pCube: Update-Efficient Online Aggregation with Progressive Feedback and Error Bounds

Mirek Riedewald Divyakant Agrawal Amr El Abbadi
Department of Computer Science, University of California, Santa Barbara
Email:{mirek, agrawal, amr}@cs.ucsb.edu

Abstract

Data Cubes are used in large data warehouses as a tool for online aggregation of information. Typically, online aggregation is supported by specifying a range query over a multidimensional data cube. As the number of dimensions increases, supporting efficient range queries as well as updates to the data cube becomes difficult. Another problem that arises with increased dimensionality is the sparseness of the data space. In this paper we develop a new data structure referred to as the pCube (data cube for progressive querying), to support efficient querying and updating of multidimensional data cubes in large data warehouses. pCube provides intermediate results with absolute error bounds (to allow trading accuracy for fast response time), efficient updates, scalability with increasing dimensionality, and pre-aggregation to support large range queries. We present both a general solution and an implementation of pCube and report the results of experimental evaluations.

Key Words: OLAP, Data Warehousing, Online Aggregation, Error Bounds, Efficient Updates

1 Introduction

Data warehouses typically accumulate and store operational data from a collection of OLTP (On-Line Transaction Processing) databases. Statistical and scientific databases (e.g., geographical information systems, census databases) contain huge amounts of "raw" information. Human analysts are interested in extracting that part of the data that is of interest for a certain task, e.g. discovering population development trends or analyzing the success of a campaign of TV commercials. That data is described by a set of attributes. One of them is chosen as the *measure* attribute (the attribute of interest, e.g. number of people). The others are the functional attributes, or simply *dimensions* (e.g. age, income, etc.). The data can conceptually be viewed as a hyper-rectangle, where each single *cell* is described by a unique combination of dimension values and contains the corresponding value of the measure attribute. We will henceforth refer to this hyper-rectangle as the *data cube*. We should point out that in this paper the data cube model only represents a way of *viewing* the data. It does not imply or exclude a certain physical data organization, like in relational tables (ROLAP) or in multidimensional databases (MOLAP).

The more complex dependencies an analyst wants to discover, the more functional attributes have to be included into the analysis. This obviously leads to a high dimensionality of the data cube. It is, however, important to note that even though the data itself is massive, the high dimensionality causes the data cube to be very *sparse*. For example, a U.S. census data cube that measures the number of U.S. citizens for the dimensions in Table 1 contains more than $2 * 10^{14}$ cells. Assuming that the U.S. census bureau stores data about 200 million Americans, at most 0.0001% of the cells will be non-empty. Assuming further, that each value of the measure attribute (in the example the number of people) can be encoded with a single byte, storing all cells will require more than 180 Terabytes.

Attribute	**Range**	**Attribute**	**Range**
Age	0 - 150	Income	0 - 99999
Weight	1 - 500	Race	1 - 5
Family type	1 - 5	Marital status	1 - 7
Class of worker	0 - 8	Education	1 - 17

Table 1: Attributes and their ranges for a census data set

An important tool for analyzing data is a range query. Range queries apply aggregate functions (like the SQL functions SUM, COUNT, AVG) to data cube cells that are selected by the given ranges on the dimensions. A range query could ask to "Return the number of U.S. citizens that are younger than 30 years and earn more than $50,000 per year". In the data cube model a range query selects a hyper-rectangle of cells. We will use the term *query-cube* to refer to this hyper-rectangle. To answer a range query, the values stored in the cells inside the query-cube must be known. For large ranges this can lead to a situation where answering the range query requires a considerable number of accesses to slow secondary storage. Nevertheless Online Analytical Processing (OLAP) is supposed to be an interactive process where analysts expect *fast responses* in the order of seconds at most.

OLAP applications typically have to deal with updates, e.g., data warehouses collect constantly changing data from a company's databases. Often it is regarded to be sufficient if those updates can be efficiently processed in batches. There are, however, applications that require *fast updates* to single cells. For instance, business leaders will want to construct interactive *what-if* scenarios using their data cubes, in much the same way they construct *what-if* scenarios using spreadsheets now. These applications require real-time (or even hypothetical) data to be integrated with historic data for the purpose of instantaneous analysis and subsequent action. In a system where updates are only propagated in batches (possibly during the night), such *what-if* analysis will not be possible. Also, while some applications do not suffer in the presence of stale data, in many emerging applications like decision support and stock trading the instant availability of the latest information plays a crucial role. Finally, the greater the number of updates the longer batch updates will make the data in the data cube inaccessible to an analyst. Finding suitable time slots for batch update windows becomes increasingly harder when businesses demand flexible work hours and 24 hour availability of their important data. Especially data collections that are accessible from all over the world (e.g., for multinational companies) do not follow a simple "high load at daytime, no accesses at nighttime" pattern. Consequently even systems that apply updates in large batches benefit from reduced update costs which shrink the size of the update window and enable the system to process more updates in the same time.

In this paper we develop the concept of a *progressive data cube*, abbreviated as *pCube*. The main idea is to build a data structure that is able to provide the user with fast approximate answers for range queries. The answer can then be progressively refined. Since an approximate answer without quality guarantees is of no use, pCube provides *absolute* error bounds for the query result. Compared to probabilistic error bounds (confidence intervals) absolute bounds offer the following advantages. First, probabilistic bounds do not guarantee that the final result will be inside the confidence interval; especially for small answer sets with high variance. Second, while non-expert users can easily understand the concept of absolute error bounds, this is not necessarily the case with confidence intervals. This argument is very important for data exploration tools that are designed for a diverse group of users.

To be more specific, pCube has the following properties.

- pCube can deal with high-dimensional sparse data cubes as well as with dense data cubes. Its access, update and resource costs depend on the number of non-empty cells, not on the size of the complete data cube (the product of the domain sizes of all dimensions).
- pCube's feedback mechanism provides the user with quick responses to range queries. A response consists of an approximate answer with absolute error bounds which are progressively refined.
- The feedback mechanism enables the user
  - to interrupt the query execution as soon as the result is good enough, or
  - to set a time limit for the query execution and get the best possible result according to this limit (with the possibility to further refine the result by allowing additional execution time).
- pCube combines pre-aggregation and hierarchical data organization to speed up query execution and enable fast updates. It does not depend on batch updates. On the other hand, pCube benefits from applying updates in batches. The batch size therefore can be used as a tuning parameter.

For popular aggregate functions like SUM (sum of all non-empty cells in a query range), COUNT (number of tuples that fall into a query range) and others, we state explicitly which values have to be stored to enable an update-efficient approximation.

The rest of the paper is organized as follows. Section 2 gives an overview of relevant related work. In Section 3 we develop the general pCube concept. Experimental results for a pCube implementation are reported in Section 4. Section 5 concludes this article.

