A Dynamic Grid File for High-Dimensional Data Cube Storage and Range-Sum Querying

Wen-Chi Hou, Southern Illinois University, USA
Xiao Guang Yu, Southern Illinois University, USA
Chih-Fang Wang, Southern Illinois University, USA
Cheng Luo, Coppin State University, USA
Michael Wainer, Southern Illinois University, USA

ABSTRACT

In this article, the authors propose to use the grid file to store multi-dimensional data cubes and answer range-sum queries. The grid file is enhanced with a dynamic splitting mechanism to accommodate insertions of data. It overcomes the drawback of the traditional grid file in storing uneven data while enjoying its advantages of simplicity and efficiency. The space requirement grows linearly with the dimension of the data cube, compared with the exponential growth of conventional methods that store pre-computed aggregate values for range-sum queries. The update cost is $O(1)$, much faster than the pre-computed data cube approaches, which generally have exponential update cost. The grid file structure can also respond to range queries quickly. They compare it with an approach that uses the $R^*$-tree structure to store the data cube. The experimental results show that the proposed method performs favorably in file size, update speed, construction time, and query response time for both evenly and unevenly distributed data.

Keywords: Data Management, Data Structure, Multidimensional Database, Query Processing

INTRODUCTION

A data warehouse is a large collection of integrated data, built to assist knowledge workers, such as executives, managers, analysts, etc., to make better and faster decisions. It is often required that data be summarized at various levels of detail and on various attributes to allow knowledge workers to analyze the data through a variety of views in on-line analytical processing (OLAP). Typical OLAP applications include product performance and profitability, effectiveness of sales programs or marketing campaigns, sales forecasting, capacity planning, etc. Data warehousing and OLAP have
increasingly become a focus of the database industry.

OLAP systems generally support a multi-dimensional data model, which is also known as the data cube (Gray, 1997). Construction of a data cube is based on the set of selected attributes of the database. Certain attributes are chosen to be the measure attributes, i.e., attributes whose values are of interest, while some others are selected as dimension or functional attributes (Geffner, 1999). The values of the measure attributes are often aggregated according to the dimension attributes for analysis. The size of a data cube can be huge when the number of combinations of dimension attribute values is large.

The storage of data cubes is essential to OLAP. Much research (Agarwal, 1996; Beyer, 1999; Han, 2001; Morfonios, 2006; Xin, 2003; Zhao, 1997) has focused on the materialization of data cubes, that is, to pre-compute and store all possible combinations of multi-dimensional aggregates for fast multi-dimensional analysis. Some notable cube materialization algorithms proposed include ROLAP-based multi-dimensional aggregate computation (Agarwal, 1996, Morfonios, 2006), multi-way array aggregation (Beyer, 1999), BUC (Han, 2001), H-cubing (Xin, 2003), Star-cubing (Zhao, 1997), Minimal cubing (Li, 2004), etc. Since materializing data cubes are generally computationally intensive and space consuming, much effort has been devoted to reducing the computation and storage space of data cubes. These efforts include partial materialization of data cubes (Harinarayan, 1996), iceberg cube computation (Han, 2001; Xin, 2003; Zhao, 1997), computation of condensed, dwarf, and quotient cubes (Lakshmanan, 2002; Lakshmanan, 2003; Sismanis, 2002; Beyer, 1999; Wang, 2002), and computation of approximate cubes (Barbara, 1997; Cuzzocrea, 2006; Shanmugasundaram, 1999). While these pre-computed data cubes can be used to answer queries quickly, tremendous overhead is incurred in maintaining these pre-computed aggregate values as updates can propagate to a large number of relevant cells.

A range-sum query is used to compute the sum of the values of data cube cells that fall in the ranges specified by the query. It is very useful in finding trends and discovering relationships between attributes in the databases. Ho et al. (1997) proposed to compute prefix sums of data cube cells for range-sum queries. Although this method can respond to queries quickly, an update in the worst case can propagate to the entire prefix-sum cube. Therefore, it may not be suitable for data cubes that undergo frequent changes. Some efforts have been made to reduce the update propagations of prefix-sum cubes. Geffner et al. (1999) computed the relative prefix-sums to limit cascading updates to sub-cubes. More recently, they proposed (Geffner, 2000) to decompose the prefix-sum cubes recursively to control the cascading of updates. Although these measures can reduce the cost of updates to a certain degree, the cost can still increase exponentially with the number of dimensions of the data cubes. Chan et al. (1999) proposed a class of hierarchically organized prefix-sum cubes to reduce the update cost. However, the cost of update still increases exponentially with the number of dimensions. Gao et al. (2005) discussed efficient processing of range-sum queries over hierarchical cubes using parallel computing systems. In general, update propagation is a common problem for all pre-computed data cube approaches even though some improvements have been made. Note that all these approaches also require at least as much space as the original data cubes to store the prefix-sum cubes.

