Indexing OLAP data *] *Copyright 1996 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.* Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

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, attributes 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. However, 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.