2 Previous Work

To support the process of extracting and analyzing data from a data collection, [13] proposes a new SQL operator — "CUBE" or "data cube". It is a d-dimensional generalization of the GROUP BY operator and computes all possible group-bys for d attributes (given in the SELECT clause). We will refer to the result of the CUBE operator as the *extended data cube*. For a selection of d attributes, there exist 2^d possible group-bys. Due to this exponential number of results, computing the complete extended data cube efficiently is one of the major challenges (see [4] for an overview and references). Especially for high-dimensional and sparse data sets computing and storing the complete extended data cube is infeasible ([22] describes the problem of data explosion). Updates to a single tuple in one of the source data sets would cause an aggregate in each of the 2^d views to be updated as well. To overcome the above problems, efficient algorithms were proposed to choose those views (and indexes) to be stored that optimize a certain cost/benefit metric (e.g., [15] [14]). [19] and [20] describe index structures especially designed for the extended data cube. The above techniques do not provide fast responses for all aggregation queries.

In [18] Ho et al. develop techniques to guarantee an access time for range SUM queries that is independent of the size of the query-cube (constant cost) by pre-aggregating information in the data cube. Their prefix-sum approach suffers from high update costs (single cell update cost is linear in the total number of cells in the data cube). [10], [6] and [9] address this issue by balancing the cost of queries and updates. None of the above approaches can, however, efficiently deal with high-dimensional sparse data sets. In fact, applying the proposed algorithms to sparse data sets leads to dense sets of the size of the complete data cube, including empty cells. Suggested techniques for dealing with sparse data sets are rather sketchy and not effective for arbitrary data distributions.

The main idea behind most approaches for handling sparseness is to find dense chunks or clusters in the data set. Then dense and sparse regions of the data set are each handled by a specialized technique. Usually these approaches try to combine the advantages of special "dense" and "sparse" storage and access techniques, e.g., [12] [7]. Techniques for sparse data sets mainly focus on space efficiency. Aspects like early feedback for queries and update-efficiency are not examined.

An established technique to support accesses to databases is to create index structures on them. [8] and [21] provide an excellent overview of the field. In [2] it is suggested to augment indexes by aggregate data to exploit summary information for a data set. This idea up to now was used to support query cost estimation and approximate answers to queries [26], but to the best of our knowledge never for early feedback with absolute error bounds on data cubes. Also, in contrast to our approach, the solution presented in [26] does not allow for efficient updates on single cells. Another feature that distinguishes pCube from index structures is that it is primarily a way of *organizing and storing* data for aggregation and summarization. The indexing comes for "free" by exploiting the data organization.

The New Jersey Data Reduction Report [2] provides an overview of various techniques that can be useful for online aggregation. Barbará and Wu [3] describe a technique to use loglinear models for compressing the data cube and obtaining approximate answers. For the technique to be efficient, dense clusters in the data cube have to be identified and are then approximated by loglinear models (other approximation models could be used as well). Returning error bounds is not explicitly discussed in the paper. It is possible to provide absolute bounds based on the approximation model and a guaranteed bound for the approximation error per cell. However, for queries that have a large intersection with the approximated chunk (but do not contain it completely) the error bounds will be very loose. Maintaining the approximation models in the presence of updates requires batch updates that access all cells in a chunk. [24] presents a compression technique that achieves very high compression ratios (1000 and more) by using density estimations for dense clusters. This approach, however, is not efficient in the presence of updates and sparse data sets. With increasing sparseness, the estimation will become much less accurate since the result is obtained by multiple integrations over the density functions. Also, the technique can not provide error bounds for its approximate answers and a query has to access the data sources if the user requests an exact result. Compressing sparse data cubes using wavelets is described in [25]. Depending on the number of wavelet coefficients accessed, the result can be more or less accurate depending on the user's demand. But since no error bounds are returned to the user, he/she can not make use of this possibility. To get an exact answer, the data sources have to be accessed, because the wavelet decomposition algorithm drops "unimportant" coefficients. Even though wavelets provide good approximations for most practical applications, they do not fulfill our requirements for error bounds for approximations and efficient updates. Another direction of research aims at using histograms for fast approximate answers [23]. Histograms are an efficient technique to summarize data. Systems like AQUA [1] use small pre-computed statistics, called synopses [11] (which usually are histograms and samples), to provide fast approximate answers. Synopses can be used to answer range queries. Since they approximate data distributions, it is not possible to get the exact answer to a query without accessing additional data (in the data sources). The AQUA system, for instance, provides fast answers with probabilistic error/confidence bounds [1] (no absolute bounds). Up to a certain degree the user can theoretically trade off accuracy for faster responses by allowing smaller samples to be used. The accuracy of the query response is limited by the information stored in the histograms and samples. For further refinements (up to the exact answer) the query has to access the data in the sources, which is very expensive. Techniques to efficiently maintain synopses typically require batch updates, which makes the synopses inaccessible during the update. pCube on the other hand can easily deal with concurrent single cell updates and queries by using locks on the tree nodes.

Recent developments show that even though the issue of fast approximate answering has gained increasing attention, there are still not many solutions that provide progressive refinement of the approximate result with error bounds. An exception is the CONTROL project [16]. Users are provided with *probabilistic* error bounds that are progressively refined. CONTROL is based on algorithms especially designed for online aggregation (e.g., [17]). In contrast to the CONTROL approach, pCube returns *absolute* error bounds. It is more specialized to efficiently support range queries by pre-computation of aggregate values for very large data cubes.

3 The pCube Concept

For large data collections accessing and aggregating the data that is relevant for a query can consume time in the order of minutes. Take for example a range query that asks for the number of Americans that are younger than 30 years. Obtaining the answer from the "raw" data requires accessing at least all those tuples (i.e., data cube cells) that satisfy the selection criterion (which could be some millions of tuples), not counting additional index accesses. In general, for an *exact* answer the value of each single cell in the query-cube must be found out and included in the aggregation on the fly. Index structures are of little help when a query-cube contains a large number of non-empty cells. The alternative way to speed up the query process is by pre-computing the aggregation values for groups of cells and to use those results when answering a query. For instance, if the number of Americans between 0 and 14 and the number of Americans between 15 and 29 years of age is known, i.e., was pre-computed and stored, then two accesses to these pre-computed values provide the answer, thus saving a large number of disk accesses and CPU time. Pre-computation (pre-aggregation) leads to the following tradeoffs. The smaller the targeted response time for a query and the more queries are to be sped up, the more storage is necessary for pre-computed values. On the other hand, the more pre-computed values depend on a certain cell, the more expensive it is to update this cell's value (since the pre-computed values have to be updated as well). High dimensionality and sparseness of the data cube affect the benefit-cost ratio of these tradeoffs negatively. More dimensions dramatically increase the number of possible range queries and storing pre-computed values for sparse data cubes leads to the problem of data explosion [22]. Approaches which are effective for low-dimensional and dense data cubes therefore are not feasible for high-dimensional sparse data cubes.

3.1 Requirements

The only way to guarantee fast responses for any range query on high-dimensional sparse and massive data cubes is to weaken the requirement to return exact answers. Instead, pCube returns *approximate* answers with absolute error bounds. To get a quick response, the content of the query-cube has to be approximated with a small number of parameters. Different queries and applications make different demands on the approximation technique.