Instead of materializing pre-computed data cubes for range-sum queries, Hu et al. (2002) chose to store data cube cells in a slightly modified version of the R*-tree (Beckmann, 1990), called the DCA-tree. Updates to the data cube are accomplished by updating the corresponding points in the R*-tree.

Similar to Hu’s approach (Hu, 2002) we attempt to design a spatial data structure to facilitate data cube storage and range-sum query processing. We propose a dynamic grid file structure as a natural fit to the structure of data cubes. The data cube space is partitioned into
grid cells and each grid cell, which contains a number of data cube cells, is stored in a separate disk block. The proposed structure adapts itself to insertions of new data by splitting the grid cells dynamically. It minimizes the drawback of traditional grid files for uneven data while enjoying its advantages of simplicity and efficiency. The dynamic grid file facilitates range-sum query processing by identifying the grid cells that intersect with the queries quickly.

It is observed that given a fixed number of nonzero data cube cells, the space requirement for a grid file grows linearly with the dimensions of the data cubes, compared with the exponential growth of the prefix sum approaches (Chan, 1999; Geffner, 1999; Geffner, 2000; Ho, 1997). The update cost is $O(1)$, compared with the exponentially increased update cost for the prefix sum approaches. The grid file can also respond to range-sum queries quickly. For example, it takes around 1 second (elapsed) time to process a range-sum query over a 24-dimensional data cube, which should be very acceptable for OLAP. The proposed file structure is very efficient even for high-dimensional data cube applications.

Our approach is similar to Hu's (Hu, 2002) as we also use a spatial data structure to store the data cube. We conducted extensive experiments to compare our grid file with Hu's DCA-tree (or the $R^*$-tree) in terms of file size, construction time, update speed, query response time, etc. The experimental results show that the new dynamic grid file performs much better than Hu's in all these measures.

The rest of the article is organized as follows. Section 2 introduces related work. Section 3 presents the dynamic grid file structure and range-sum query evaluation algorithm. Section 4 reports the performance evaluation. Section 5 concludes this article.

RELATED WORK

There has been considerable research on data cubes, including computing data cubes (Agarwal, 1996; Gupta, 1997; Harinarayan, 1996; Johnson, 1997; Shukla, 1996), pre-computing subsets of a data cube (Gupta, 1997; Harinarayan, 1996), estimating the size of multidimensional aggregates (Johnson, 1997), and indexing pre-computed summaries (Shukla, 1996). In this article, we shall focus on those methods that process range-sum queries over data cubes.

The range-sum query is a useful tool for analysis. It sums up the measure attribute values of data cube cells that fall in the ranges specified by the query. The range-sum query can be very useful in finding trends and discovering relationships among attributes in databases. With the growing interest in database analysis, particularly in OLAP, efficient range-sum query processing is becoming an increasingly important issue in database research.

Prefix-sum cube (PC) (Ho, 1997) precomputes the prefix sums of data cube cells so that range-sum queries can be answered quickly. The value of a cell $P(x_1, x_2, \ldots, x_n)$ in the prefix sum cube is the sum of the values of data cube cells $C(y_1, y_2, \ldots, y_n)$, where $D$ is the dimension of the data cube, and $y_i \leq x_i$, $i = 1, \ldots, D$. The prefix-sum cube requires the same amount of storage space as the data cube itself. A range-sum query that specifies range constraints on $q$ of the $D$ dimensions can be answered in $2^q$ accesses, each for a corner cell of the $q$-dimensional hyper-rectangle defined by the $q$ ranges of the query. While this approach may respond to queries fast, its update cost can be prohibitive since a modification to a data cube cell can propagate to a large number of cells in a PC. It has been shown that the update cost of this prefix sum method is $O(n^p)$, where $D$ is the dimension of the data cube and $n$ is the size of each dimension. In the worst case, it could require rebuilding the entire prefix-sum cube. It is not suitable for situations where data undergo constant changes.

