrieval. But the disadvantage is that, the indexed relation has to be organized in the order determined by each dense clusters. An advantage of the first approach is that missing combinations in any dense cluster will incur smaller storage overhead than with the second approach. In either case, we can use array clustering techniques as discussed in [SS94, Jag90] to improve spatial locality instead of storing the array in the linear fortran order that destroys spatial locality.

3.2 Finding Dense clusters

In many cases, the dense clusters can be identified at the high level by a domain expert. For instance, it might be possible for the DBA, to infer that certain stores sell all `products` or that some collection of `products` are sold everyday in every store or that woolen clothes are sold in all store in all winter months and so on. In the absence of such knowledge, one can use clustering algorithms to automate the search for dense regions. The requirements on the clustering algorithms are slightly different in our case because we require that each cluster be rectangular. Hence, our goal is to find rectangular shaped regions so that the fraction of points present in any such regions is more than such fixed threshold. Such algorithms [BR91, SB95] are common-place in image analysis and other applications of similar flavor. It is necessary, however, to evaluate how these algorithms scale with input size and how well they handle arrays of dimensionality greater than two.

3.3 Comparing Bit-mapped indices and R-trees

Multi-dimensional index trees like R-trees have a number of advantages over bit-mapped indexing schemes. Exact match and range queries on multiple dimensions can be answered by simply searching the index tree. The overhead of bit ANDing/ORing operations can thus be avoided. The space overhead will be smaller because R-trees only index the region where points are present and hence do not index the "0"s as in the bit-mapped techniques. R-trees also can be expected to be more space-efficient than bit-mapped indices especially when the bit-maps are not compressed. When they are compressed, the overhead of ANDing/ORing will increase and thus the retrieval performance could suffer. R-trees also are more efficient to update than bit-mapped indices.

However, there could be situations where a search on the R-tree could take longer than bit ANDing/ORing operations on an index tree especially when the query rectangle is large. R-trees are better when most of the dimensions have predicates on them. Their performance can be expected to be worse when only a few dimensions are restricted, this may not be a big handicap since most OLAP queries leave at most two dimensions unspecified since cross-tabular presentations are cumbersome for larger than two dimensions. The exact difference will depend on the number of such dense clusters one can find.

Thus, in conclusion, we can state that, R-trees should be preferred when partial search queries have few unspecified dimensions, the dense regions are large, overhead of bit operations are high, and updates more common. Bit-maps should be prefered when dimensions have low cardinality, queries have few restricted dimensions and data is very sparse so that the chance of finding dense regions is small.

References

[AAD+96] S. Agarwal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. of the 22nd Int'l Conference on Very Large Databases*, pages 506–521, Mumbai (Bombay), India, September 1996.

[BR91] M. Berger and I. Regoutsos. An algorithm for point clustering and grid generation. *IEEE transactions on systems, man and cybernetics*, 21(5):1278–86, 1991.

[CCS93] E. F. Codd, S. B. Codd, and C. T. Salley. Beyond decision support. *Computerworld*, 27(30), July 1993.

[CD96] S. Chaudhuri and U. Dayal. Decision support, data warehousing and olap, 1996. VLDB tutorial, Bombay, India.

[Col96] G. Colliat. Olap, relational and multidimensional database systems. *SIGMOD Record*, 25(3):64–69, Sept 1996.

[Ear94] Robert J. Earle. Arbor software corporation, u.s. patent # 5359724, Oct 1994. `http://www.arborsoft.com`.

[Ede95] Herb Edelstein. Faster data warehouses. TechWeb, Dec 4 1995. http://techweb.cmp.com/iwk/.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tabs and sub-totals. In *Proc. of the 12th Int'l Conference on Data Engineering*, pages 152–159, 1996.

[GDCG91] E.L. Glaser, P. DesJardins, D. Caldwell, and E.D. Glaser. Bit string compressor with boolean operation processing capability, July 1991. U.S. Patent # 5036457.

[GR94] J. Gray and A. Reuter. *Trasaction Processing: Concepts and Techniques*, chapter 15, pages 858–60. 1994.

[Gri96] Seth Grimes. On line analytical processing, 1996. http://www.access.digex.net/ grimes/olap/.

[Gut94] R. H. Guting. An introduction to spatial database systems. *VLDB Journal*, 3(4):357–399, 1994.

[HRU96] V. Harinarayan, A. Rajaraman, and J.D. Ullman. Implementing data cubes efficiently. In *Proc. of the ACM SIGMOD Conference on Management of Data*, June 1996.

[Jag90] H V Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, 1990.

[JS96] T. Johnson and D. Shasha. Hierarchically split cube forests for decision support: description and tuned design, 1996. Working Paper.