(1) When the query-cube is large and contains a large number of non-empty cells, it can only be approximated quickly if the approximation model's granularity is coarse. Queries with small query-cubes on the other hand are best approximated by choosing a finer approximation model. Popular approximation techniques like (multidimensional) histograms only offer a single granularity to all queries, i.e., favour either large or small query-cubes.

(2) Applications require different degrees of accuracy. The degree of accuracy should be chosen by the user, not by the system (designer). In some cases the required accuracy is not determined *a priori* (e.g., when it suffices to find out if the result is a positive number, a result range $100 \leq x \leq 10000$ is acceptable, but the range $-1 \leq x \leq 10$ is not).

(3) The requirement of efficient updates rules out techniques where an update to a single cell can cause expensive recomputations (e.g., with costs in the order of the data cube size or the number of non-empty cells in the data cube).[1]

3.2 The pCube Structure

In the following a pCube structure that fulfills the above requirements is described. pCube is organized as a tree. Each node covers a region of the data cube. We will use the term *node-region* to denote

[1]In Section 2 some of those approaches were discussed.

this region (the node-region could be a hyper-rectangle, hyper-sphere, etc.). The root node covers the complete data cube (or any region that contains all non-empty cells in the data cube). Leaf nodes cover single *non-empty* cells. Any other node covers a part of the node-region of its parent, such that all children together completely cover their parent's node-region (or at least all non-empty cells covered by the parent).[2] Each leaf node stores the value of the cell it covers. Internal nodes store information about the cells in their node-region, which will henceforth be referred to as *node-information*. For each aggregate function supported by pCube, the node-information contains the pre-computed aggregate value for the cells in the node-region or values that enable the computation of the aggregate value in constant time. Additional parameters are stored that allow for the fast computation of absolute error bounds for each of the supported aggregate functions. To guarantee space and update efficiency, the size of the node-information is bounded by a (*a priori* known) constant.

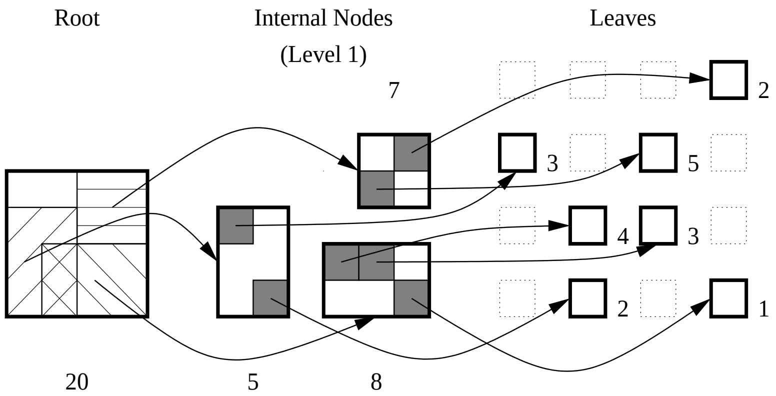

Figure 1: pCube with R-tree-like structure for the aggregate function COUNT

Figure 1 shows an instance of pCube that supports the aggregate function COUNT. For a combination of dimension values the measure attribute stores the number of tuples with this value combination. COUNT range queries return the number of tuples that fall into a certain range. The node-information simply consists of the sum of the values of the non-empty cells in the node-region. Since there are overlapping node-regions of sibling nodes, it has to be ensured that no tuple is counted more than once. Non-empty cells that fall into more than one node-region of sibling nodes have to be assigned to only one of the nodes.

3.3 Querying pCube

A query is answered by descending the tree. The further the query descends, the smaller the node-regions, i.e., the better the approximation based on the node-information. Each time the query descends to a new node, a finer intermediate result (with better error bounds) can be output. Thus the user receives continuous feedback of improving quality. Both small and large query-cubes can be supported efficiently. For large query-cubes the coarse granularity in the beginning allows for fast results which quickly improve. Small query-cubes typically intersect a smaller number of tree nodes, and therefore the query quickly reaches lower level nodes that provide more accurate answers. The pre-computed aggregate value at a tree node can dramatically reduce the response time. If a query-cube completely contains a node-region, this value represents the *exact* aggregate value for all query-cube cells in the node-region. Consequently, no descendents of the node need to be accessed and the aggregate function does not need to be computed on the fly. In our census example for the aggregate function COUNT, a tree node whose node-region contains all cells that satisfy $10 \leq \text{age} < 30$ and $0 \leq \text{income} \leq 30000$ (all other attributes range over their complete domain) stores the number of people that fall into this range in its node-information. If, for instance, a query asks for the number of people that are younger than 30 and earn less than \$40,000, the pre-computed value in the node contributes the exact answer for all

[2]Note that there might be non-empty intersections, e.g. overlapping hyper-rectangles for structures similar to the R-tree.

people between 10 and 30 that earn less than $30,000. Thus, a single access to the node provides an exact answer for a large number of cells, without requiring to descend further in that subtree.

The cost of a query (measured as the number of disk/memory accesses, or response time) in the worst case depends linearly on the number of node-regions the query-cube intersects. This cost decreases if node-regions are completely included in the query-cube and if the user is satisfied with approximate results.

3.4 Obtaining Absolute Error Bounds

The computation of the error bounds is based on the following observation. When query-cube and node-region have no cells in common, the answer is empty (which is an exact answer). If the query-cube contains the complete node-region, the aggregate value in the node already provides the exact answer for those cells (no approximation error at all). In the non-trivial case when the query-cube contains only some of the node-region's cells, error bounds are obtained in constant time by only knowing the node-information and properties of the aggregate function. The node can either store bounds on the extreme values or store enough information to allow their fast computation. For the COUNT function a node only needs to store the sum of the values of the non-empty cells (which contain the number of tuples with the corresponding dimension values) in its node-region. This number defines the upper bound, while 0 is the trivial lower bound. Table 2 contains selected aggregate functions (including the SQL standard aggregate functions) together with auxiliary values (explained in Table 3) for the computation of absolute error bounds. The last column contains the formula to compute the aggregate value for the non-empty cells in the node region. Note that, except for MIN and MAX, the value that aggregates all non-empty cells in the node-region can be computed from the stored values that are used for the computation of the bounds.