Several attempts have been made to reduce the cost of updates in prefix-sum cubes, however, at the price of increased range query complexity. Geffner, et al. (1999) presented the relative prefix-sum cube approach (RPC).
It partitions a data cube into a number of disjoint sub-cubes of equal size and calculates local prefix-sums for each sub-cube. It limits the cascading updates to individual prefix-sum sub-cubes, thereby reducing the update cost. The update cost is reduced to $O(m^p)$ in RPC, where $m$ is the number of partitions on each dimension of the sub-cubes. Since a query may cover an area that crosses sub-cube boundaries, RPC precomputes an additional overlay cube to convey information across the sub-cube boundaries. It has been shown that a range-sum query can be answered in no more than $(D+2)2^n$ accesses, where $D$ is the dimension of the data cube.

Chan et al. (Chan, 1999) decomposed the data cube space hierarchically. Prefix sums are calculated for cells based on their locations in the hierarchical structure. A data cube cell update can affect only the cells in the corresponding sub-cube at the same level. It has a worst case update cost of $O(n^p)$, where $D$ is the dimension of the data cube and $n$ is the size of each dimension, and a best case of $O(m^p)$, where $m$ is the size of each dimension of the sub-cube. A range-sum query is converted into a set of local range-sum queries and local prefix range-sum queries. A range sum query has a time complexity of $O(s^22^n)$, where $s$ is the number of sub-cubes at each level.

Hu et al. (2002) proposed the DCA-tree, which is a variant of the $R^*$-tree. Nonzero cells of a data cube are treated as points and stored in the DCA-tree in the same way as in the $R^*$-tree. Each minimum bounding rectangle/region (MBR) is associated with the aggregate value of the cells in it. It uses the $R^*$-tree to reduce overlaps among bounding rectangles when splitting. The DCA-tree reduces the update cost to $O(\log N_c)$, where $N_c$ is the number of changed cells.

A DYNAMIC GRID FILE FOR RANGE-SUM QUERIES

The majority of research work on fast processing of range-sum queries over data cubes (Chan, 1999; Geffner, 1999; Geffner, 2000; Ho, 1997) has chosen to store pre-computed aggregate values. While these approaches may be able to answer queries quickly, they are accompanied by serious drawbacks in terms of storage usage and update complexity of the pre-computed aggregate values. In general, the storage space and update complexity of these methods increase exponentially with the dimensionality of the data cubes, rendering these approaches infeasible for high-dimensional data cube applications. For example, consider a data cube of only 6 dimensions with each dimension having 100 values. The data cube would comprise at least $10^{22}$ data cube cells, which may already exceed the capacity of many modern disks.

The DCA-tree uses an $R^*$-tree to store the data cube. It stores only the nonzero cells of the data cube. An update to the data cube is accomplished by an update to the corresponding $R^*$-tree, and a range-sum query over the data cube is achieved by a spatial search in the $R^*$-tree.

In this article, we propose to use the grid file for high-dimensional data cube storage and range-sum querying.

The Dynamic Grid File Structure

The data cube is partitioned into hyper-rectangles, called data cube cells, and each cell is associated with a set of measure attribute values. The grid file (Nievergelt, 1984) is a natural fit to the storage of data cubes as it conforms to the structure of the data cubes nicely. The grid file structure we propose here has two parts: a grid file and a grid file directory. The former stores the nonzero data cube cells while the latter is an index into the grid file for fast accesses to the desired cells.

The Grid File

Let us first discuss the grid file. Assume each dimension is divided into $p_i \geq 1, 1 \leq i \leq D$, partitions, where $D$ is the number of dimensions of the data cube. The entire data cube space is thus partitioned into $G = \prod_{i=1}^{D} p_i$ multi-dimensional grid cells, of which each encloses
Figure 1. A dynamic grid file

A fixed number of data cube cells. Each grid cell \((j_1, j_2, \ldots, j_p)\), \(1 \leq j_i \leq p_i\), \(i = 1, 2, \ldots, D\), is intended to be stored in a disk page (block) in the grid file. Nonzero data cube cells are represented by their coordinates as points. The values of a data cube cell become the values of the corresponding point. Points that fall in the same grid cell are intended to be stored, if possible, on the same disk page in the grid file. The grid cell \((j_1, j_2, \ldots, j_p)\) is stored as the \(b^{th}\) page \((1 \leq b \leq \prod_{i=1}^{D} p_i)\) of the grid file based on the following formula:

\[
b = (j_1 - 1)p_2p_3\cdots p_p + (j_2 - 1)p_3\cdots p_p + \cdots + (j_{p-1} - 1)p_p + j_p
\]

Figure 1 shows a 2-dimensional data cube space that is divided into \(4^2 = 16\) equal sized grid cells. Each grid cell is intended to be stored as a physical block, numbered from 1 to 16, on the disk.