[LS90] David Lomet and Betty Salzberg. The Hb-Tree: a multiattribute indexing method with good guaranteed performance. *ACM TODS*, 15(4):625–658, December 1990.

[OG95] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, Sept 1995.

[Pow93] S.R. Powers, F.A. Zanarotti. Database systems with multidimensional summary search-tree nodes for reducing the necessity to access records, Oct 1993. U.S. Patent # 5,257,365.

[Rad95] Neil Raden. Data, data everywhere. *Information Week*, pages 60–65, October 30 1995.

[Sar97] Sunita Sarawagi. Techniques for indexing olap data. Technical report, IBM Almaden Research Center, 1997. In preparation.

[SB95] P. Schroeter and J. Bigun. Hierarchical image segmentation by multi-dimensional clustering and orientation-adaptive boundary refinement. *Pattern Recognition*, 28(5):695–709, May 1995.

[SS94] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *Proceedings of the tenth international Conference on Data Engineering*, Feb 1994.

CALL FOR PAPERS

DATA ENGINEERING

14th International Conference on Data Engineering

February 23 - 27, 1998

Adam's Mark Hotel, Orlando, Florida, USA

Sponsored by the IEEE Computer Society

IEEE

SCOPE

Data Engineering deals with the use of engineering techniques and methodologies in the design, development and assessment of information systems for different computing platforms and application environments. The 14th International Conference on Data Engineering will continue in its tradition of being a premier forum for presentation of research results and advanced data-intensive applications and discussion of issues on data and knowledge engineering. The mission of the conference is to share research solutions to problems of today's information society and to identify new issues and directions for future research and development work.

TOPICS OF INTEREST

For a detailed list of topics of interest, as well as other conference details, see our Web page.

PAPER SUBMISSION

Six copies of original papers not exceeding 6000 words (25 double spaced pages) should be submitted by June 2, 1997 to the Program Co-Chair:

Susan Urban
Department of Computer Science and Engineering
Arizona State University
P.O. Box 875406
Tempe, AZ 85287-5406 USA
Phone: +1-602-965-2784
Fax: +1-602-965-2751
E-mail: s.urban@asu.edu

Authors are also required to send an abstract of the paper by e-mail to the above address by May 26, 1997. The message should be in ASCII format and contain the title of the paper, authors' names, the abstract, the name of one or two areas most relevant to the paper and whether the paper is submitted to the Industrial program

INDUSTRIAL PROGRAM

The conference program will include a number of sessions devoted to industrial developments, applications, and experience in using databases. Papers intended for this program should be clearly marked as industrial track papers at the time of submission. Questions concerning possible submission to this program should be directed to the industrial program chair.

PANELS, TUTORIALS and RESEARCH PROTOTYPE DEMONSTRATIONS

The research and the industrial track will be complemented by panels, tutorials, and research prototype demonstrations. Proposals for each program should be sent to the appropriate track chair by June 2, 1997.

Panel Submissions to: Prof. Calton Pu, Dept. Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, P.O. Box 91000 Portland, OR 97291-1000 USA. Shipping address (for express mail only):20000 N.W. Walker Road, Beaverton, OR 97006 USA

Tutorial Submissions to: M. Tamer Ozsu, Professor, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1, E-mail: ozsu@cs.ualberta.ca

Research Prototype Demonstrations to: Dr. Sharma Chakravarthy, Computer and Information Science and Engineering, E470 Computer Science and Engineering Building, University of Florida, PO Box 116125, Gainesville, FL 32611-6125 USA, E-mail: sharma@cise.ufl.edu

PUBLICATIONS & AWARDS

All accepted papers will appear in the Proceedings published by the IEEE Computer Society. Authors of selected papers will be invited to submit extended versions for possible publication in the *IEEE Transactions on Knowledge and Data Engineering* and in the *Journal of Distributed and Parallel Databases*. An award will be given to the best paper. A separate award honoring K.S. Fu will be given to the best student paper (authored solely by students).

IMPORTANT DATES

- Abstracts of paper submissions: May 26, 1997
- Paper submissions: June 2, 1997
- Panel, Tutorial and Demonstration Proposals: June 2, 1997
- Notification of Acceptance: September 22, 1997
- Final Papers Due: November 15, 1997
- Tutorial Dates: February 23-24, 1998
- Conference Dates: February 25-27, 1998

ORGANIZING COMMITTEE

Steering Committee Chair: Joseph E. Urban, Arizona State University, USA
General Chair: Philip Yu, IBM T. J. Watson Research Center, USA
Program Co-Chairs: Susan D. Urban, Arizona State University, USA
Elisa Bertino, University of Milano, Italy
Industrial Program Chair: Surajit Chauduri, Microsoft Corporation, USA
Panel Coordinator: Calton Pu, Oregon Graduate Institute, USA
Tutorial Coordinator: Tamer Ozsu, University of Alberta, Canada