Function	**Auxiliary values**	**Upper**	**Lower**	**Aggregate for node-region**
COUNT	sum	sum	0	sum
SUM	sum_+, sum_-	sum_+	sum_-	$\mathrm{sum}_+ + \mathrm{sum}_-$
AVG (average)	cnt, sum_+, sum_-	sum_+	sum_-	$(\mathrm{sum}_+ + \mathrm{sum}_-)/\mathrm{cnt}$
MAX	sum_+, sum_-	sum_+	sum_-	Additional value required
MIN	sum_+, sum_-	sum_+	sum_-	Additional value required
Variance	cnt, sum_+, sum_-, sum_2	sum_2	0	$\mathrm{sum}_2/\mathrm{cnt} - ((\mathrm{sum}_+ + \mathrm{sum}_-)/\mathrm{cnt})^2$
Standard deviation	cnt, sum_+, sum_-, sum_2	$\sqrt{\mathrm{sum}_2}$	0	$\sqrt{\mathrm{sum}_2/\mathrm{cnt} - ((\mathrm{sum}_+ + \mathrm{sum}_-)/\mathrm{cnt})^2}$

Table 2: Values that provide absolute error bounds for aggregate functions

Symbols	**Meaning**
sum	Sum of the values of all non-empty cells in the node-region
sum_2	Sum of the squares of the values of all non-empty cells in the node-region
sum_+	Sum of the positive values of all non-empty cells in the node-region
sum_-	Sum of the negative values of all non-empty cells in the node-region
cnt	Number of non-empty cells in the node-region

Table 3: Node-information values and their meaning

Note that the values in Table 3 can be updated in constant time when the former and the new value of an updated cell in the node-region is known. In some cases more sophisticated techniques might lead to better approximations, at the cost of increased storage requirements and more expensive computations (or updates). Essentially the choice of which kind of information to store at a tree node depends on the application (which types of range queries are supported, how sensitive the query result is to approximation errors).

Figure 2 illustrates the technique for the aggregate function COUNT for a non-overlapping tree structure. The shaded region indicates the intersected region between node-region and query-cube for all nodes that have to be accessed to get the exact result. Note that the node-region of one of the internal nodes is completely included in the query-cube, therefore the corresponding leaf nodes do not need to be accessed, even though their node-regions are covered by the query-cube. Each node stores the sum of the values of the non-empty cells it covers. In case the query-cube completely contains the node-region, upper and lower bounds for the result are equal to the count value stored in the node. Otherwise, when query-cube and node-region intersect, at least none and at most all non-empty cells of the node-region can fall into the intersection, providing the error bounds. Even though the approximation is very coarse at the upper nodes near the root, it is refined quickly when the query descends the tree. In the example the approximate value is obtained based on the assumption that the non-empty cells are uniformly distributed over the node-region. Thus the query result is approximated as 9.4 at the root and as 10.5 at the next level of nodes. Finally, after accessing the values in the leaves the exact result of 12 is returned.

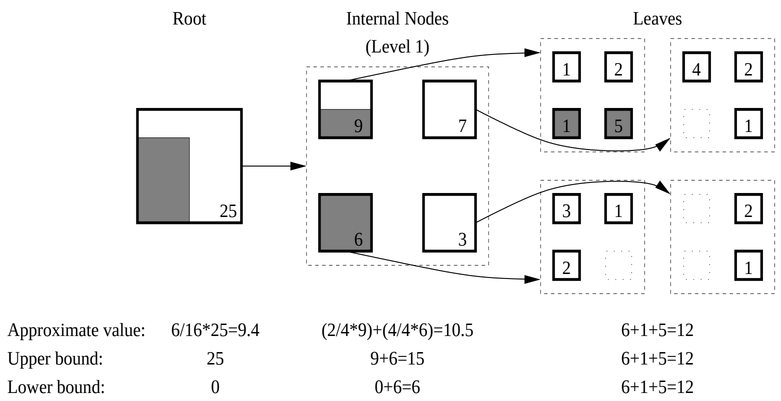

Figure 2: Obtaining error bounds for a pCube with quadtree-like structure

3.5 Updating pCube

We define an *update* to be any operation that changes the value of a data cube cell (e.g., an insert is an update that changes a cell's value from "empty" to a certain value; a delete is defined similarly). An update to a cell affects the node-information of all nodes that cover this cell. An update takes advantage of the tree structure by first descending the tree until the corresponding leaf node is reached. The leaf's value is changed. Then, knowing the former and the new value of the updated cell, all affected nodes on the path from the root to the leaf are updated (bottom-up). The overall update cost depends not only on the number of nodes that are affected, but also on the cost of updating a node's information about the cells in its node-region. Ideally the node-information can be updated in constant time without additional memory/disk accesses. Then, for trees with non-overlapping node-regions of sibling nodes, the update cost is linear in the height of the tree. Otherwise, in the worst case all internal tree nodes might have to be accessed. Also, updates can lead to node/page overflow or underflow. How those cases are handled depends on the specific tree structure used to implement the pCube. Basically techniques similar to those known from index structures can be applied. Figure 3 illustrates the insertion of a new value into a pCube; deleting the cell would reverse the changes.

3.6 Efficient Updates and Fast Aggregation

The way pCube's nodes divide the space determines pCube's structure and therefore heavily influences the query and update costs. In general, techniques have to be chosen that guarantee that both costs depend on the number of *non-empty* cells, rather than on the dimensionality or the size of the data cube (which includes empty cells). Possible choices include structures that are similar to the R-tree, X-tree, kdB-tree, etc. (see [8] for references).

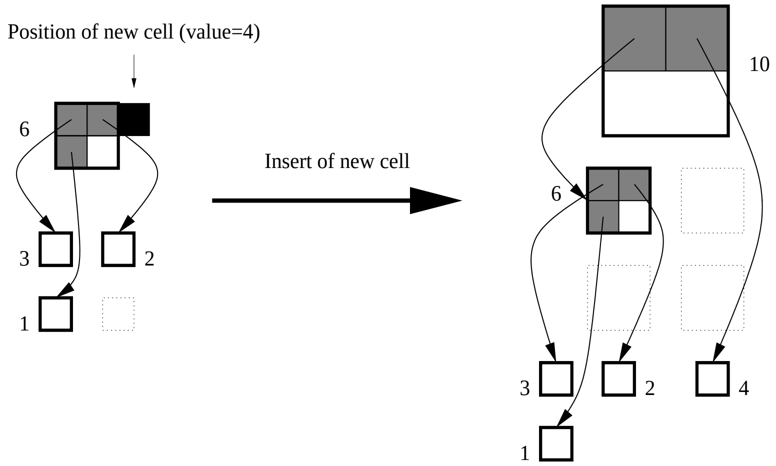

Figure 3: Effects of an update on a pCube with quadtree-like structure

Generally pCube can be constructed for any aggregate function. There are, however, aggregate functions that can not be supported efficiently. The main problem is caused by nodes on the upper tree levels (close to the root) that aggregate a large number of values. We require the cost of recomputing the aggregate to be independent of the number of aggregated values, i.e., to be bounded by a constant. It should be possible to compute the new aggregate value only from the knowledge about the update operation (previous contents of the updated data cube cell, new contents of that cell) and a small amount of auxiliary information about the values. Table 2 contains some popular aggregate functions and formulas for obtaining absolute error bounds and aggregate values from auxiliary values that can be efficiently incrementally maintained (see Table 3).

COUNT can be supported efficiently by storing the number of tuples that are in the node-region. For the AVG (average) this direct approach fails. Knowing the former average for the cells in the node-region and the update on a single cell does not provide enough information for computing the new average. However, if instead the sum and the number of values are known, their average can easily be computed and incrementally updated.

MAX and MIN can not be maintained efficiently if the current MAX/MIN value is affected by the update. It is, however possible to obtain absolute error bounds for the approximation of both functions. The required values (sum_+ and sum_-, see Table 3) can be efficiently maintained. Aggregate functions like MostFrequentValue can not be efficiently supported by pCube.