• Initial Grid File Structure

Let us discuss how to determine the number of grid cells \(G = \prod_{i=1}^{p} p_i\) into which the data cube space is to be partitioned. Let \(M\) be the number of measure attributes and \(D\) the number of the dimensional attributes. Let \(N\) be the initial number of nonzero data cube cells, which can be an estimate. Assume the storage space for each dimensional or measure attribute value is \(V\) bytes. Then, the estimated data set size \(S\) can be computed by the following formula:

\[
S = N(D + M) V.
\]

Extra space can be reserved by replacing \(S\) with a greater value for later insertions of data points.

Let \(\text{page\_size}\) be the size of a page on the disk, e.g., 8K bytes. \(S / \text{page\_size}\) is the size of the dataset in pages. The number of grid cells \(G\) ought to satisfy the following relationship:

\[
G = \prod_{i=1}^{D} p_i \geq S / \text{page\_size}.
\]

where \(p_i (\geq 1)\) is the number of partitions in the \(i^{th}\) dimension. If one wishes not to partition a dimension, say \(i\), then \(p_i\) is set to 1. Hereafter, we shall call the set of dimensions, whose \(p_i > 1\), the partition dimensions.

The selection of partition dimensions can have a great impact on the search. If a dimension is partitioned, the portions that satisfy the search criteria can be significantly reduced. The selection of partitioned dimensions should be performed with care.
range constraints on that dimension can be easily identified and the search can be effectively confined. Heuristics, such as selecting the most frequently referenced or important dimensions can be very useful as many queries can benefit from such partitions.

- Handling Overflow

When overflows happen in a grid cell, the cell is split into two subcells. The subcells can split themselves repeatedly as necessary. The splitting is performed along a selected dimension, here called a split dimension. Note that a split dimension is used to distribute overflow points in a (sub)cell while a partition dimension is used to confine search area of a query. Many heuristics, such as random selection, selecting the most spread dimension, etc., can be used in selecting the split dimension. However, unless the data exhibit extreme peculiarity in a (sub)cell, the difference among these splitting methods should be small. Therefore, for simplicity, we choose the split dimension in a round-robin fashion as it does not require memorizing the split dimension.

To distribute points in an overflow grid cell evenly to two sub-grid cells, the cell splits at the middle of the range of the points' values on the selected dimension. For example, in Figure 1, the grid cell C overflows. The points in C are (3.1, 1.9), (3.2, 1.75), (3.25, 1.4), (3.3, 1.09), (3.19, 1.11), (3.5, 1.12), and (3.6, 1.71). Assume the split dimension is the 2nd dimension. The set of the 2nd dimension values are {0.9, 1.11, 1.12, 1.71} and the middle of the range is computed as (minimum + maximum) / 2, which is (1.9 + 1.09) / 2 = 1.49. Thus, the cell C splits into two at 1.49 along the 2nd dimension as shown in the figure. The points (3.3, 1.09), (3.19, 1.11), (3.5, 1.12), and (3.25, 1.4) go to one sub-cell while (3.2, 1.75), (3.6, 1.71), and (3.1, 1.9) go to the other sub-cell. This scheme requires storing minimum and maximum values of the points on each dimension in a cell. Although we could have chosen the median or mean of the values to split, the middle range value is probably the simplest to compute as it does not need to examine the points in the cell. In addition, the minimum and maximum information on each dimension can be used for another purpose (discussed in Section 3.2).

**The Grid Directory**

A grid directory is stored to facilitate query processing. Each entry in the directory corresponds to a grid cell in the file. It records the sum of the values of the points in the grid cell and the minimum and maximum coordinate values along each dimension of these points (as just mentioned earlier), and a pointer to the disk page that physically stores the points of the grid cell. The minimums and maximums indeed represent the minimum bounding rectangle/region (MBR) of the points in the grid cell. This information is not only used in splitting cells, but also used in determining more precisely whether the points in a cell intersect with the query window or not.