PROGRAM VICE CHAIRS

Data Management on the World Wide Web
Daniel Barbara, Bellcore, USA
Management of Uncertainty and Information Quality
Ami Motro, George Mason University, USA
Interoperability and Cooperative Information Systems
Dimitrios Georgakopoulos, GTE Laboratories, USA
Multimedia Systems and Digital Libraries
Kien Hua, University of Central Florida, USA
Parallel and Distributed Database Systems/ Mobile Computing
Masaru Kitsuregawa, University of Tokyo, Japan
Extensible and Active Database Systems
Stefano Ceri, Politecnico di Milano, Italy
Object-Oriented Database Systems
Elke A. Rundensteiner, Worcester Polytechnic Institute, USA
Temporal and Spatial Database Systems
Christian S. Jensen, Aalborg University, Denmark
Data Mining/Data Warehousing
Umesh Dayal, Hewlett Packard, USA
Transaction Processing and Real-Time Systems
Andreas Reuter, University of Stuttgart, Germany

EUROPEAN COORDINATOR

- Wolfgang Klas, University of Ulm, Germany

FAR EAST COORDINATOR

- Beng Chen Ooi, National University of Singapore, Singapore

LOCAL ARRANGEMENTS

- Kate Kinsley, Datawise, Inc., USA
- Ali Orooji, University of Central Florida, USA

FINANCE CHAIR

- Kun-lung Wu, IBM T. J. Watson Research Center, USA

PUBLICITY CHAIR

- George Karabatis, Bellcore, USA

REGISTRATION CHAIR

- Vijay Atluri, Rutgers University, USA

PUBLICATION CHAIR

- Karen Davis, University of Cincinnati, USA

PUBLISHER EXHIBITS CHAIR

- Ling Liu, University of Alberta, Canada

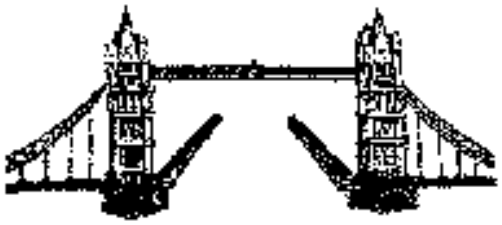
RESEARCH EXHIBITS CHAIR

- Sharma Chakravarthy, University of Florida, USA

Visit the Data Engineering Web Site at: http://gump.bellcore.com:8000/icde98/

BNCOD15 Call for Participation

Fifteenth British National Conference on Databases



London, England 7th - 9th July, 1997

The Conference. The British National Conference on Databases is an annual forum for database research, first being held in 1981. The conference provides an excellent opportunity for industrial and academic database researchers to discuss their requirements and ideas. BNCOD15 will be held at Birkbeck College, University of London.

Conference format. The conference will include research papers, industrial presentations, panels, poster and demonstration sessions, and an exhibition. Invited speakers include Prof. Larry Kerschberg, Director of the Center for Information Systems Integration and Evolution, George Mason University. Particular emphasis will be given in the conference to the topics of distribution and heterogeneity, advanced database applications and mining databases.

Social Activities. The conference dinner will be held at the Museum of London in the Lord Mayor's Coach and Medieval Galleries, which is situated by the Roman Walls of the old city. We anticipate the option of taking a river bus to the Museum, giving a view London's principal attractions (such as the Houses of Parliament, St Pauls Cathedral, the Globe Theatre, Tower Bridge and HMS Belfast) from the river.

The Location . Birkbeck College, founded in 1823 and incorporated by Royal Charter in 1926, is one of the multi-faculty Colleges of the University of London. The College is located in Malet Street, at the heart of the Bloomsbury area of Central London. The British Museum is within a few minutes walk of the College, and many other museums and art galleries are within walking distance. The area is also surrounded by numerous theatres, cinemas and restaurants, and is close to the excellent shopping facilities of the West End. The college is easily reached from Heathrow and Gatwick airports and the main London Railway stations.

Registration. Early registration closes: 31st May, 1997

For further information about the conference contact:

BNCOD15 Administrator
Department of Computer Science
Birkbeck College
Malet Street
London WC1E 7HX
England

Tel: +44 (0) 171 631 6722
Fax: +44 (0) 171 631 6727
EMail: bncod15@dcs.bbk.ac.uk
WWW: http://www.dcs.bbk.ac.uk/bncod15

Organisations and individuals wishing to exhibit at the conference should contact the Organising Committee chair.

R.G. Johnson	Organising Committee Chair
P.J.H. King	Programme Chair
C. Small	Proceedings
N.J. Martin	Treasurer
P. Douglas	Social

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398