4 Experimental Results

To evaluate the pCube concept we performed extensive experimental evaluations. We start with a short description of the pCube implementation and the data sets used, and then discuss selected experimental results.

4.1 The pCube Implementation

The concept of pCubes is very general, and leaves much freedom for an implementation to be optimized for a particular application. The only assumption we make for this implementation is that the values of the dimensions and the measure attribute can be expressed with data types of a small bounded size, e.g., integer, long, float, double (e.g., 32 or 64 bits, respectively). Most attributes of real-world data collections have this property. For attributes with large data types, like strings, functions can often be found that compress the attribute's range.

The structure used for this pCube implementation is based on spatial decomposition. The children of a node partition the node-region of their parent into non-overlapping regions of equal size by dividing each dimension in half. Among all possible tree nodes, we materialize in the tree only those nodes that contain non-empty cells and have two or more children containing non-empty cells. Figure 4 shows an example. Thus it is guaranteed that the number of internal tree nodes is less than the number of leaves. Since the number of leaves equals the number of non-empty cells in the data cube, our pCube implementation's *space requirement* depends linearly on the number on non-empty cells.

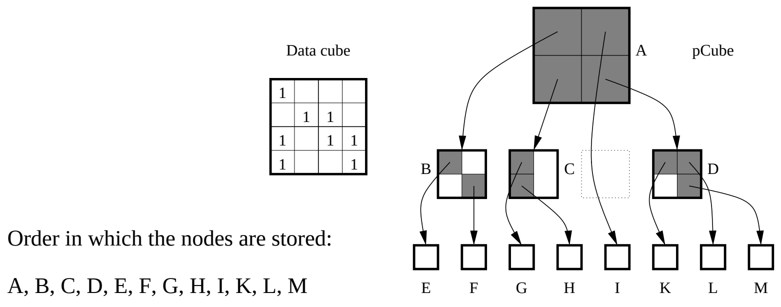

Figure 4: Example for the pCube structure

Depending on which aggregate function pCube supports, the nodes store values that summarize the information in their node-region (see Tables 2 and 3). For instance, to support the aggregate function SUM, each internal node stores the sum of the positive and the sum of the negative values of its node-region.

Queries are performed as described for pCube in general. The cost of a query in the worst case depends linearly on the number of node-regions that intersect the query-cube. Consequently the *cost of a query* is in the worst case linear in the number of non-empty cells. On the average, however, better behavior can be expected because queries can take advantage of pre-computed values.

To update a cell, first the tree is descended to find the region where the cell is located. If the update inserts a new cell or deletes a cell, it could be necessary to insert a new node or shrink/delete a node, respectively. These operations, however, due to the spatial decomposition approach, do not propagate for more than one level of the tree. After changing the cell's value, the node-information is updated for all nodes on the path from the cell up to the root. For efficiently updatable functions (e.g. COUNT, SUM, etc.) this results in an overall worst case update cost that is linear in the height of the tree. Since we assumed that the data types of the attributes are of bounded size, the spatial decomposition guarantees that the height of the tree is also bounded by a constant. Consider an example where no data type uses more than 32 bits. At each level of the spatial decomposition each dimension is divided in half. Hence the height of the tree is bounded by 32.[3] Thus the overall *cost of an update* is bounded by a constant.

For large data sets the complete pCube structure will not fit into a computer's main memory and has to be stored on disk (secondary memory). We store pCube in a packed format onto disk pages where a page can contain more than one node. Approximate results with error bounds could be output after accessing all relevant nodes in a page, while the next page is retrieved from disk.

The nodes on the disk pages are ordered by their level (i.e., how often the space was divided), nodes on the same level are stored in Z-order (space-filling curve, see for instance [8]). See Figure 4 for an example. Thus for queries that need to access a high percentage of the internal nodes, the corresponding pages can efficiently be accessed with sequential I/O. By keeping the first pages that contain the high-level (close to the root) nodes in fast storage (cache/memory) pCube can quickly return results without accessing the disk. Note, that maintaining the node order on the disk pages results in extra costs for insert and delete operations. Whenever an insert creates a new node or a delete causes a node to be shrunken, the correct page for inserting the node must be located. By indexing the page numbers with a B-tree and using the "greatest" node (according to the ordering of the nodes) in the page as the index attribute, the page can be found at an extra cost which is logarithmic in the number of non-empty cells.

[3]For floating point data types that encode a large range with a comparably small number of bits, the floating point numbers are mapped to corresponding bit strings (order-preserving). Then the spatial decomposition and the accesses are performed based on the "encoded" numbers.

4.2 Results

We ran various experiments for real-world and artificial data. Artificial data sets included dense and sparse data cubes. We examined the effect of skew (Zipf distributed values [27]), clusters and dimensions with different domain sizes.

The results presented here were obtained by executing 30,000 random range queries of different selectivity on the data cubes. In each dimension, a range was independently and uniformly selected based on selection predicates. Let $[a \dots b]$ denote the set $\{a; a+1; \dots; b\}$ and let dimension D's domain be $[\min_D \dots \max_D]$. Then the following ranges were selected for dimension D (where x and y are the randomly determined bounds):

1. $[\min_D \dots x]$ with 30% probability
2. $[x \dots y]$ with 10% probability
3. $[x \dots x]$ (single value) with 10% probability
4. $[\min_D \dots \max_D]$ (complete domain) with 50% probability

Note, that this selection favors queries that are disadvantageous for our pCube implementation. It is very likely that range queries are generated that cut through many dimensions, i.e., that intersect a large number of tree nodes, without completely covering their node-regions. Furthermore, in many high-dimension applications, it is realistic to assume users to specify a range over a limited number of dimensions and leave other dimension attributes unspecified.

In each experiment we measured the number of page accesses for obtaining results of different approximation qualities for queries with a selectivity between 0% and 100%. Those numbers are compared to two benchmark algorithms. For our experiments we set pCube's parameters such that the cells are stored in Z-order. We used the same ordering to pack the data onto disk pages for the benchmark algorithms.

The first comparison benchmark algorithm ("ideal index") behaves like an optimal index structure where an index on each dimension exists. In contrast to real index structures in this algorithm index accesses and the computation of the relevant data pages incur no overhead. Thus, the cost is equal to the number of pages that contain non-empty cells in the query's range. Accesses are random I/O. The second comparison benchmark algorithm ("ideal scan") simulates an optimal index for range queries. Knowledge about the first and the last page that contain cells in the query range are available at no cost. Its cost is measured as the number of sequential page accesses to pages between the given pages. Both benchmark algorithms do not provide early feedback.