In Figure 2, we show the structure of the directory. Initially, each entry in the directory

---

**Figure 2. The grid directory**

---

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
has a sum, a min and a max for each dimension, a split (value), and two pointers, ptr1 and ptr2, where ptr1 points to the disk page storing the grid cell (while ptr2 is not used initially). When a grid cell overflows and splits into two subcells, the directory entry spawns two subentries, one for each subcell. The split value and pointers to these two subentries are registered in the split, ptr1, and ptr2 of the original entry, respectively. The grid file can grow like a binary tree if overflows happen repeatedly in a (sub)cell, as shown in Figure 2. Note that only the ptr1 field of the leaf nodes of the binary tree points to a disk page in the grid file; other pointers in the non-leaf nodes point to nodes of the binary trees.

Each entry or subentry requires a small amount of space: 4 bytes for the sum, 4 bytes for the min and max on each dimension, and 4 bytes for each of the split, ptr1, and ptr2. That is, totally 16+8D bytes for each entry in the directory. The entire directory structure, including the spawned entries, is generally small. Therefore, in this research we load it into memory at initiation (before any querying and updates are posted). It can be observed that given an arbitrary data cube cell, by searching the in-memory directory, the physical disk page storing the data cube cell can be directly located in one disk access.

**Update of the Grid File**

As mentioned above, given a data cube cell, one can directly locate the page storing it through the grid directory. For an insertion, the page is first fetched into memory (in 1 disk access). If there is enough space left in the page, the point is added to it and the page is written back to the disk (in 1 disk access). If the page overflows, it is split into two. The level of the directory entry in the tree corresponding to the data cube cell determines the dimension to split (as it is selected in a round robin fashion), and the minimum (min) and maximum (max) coordinate values of the points in the cell on the selected dimension are used to split the points. The directory (sub)entry spawns two (sub)entries with each one pointing to a split page on disk. At the end, the two pages are written back to the disk (2 disk accesses) and the sum fields of the entries and their ancestors are updated accordingly to reflect the insertion. All in all, no more than three disk accesses are needed for an insertion. Figure 3 summarizes the operations.

As new points are added to the file, the average amount of unused space in a page decreases and the chance of splits increases. As splits increase, more and more space becomes available and the rate of splits decreases. This cycle repeats as points are added to the file.

**Figure 3. Insertion operation**

```
Insertion(point)
{  fetch the page containing the point into memory;
    if the page has enough space
      add the point to the page;
    else
      split the page into two;
        spawn two new directory entries, each pointing to a split page;
        compute the sum values for the two new entries;
    }
  write the page(s) to disk;
  update the sum fields of ancestor entries;
}
```
We shall empirically study this phenomenon in Section 4.

To delete a point, the page containing the point is first brought into memory as in the insertion operation. The point is then removed and the page is written back to the disk. Only two disk accesses are required. If a page becomes empty after deletions, it is released by adding it to an "unused page" list. The "unused pages" on the list can be reused later when a page splits requiring an additional page. The sum fields of the corresponding entry and its ancestors in the directory are updated accordingly.

Update is performed in a similar way by first fetching the page into memory, updating the value, and then writing it back to the disk (in two disk accesses). The sum fields of the entry and its ancestors are modified accordingly.

**Range-Sum Queries**

The grid file is originally designed for querying spatial data. Now we shall adapt it for range-sum queries.

When performing a range-sum query, the grid cells that intersect with the query window need to be identified. If a grid cell is completely contained in the query window, we can directly use the stored sum value for that grid cell in the grid directory. Otherwise, we have to access the grid cells on disk. For instance, the cells 10 and 11 in Figure 4 are fully contained in the window. Therefore, we can simply retrieve their sums from respective directory entries without physically accessing the disk blocks. If the cells are not fully contained in the range-sum window, like the cells 5, 6, 7, 8, 9, 12, 13, 14, 15, 16, they may have to be retrieved from the disk to check if there are any points in the cells falling in the window.

Recall that we have also stored the MBRs in the directory, i.e., the minimum and maximum coordinate values on each dimension of the data cube cells. One can further eliminate the need to access those grid cells, such as 12, 13, and 15, that do not intercept with the query window using the MBRs. As a result, only grid cells 5, 6, 7, 8, 9, 14, and 16 need to be retrieved from the disk. The detailed range-sum query algorithm is presented in Figure 5.

---

*Figure 4. Range-sum query*
Comparisons

Let us make some general comparisons between the prefix-sum methods (Chan, 1999; Geffner, 1999; Geffner, 2000; Ho, 1997) and the spatial data structure methods, including the DCA-tree (Hu, 2002) and our dynamic grid file. Let \( n \) be the size of each dimension and \( D \) the dimension of the data cube. The prefix-sum approaches need a space at least as large as the size of the data cube \( (n^D) \), while the spatial methods need only to store the nonzero cells of the data cube, which is no greater than the number of tuples in the data set. Since the space requirements of prefix-sum methods grow exponentially with the dimension, prefix-sum methods generally are not suitable for high dimensional data cube applications. On the other hand, spatial approaches have no such limitation.

The prefix-sum cube (PC) (Ho, 1997) has the fastest range-sum query processing time among the prefix-sum methods with \( 2^n \) accesses, where \( q (\leq D) \) is the number of dimensions on which the query poses range constraints. Recall that RPC (Geffner, 1999) requires \((D+1)2^n\) accesses while Chan’s hierarchical cubes HC (Chan, 1999) need \( s^22^n \) accesses. As for the spatial methods, they need to search the area specified by the \( q \) ranges. In general, the larger the \( q \) value, the smaller the area specified. When \( q \) is small, e.g., 3 or 4 dimensions, PC can be faster than the spatial methods because \( 2^n \) is small and the search area may be large for the spatial methods. When \( q \) is large, e.g., \( q > 8, 2^n \) becomes large while the search area becomes much smaller. Consequently, PC can be slower than the spatial methods. In fact, PC may not even be feasible for such medium dimensional cubes due to its huge storage requirements.

As for update, the prefix-sum methods are generally more complex and time consuming than the spatial methods. Even for the update efficient prefix-sum methods, such as RPC (Geffner, 1999; Geffner, 2000) and HC (Chan, 1999), they still need \( O(m^D) \) accesses, where \( m \) \((\leq n) \) is the size of the dimension of the sub-cube. In high dimensional cases, the update cost can be substantial. Recall that the grid file needs only 2 to 3 disk accesses, regardless of dimension.

As a short summary, the prefix-sum methods are generally suitable for only low-dimensional and static applications, while the spatial methods are suitable for a much wider range of dimensions, including medium and high dimensions, and for both dynamic and static applications.

PERFORMANCE EVALUATION

In this section, we report the experimental results of the dynamic grid file and the DCA-tree.
under a wide range of dimensions, from 4 to 40. The prefix-sum methods are excluded from the comparisons as they are not suitable for the medium and high dimensional experiments we performed. In fact, all these methods would require too much space, beyond the capacity of many, if not all, modern disks, for all the experiments except the one that has the lowest dimensions, i.e., 4.

We have implemented the dynamic grid file in C++ and compiled it with GNU C/C++ Complier V3.2.3. The test platform is Redhat Linux 3.2.3 running on a Dell Precision 360 workstation with a 3.3 GHZ CPU and 1GB RAM. Each dimension attribute is assumed to have a domain of [0, 99]. Both uniformly and nonuniformly distributed datasets are generated. Each dataset has 100,000 points (or 100,000 nonzero data cube cells). In the nonuniformly distributed datasets, points are distributed among 10 clusters as it is claimed that real-life data are often clustered (Dobra, 2002; Vitter, 1999). The centers of the clusters are randomly picked and the points are normally distributed around the centers with a standard deviation of 10% of the range along each dimension. The physical page size is set to 8,192 bytes to match the default logical disk block size of the operating system. We measure the number of disk accesses, as well as the CPU time, to compare the performance.

To divide the data cube space into grid cells, we determine the number of dimensions d to be partitioned and the number of partitions p in each selected dimension following Eq. (3) of Section 3. In order to partition as many dimensions as possible, we have chosen to partition each selected dimension into two, the smallest number of partitions (except for no partition). In Table 1, we list the d and p values for our dynamic grid files with 100,000 points in 4 to 40 dimensions.

In the following subsections, we present the experimental results on construction time, file size, CPU time, and number of disk accesses for both uniform and nonuniform datasets.