Figures 5 and 6 contain the experimental results for a real-world data set (CENSUS, see Table 4). The data was obtained from the U.S. Census Bureau database [5]. We extracted the 1995 Current Population Survey's person data file. The data cube has 10 dimensions (age, class of worker, family type, weight, age recode, educational attainment, marital status, race, hispanic origin, sex). The data cube is extremely sparse (less than 0.0001% non-empty cells). Figure 5 shows that queries with large query-cubes benefit from storing aggregate information at internal nodes, irrespective of whether the query runs till completion (exact result) or is aborted earlier (when the approximation is good enough). If a user is satisfied with a relative approximation error of 20%, the savings in access costs are substantial. For large queries the algorithm can take advantage of the order in which the nodes are stored and simply scan the pages that contain the internal nodes (since most of them intersect the query-cube anyway). In the case of the first benchmark algorithm (Figure 6, left graph), the larger the query-cube, the more non-empty cells are included and the more pages have to be accessed. Note, that real-world algorithms require additional accesses to index pages to find the relevant data pages. All accesses are random I/O. In contrast, the scanning benchmark algorithm (right graph) can take advantage of faster sequential I/O. On the other hand, it has to scan almost the entire set of data pages, even for small ranges. The results show that the user can be provided with fast approximate answers at almost no extra costs. When the application does not require exact results, savings in query costs can be substantial.

For a dense data set pCube performs even better (see Figures 7 and 8). The results shown here were obtained for a four-dimensional data cube where each dimension has 16 possible values (see Table 4 for more details). Especially for large query-cubes pCube clearly outperforms the benchmark algorithms. The main reason is that for a dense data cube the spatial decomposition produces a balanced tree where

Data set	Dimensions	Aggregate function	Non-empty cells	Internal nodes (pCube only)
CENSUS	10	COUNT	45988 (352 pages)	11598 (161 pages)
DENSE	4	SUM	39319 (151 pages)	4369 (80 pages)
ZIPF	11	SUM	49996 (404 pages)	10698 (170 pages)

Table 4: Data sets used in the experiments

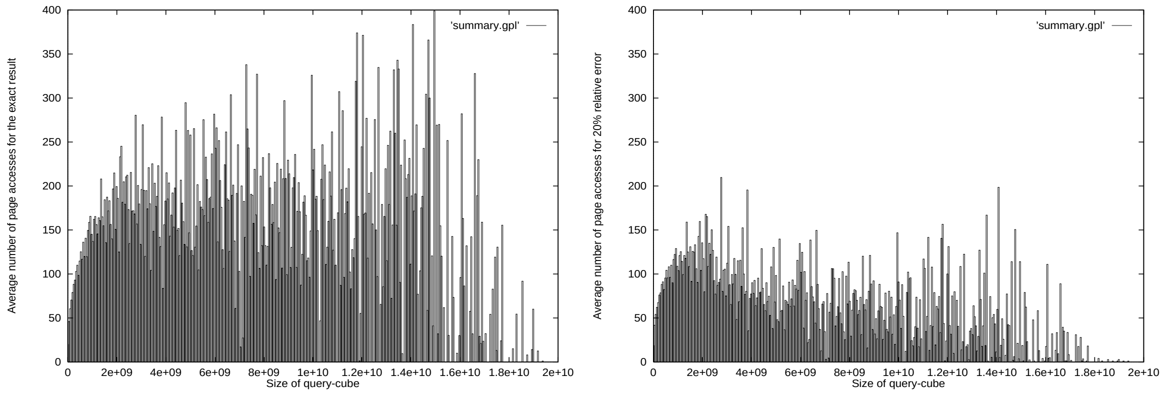

Figure 5: CENSUS, pCube, page accesses for the exact result and a relative error of 20%

almost all possible internal nodes are created. Thus the number of leaf nodes accessed is dramatically reduced.

Since our pCube implementation is based on a regular spatial decomposition technique, we expected a performance degradation with increasing skew in the data distribution. The worst results were obtained for an 11-dimensional data cube (dimension domains of size 2, 4, 8, 16, 32, 64 and the others 128) where the values for eight dimensions are Zipf distributed (skew parameter between 1.8 and 1.2), and the values for the other three are uniformly distributed. Due to the skew the tree is poorly balanced. Queries that access nodes in the dense region of the data cube intersect many tree nodes, resulting in a large number of disk accesses. However, even for this "worst case scenario" the pCube implementation provides fast answers with error bounds at moderate extra costs (Figures 9 and 10).

In the appendix further results for the three data sets are presented for 5% and 40% relative error.

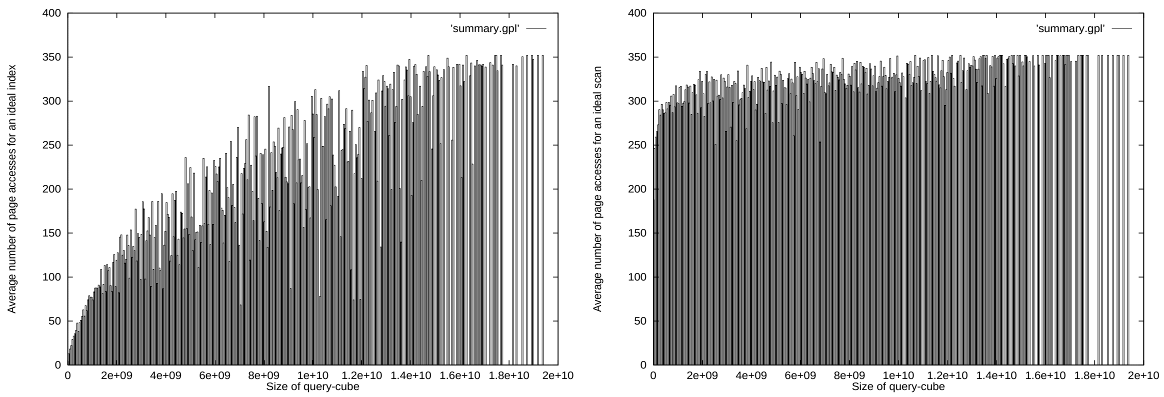

Figure 6: CENSUS, Benchmark, page accesses for the exact result

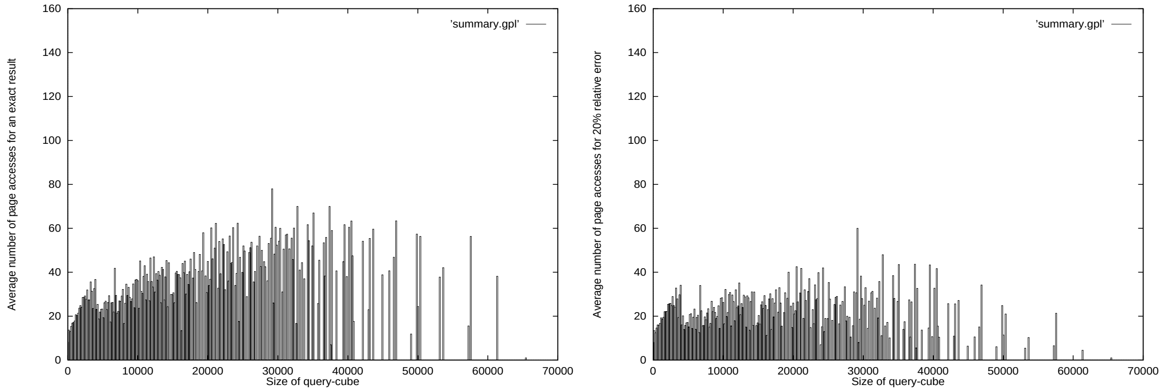

Figure 7: DENSE, pCube, page accesses for the exact result and a relative error of 20%

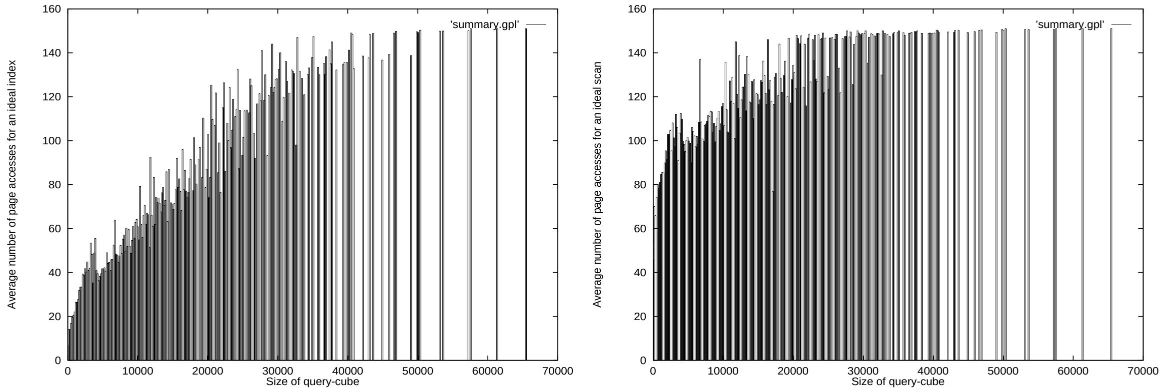

Figure 8: DENSE, Benchmark, page accesses for the exact result

5 Conclusion

The experiments show that pCube is a powerful concept for supporting efficient online aggregation. In contrast to earlier approaches, pCube provides early feedback with absolute error bounds for range queries on data cubes. Since the worst case update and query costs depend on the number of *non-empty* cells, rather than on the dimensionality or the size of the data cube (including empty cells), our technique can efficiently handle high-dimensional and sparse data cubes. The combination of a hierarchical data structure and simple approximation models at the tree nodes enables fast updates and progressively improving error bounds.

In real-world applications attributes often form hierarchies. The selection of ranges often corresponds to those hierarchies (e.g., one can expect more queries that summarize over the range 01/01/1999-12/31/1999, than for the range 01/14/1999-01/13/2000). Incorporating this knowledge into the pCube implementation would lead to a better "match" of query boundaries and node-regions. For pCubes that are based on spatial decomposition a fixed hierarchy can be chosen for an attribute. Then the partitioning is performed based on the hierarchy (e.g., instead of partitioning a week into two "half-weeks" it will be split into seven days). We expect such a partitioning to result in performance improvements.

The concept of pCube is very general. Our experiments were performed for just one of its many possible implementations. Still the results are very encouraging and show, that early feedback with absolute error bounds can be added to data analysis tools at a low extra cost. There are, however, aggregate functions like MostFrequentValue, for which pCube in its current version is not applicable. Our future work includes the development of concepts that provide pCube's properties for diverse functions and hierarchy based partitioning.

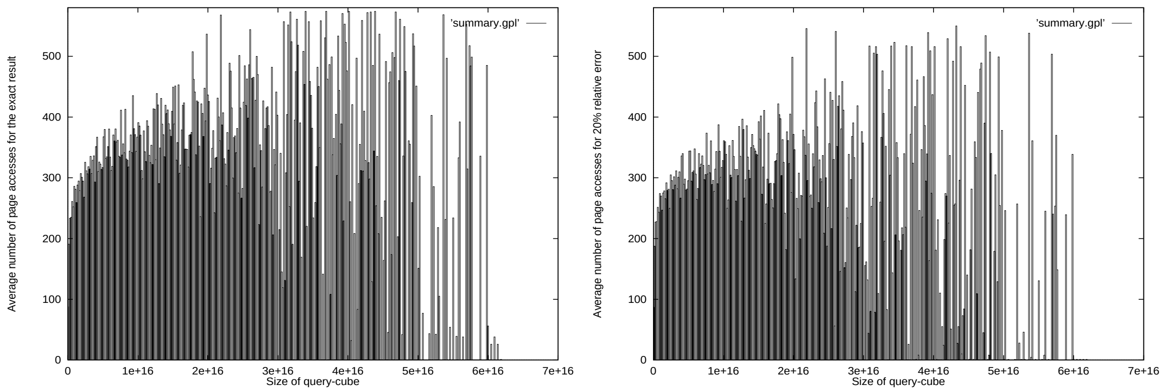

Figure 9: ZIPF, pCube, page accesses for the exact result and a relative error of 20%

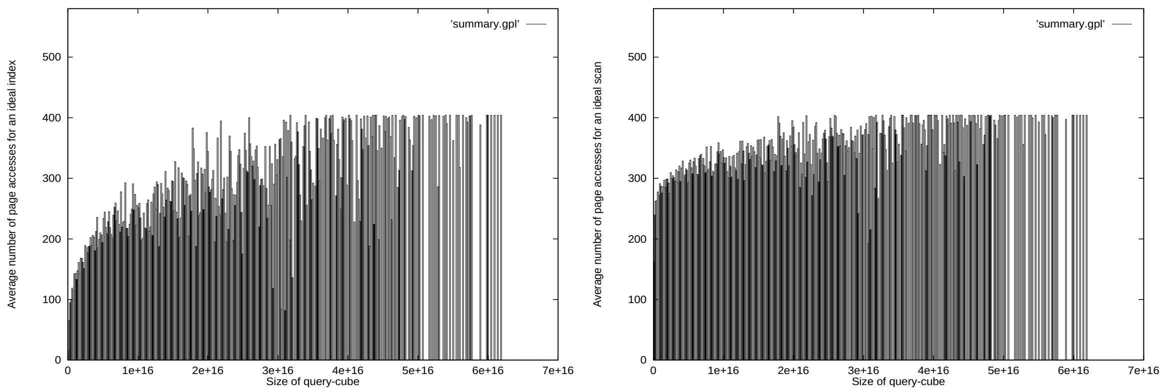

Figure 10: ZIPF, Benchmark, page accesses for the exact result

References

[1] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *Proc. SIGMOD*, 1999.

[2] D. Barbará et al. The new jersey data reduction report. *Data Engineering Bulletin*, 20(4), 1997.

[3] D. Barbará and X. Wu. Using loglinear models to compress datacubes. Technical report, George Mason University, 1999.

[4] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg CUBEs. In *Proc. ACM SIGMOD*, 1999.

[5] U.S. Census Bureau. http://www.census.gov/main/www/access.html.

[6] C.-Y. Chan and Y. E. Ioannidis. Hierarchical cubes for range-sum queries. In *Proc. 25th VLDB*, 1999.

[7] D. W. Cheung, B. Zhou, B. Kao, K. Hu, and S. D. Lee. DROLAP — a dense-region based approach to on-line analytical processing. Technical report, Univ. of Hong Kong, 1999.