**Construction Time and File Size**

As stated earlier, the dynamic grid file is simple and easy to build and maintain. These advantages demonstrate themselves in the experiments, as shown in Figures 6 and 7. Due to the complex nature of the DCA-tree, its construction time and file size are substantially higher than those of the dynamic grid file. The DCA-tree’s construction time increases faster with the increase in dimension because the overlaps of MBRs exacerbate in higher dimensional spaces. Building a DCA-tree could take 10 times longer than building a dynamic grid file when the dimension is greater than 24 in the experiments, as shown in Figure 6.

The DCA-tree generally stores many pointers and rectangles in the data file. In addition, it also stores supporting files, such as the data-path directory and dir-path directory. Consequently, the DCA-tree used 1.72 to 3.21 times more space than the dynamic grid file as shown in Figure 7.

The construction time and file size for the nonuniform datasets are slightly higher than those for the uniform ones. The increases in construction time and file size in the dynamic grid file are due to the additional pages allocated for the overflow grid cells. Both approaches seem to handle nonuniform distributions very well.

<table>
<thead>
<tr>
<th>Table 1. Features of the dynamic grid file</th>
</tr>
</thead>
<tbody>
<tr>
<td>D - dimensionality</td>
</tr>
<tr>
<td>d - # of dimensions to be partitioned</td>
</tr>
<tr>
<td>p - # of partitions in the selected dimensions</td>
</tr>
</tbody>
</table>

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
Figure 6. Construction time

Insertion

After the file is loaded with the initial 100,000 points, an additional 5,000 points are inserted into the file to measure the cost of insertion. Here, we measure the number of disk accesses to the data file (Figures 8) and CPU time (Figure 9) for inserting a point. Note that all of the disk accesses here refer to accesses to data files, not including accesses to indices or directories, as they are generally small and can be loaded entirely into memory. Later, we will show (in Figure 10) how the file grows due to splitting when there are a large number of points inserted.

As discussed earlier, it takes one read to get the desired grid cell into memory and one or two writes (if a page splits) to store the page(s) back to the dynamic grid file. Since page splits do not occur very often in the experiments, the average number of disk accesses for inserting a point into the dynamic grid file is just slightly

Figure 7. File size

---

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
above 2, as shown in Figure 8. Since the MBRs of a DCA-tree overlap significantly, it may take more than one read to find a desired point and thus requires more disk accesses than the dynamic grid file. As the dimension increases, overlaps become more severe and more disk accesses are incurred, compared with the almost constant cost of the grid file.

Figure 9 shows the average CPU time for an insertion. The computation involved in an insertion to the dynamic grid file is simple and straightforward. The slight increase in CPU time in higher dimensions is due to the greater number of coordinate values needing to be checked. As mentioned above, the DCA-tree may need to examine more than one page due to the overlaps among MBRs. As a result, DCA-tree takes more time to insert a point than the grid file. As the dimension increases, overlaps become more severe and the cost of insertion increases more quickly in the DCA-tree.

In Figure 10, we show how the file grows through splitting by continuously inserting points into a 4-dimensional data cube. As points are inserted into the dynamic grid file, the amount of unused space in each page decreases, which increases the chance of splitting. Splitting increases the size of the file and leaves more room in the pages for new points. When space becomes abundant, the rate of splitting moderates. This cycle repeats. This phenomenon is more evident in the uniform data case than the nonuniform data case as pages tend to get full around the same time in the former case.

The DCA-tree does not reserve pages at the beginning. It grows by splitting. The entire splitting process is rather smooth and thus results in an almost linear increase in the file size. The file size is generally greater than ours as it stores additional information in the data file.

We have also measured the space utilization, that is, the average percentage of space being occupied by the data. In general, the dynamic grid file has space utilization ranging from 72% to 88% for the uniform data sets and 65% to 83% for the nonuniform data sets. These figures are obtained when the number of points grows from 200,000 to 500,000. As for the DCA-tree, the space utilization ranges from 74% to 82%. There is little difference in space utilization between uniform and nonuniform distributions for the DCA-tree.

**Range Queries**

**CPU Time**

In the experiments, we generated queries with two or four range constraints to see how the

---

**Figure 8. Disk accesses for an insertion**

![Graph showing disk accesses for an insertion]
complexity of queries can affect the performance. All of the two or four range constraints are assumed to be defined over the partition dimensions. Each range constraint covers 10% of the range of the respective dimension. Figure 11 shows the CPU time consumed by the dynamic grid file and the DCA-tree for up to 40 dimensions. It is observed that the DCA-tree took more CPU time than our dynamic grid file for both the uniform and nonuniform datasets. This is mainly due to the larger number of MBRs needing to be examined in the DCA-tree. The CPU time of the DCA-tree increases faster than that of the dynamic grid file as the dimension increases. This is because the number of MBRs intersecting with the query window in the DCA-tree increases faster than that for the dynamic grid file as the dimension increases.