[8] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2), 1998.

[9] S. Geffner, D. Agrawal, and A. El Abbadi. The dynamic data cube. In *Proc. EDBT*, 2000. To appear.

[10] S. Geffner, M. Riedewald, D. Agrawal, and A. El Abbadi. Data cubes in dynamic environments. *Data Engineering Bulletin*, 22(4), 1999.

[11] P. B. Gibbons and Y. Matias. Synopsis data structures for massive data sets. Technical report, Bell Labs, 1998.

[12] S. Goil and A. Choudhary. BESS: Sparse data storage of multi-dimensional data for OLAP and data mining. Technical report, Northwestern University, 1997.

[13] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, pages 29–53, 1997.

[14] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for OLAP. In *Proc. ICDE*, 1997.

[15] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. ACM SIGMOD*, 1996.

[16] J. M. Hellerstein et al. Interactive data analysis with CONTROL. *IEEE Computer*, August 1999.

[17] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proc. ACM SIGMOD*, 1997.

[18] C. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *Proc. ACM SIGMOD*, 1997.

[19] T. Johnson and D. Shasha. Some approaches to index design for cube forests. *IEEE Data Engineering Bulletin*, 20(1), March 1997.

[20] Y. Kotidis and N. Roussopoulos. An alternative storage organization for ROLAP aggregate views based on cubetrees. In *Proc. ACM SIGMOD*, 1998.

[21] P. E. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proc. SIGMOD*, 1997.

[22] N. Pendse and R. Creeth. The OLAP report. http://www.olapreport.com/Analyses.htm, 1999. Parts available online in the current edition.

[23] V. Poosala and V. Ganti. Fast approximate answers to aggregate queries on a data cube. In *Proc. SSDBM*, 1999.

[24] J. Shanmugasundaram, U. Fayyad, and P. S. Bradley. Compressed data cubes for OLAP aggregate query approximation on continuous dimensions. In *Proc. SIGKDD*, 1999.

[25] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proc. ACM SIGMOD*, 1999.

[26] W. Wang, J. Yang, and R. Muntz. STING: A statistical information grid approach to spatial data mining. In *Proc. VLDB*, 1997.

[27] G. K. Zipf. *Human Behaviour and the Principle of Least Effort.* Addison-Wesley, Reading, MA, 1949.

A Additional Experimental Results

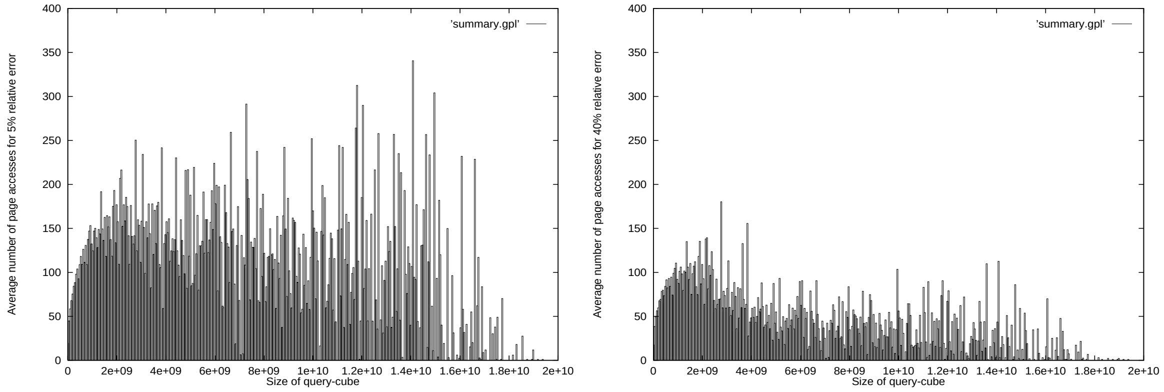

Figure 11: CENSUS, pCube, page accesses for relative errors of 5% and 40%

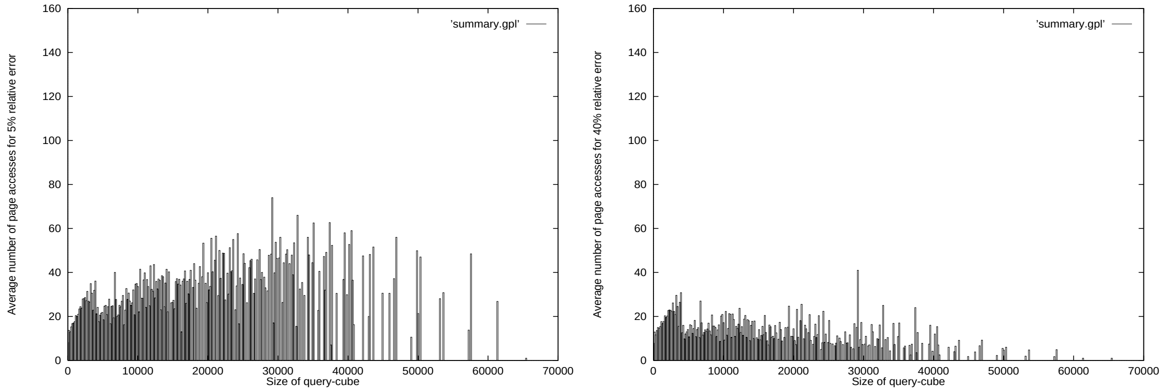

Figure 12: DENSE, pCube, page accesses for relative errors of 5% and 40%

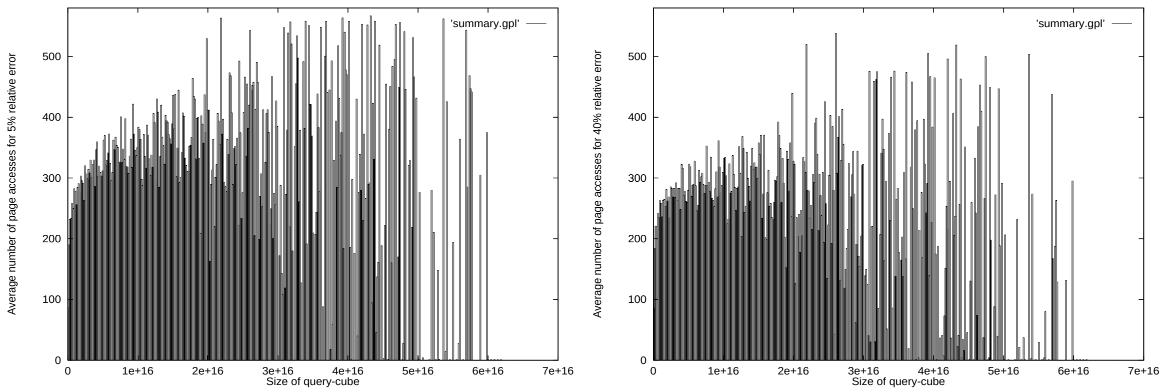

Figure 13: ZIPF, pCube, page accesses for relative errors of 5% and 40%