The nonuniformity of the data distributions seems to have a greater impact on the DCA-tree.
than the dynamic grid file. This is because as more and more data points fall within close vicinity, more overlaps among MBRs occur and thus more time is needed to process queries in the DCA-tree. As for the dynamic grid file, the effect of data distributions is minimal. It shows that our method of splitting cells into nonoverlapping sub-cells is highly effective and our approach performs very well for both distributions.

Figures 11 (a) and (b) show the execution time of queries with two and four range constraints, respectively. The number of cells satisfying queries with two constraints (less restrictive) is greater than those with four constraints (more restrictive). As a result, the time spent in the former (Figure 11 (a)) is greater than that the latter (Figure 11 (b)).

Figure 11. CPU time for range-sum queries
Number of Disk Accesses

Figure 12 shows the number of disk accesses of the two approaches. Once again, the dynamic grid file performs much better than the DCA-tree. Similar to the CPU time, the disk access count of the dynamic grid file increases slower than that of the DCA-tree.

Greater numbers of disk accesses are needed for queries over nonuniform datasets for both approaches. The increase in the DCA-tree is due to the increased overlaps among MBRs. The slight increase in the grid file is due to the increased splits of the grid cells caused by the concentration of data points.

As explained above, as the number of constraints increases, the query becomes more selective, and the number of cells satisfying the query decreases. As a result, more disk accesses are required for the two-constraint queries than for the four-constraint ones.

Figure 12. Disk accesses for rang-sum queries
CONCLUSION

In this article, we propose the dynamic grid file for multi-dimensional data cube storage and range-sum query processing. While the conventional prefix sum data cubes can be used to answer range-sum queries quickly for low dimensional data cubes, they require tremendous amounts of space to store and considerable computations to update. The proposed enhancement to the traditional grid file has a natural appeal to the data cube storage as it conforms to the grid structure of the data cubes. It adapts itself to insertions of new data by splitting the grid cells as needed. It minimizes the drawback of the traditional grid files for uneven data while enjoying its advantages of simplicity and efficiency. The space requirement grows linearly with the dimension of the data cubes, compared with the exponential growth for the conventional prefix-sum data cubes. The experimental results show that the dynamic grid file outperforms the R*-tree based DCA-tree in all ranges of dimensions tested for both uniformly and nonuniformly distributed datasets. The proposed file structure can also answer range-sum queries quickly. We believe the dynamic grid file presents a promising solution to medium to high-dimensional OLAP applications.

REFERENCES


Han, J., Pei, J., Dong, G., & Wang, K. (2001). Efficient Computation of Iceberg Cubes with Complex Measures. *ACM SIGMOD Conference* (pp. 1-12).


Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
Lakshmanan, L., Pei, J., & Han, J. (2002). Quotient cube: How to summarize the semantics of a data cube. *VLDB conference* (pp. 766-777).


Wen-Chi Hou received the MS and PhD degrees in computer science and engineering from Case Western Reserve University, Cleveland Ohio, in 1985 and 1989, respectively. He is presently a professor of computer science at Southern Illinois University at Carbondale. His interests include statistical databases, mobile databases, XML databases, and data streams.

Xiaoguang Yu received her MS from Department of Computer Science at Southern Illinois University in 2005.

Chih-Fang Wang received his PhD in computer engineer at University of Florida in 1998. His current research interests include sequential, parallel and distributed algorithms, data structures, high performance computing, optical networks, quantum computing, DNA computing, bioinformatics, wireless/mobile security, data mining, and mobile agents and secure mobile agent platforms.

Cheng Luo received his PhD in computer science from department of computer science at Southern Illinois University, Carbondale, IL, USA in 2007. He is currently an assistant professor in the Department of Mathematics and Computer Science at Coppin State University, Baltimore MD. His research interests are in databases and data mining.

Michael Wainer obtained his PhD in computer and information science from the University of Alabama at Birmingham, in 1987 and is currently an associate professor of computer science at Southern Illinois University Carbondale. His research interests lie in the areas of software development, computer graphics and human computer interaction. He is particularly interested in interdisciplinary work which utilizes the computer as a tool for design and visualization.