Bulletin of the Technical Committee on

Data Engineering

December 1999 Vol. 22 No. 4

IEEE Computer Society

Letters

Special Issue on Retrieval and Maintaince of Summary Data

Conference and Journal Notices

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

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

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

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

The web page for the Data Engineering Bulletin is http://www.research.microsoft.com/research/db/debull. The web page for the TC on Data Engineering is http://www.ccs.neu.edu/groups/IEEE/tcde/index.html.

Letter from the Editor-in-Chief

Changing Bulletin Editors

The Bulletin practice is to appoint editors for two years. Each editor is responsible for producing two issues, one per year, during that time. Among the current editors, Surajit Chaudhuri and Donald Kossmann have now completed their two issues and are due to "retire". Our profession depends on hardworking volunteers for the vitality of publications like the Bulletin. So I would like to thank both Surajit and Donald for their hard work and very successful results. Producing high quality issues of the Bulletin is not an accident. It is the result of the work of very capable issue editors.

I am now delighted to announce the appointment of two new editors, Sunita Sarawagi and Alon Levy. Sunita and Alon both have outstanding research reputations.

Sunita Sarawagi is currently on the faculty in the School of Information Technology at IIT Bombay. She has done research in data warehousing, OLAP, data mining and tertiary storage. Her career began as a UC Berkeley student, following which she went to the IBM Almaden Research Center. Sunita was co-author of a paper that received a 1998 SIGMOD best paper award.

Alon Levy is a Seattle neighbor of mine as he is on the faculty of the Computer Science and Engineering Department at the University Washington. Alon's expertise includes data integration, web-site management, semi-structured data, query optimization, and the interaction of databases with AI. He is a graduate of Stanford and was on the technical staff at ATT Bell Labs. In June he co-founded a company building tools for XML data.

This Issue

Databases are increasingly used for data analysis. When done "on-line", this is referred to as OLAP, a field that has become increasingly important over the past five or more years. The data cube is an important part of OLAP work, but by no means the only part. The current issue looks at how summary data can be maintained and retrieved, which is surely relevant in a very broad and important way to data analysis, including the data cube.

In a certain sense, most of the work that is reported in the current issue is related to performance. Users want to summarize data so that they can browse it, explore it, understand it. The work reported here is mostly about how this summarizing can be done efficiently. Were efficiency not an issue, the functionality would not require much more than a relational database. But performance is, of course, critically important. Indeed, we need real insights in this area, where approximation and sampling play a frequent role in making summarization feasible.

Amr El Abbadi,the issue editor, has gathered for this issue a diverse collection of techniques that address the retrieval and maintenance of summary information. These include histograms, sampling, incremental maintence of cubes and more. This material will become increasingly important in the future as data analysis becomes even more pervasive. I want to thank Amr for assembling this issue, which well represents some of the most promising techniques in this important area.

David Lomet
Microsoft Research

TC on Data Engineering: Election of Chair for 2000-01

The Chair of the IEEE Computer Society Technical Committee on Data Engineering (TCDE) is elected for a two-year period. The mandate of the current Chair, Betty J. Salzberg, terminates at the end of 1999. Hence is time to elect a Chair for the period January 2000 to December 2001. Please vote using the ballot on the next page.

The Nominating Committee, consisting of Paul Larson, Erich Neuhold, Masaru Kitsuregawa, and Meral Ozsoyoglu, is nominating Betty J. Salzberg for a second term. The Committee felt that it was in the best interest of the TCDE that Betty continue for another term and, hence, decided not to nominate additional candidates.

BETTY J. SALZBERG

Biography Betty Salzberg is a Professor of Computer Science in the College of Computer Science at Northeastern University in Boston. Her Ph.D. was in Mathematics at the University of Michigan. Her areas of expertise in database systems include access methods, concurrency and recovery and on-line reorganization.

She has published in the ACM Transactions on Database Systems, the VLDB Journal, Information Systems, Acta Informatica and Computing Surveys. She has written two textbooks on database systems. Her work has appeared in IEEE ICDE, ACM SIGMOD and VLDB conferences. Her research has been funded by the NSF since 1988 and she has also received grants from Digital Equipment Corporation and Microsoft.

Professor Salzberg is the current chair of the IEEE TCDE and served as an associate editor of the IEEE Data Engineering Bulletin in 1996-97. She has served several times on the program committees of the VLDB conference, the SIGMOD conference and the IEEE ICDE conference. In 1998, Professor Salzberg chaired the organizing committee for the NSF Workshop on Industrial/Academic Cooperation.

Position Statement The Technical Committee on Data Engineering publishes a newsletter, the Data Engineering Bulletin, and sponsors the ICDE (International Conference on Data Engineering) and the associated workshop RIDE (Research Issues in Data Engineering). These are the main tasks of the committee.

In addition, in the last two years, we have cooperated with ACM SIGMOD in making available the ICDE proceedings and the Bulletin on CDROM. The CDROM, which also contains SIGMOD and VLDB proceedings, will be available for TCDE members for free. It is inevitable that people with the same interests (TCDE and SIGMOD) will want to share information. Whether they belong to the IEEE or ACM or both, people working in the database industry or on database research want the same access to journals, conferences and newsletters covering their speciality. This is a trend I see continuing and one which our TC must encourage by collaboration with our counterparts in the ACM.

Another trend of which we are all aware is web-based communication. The TCDE has increased our presence on the web. Our newsletter is now almost wholly web-based and we have a Technical Committee web page with versions in English, Chinese, Japanese and Malaysian and with sites in Japan and Europe (Norway) as well as America. The American page is at http://www.ccs.neu.edu/groups/IEEE/tcde/.

In my last position statement, I suggested that there be more of an industry presence in the IEEE TCDE. The workshop I proposed to the NSF on industrial/academic cooperation in database systems took place in October 1998 and its webpage is available at http://www.ccs.neu.edu/groups/IEEE/ind-acad/. Three of the eight people on the TCDE executive board are from industry. One of the co-program chairs and the general chair of ICDE 2000 and the keynote speaker and several of the other members of the organizing committee are from industry. As continuing chair, I would try to increasingly involve industry people more in the TCDE and in its conference, ICDE. Since research in database systems is not purely theoretical, it suffers without the infusion of experience and information practitioners are able to supply.

The Nominating Committee and the entire TC Executive Committee urge you to vote and to return your ballot to the IEEE Computer Society address given on the ballot on the following page.

Paul Larson
Chair of the Nominating Committee

ELECTION BALLOT

TECHNICAL COMMITTEE ON
DATA ENGINEERING

The Technical Committee on Data Engineering (TCDE) is holding an election for Chair. The current term of Betty J. Salzberg has expired. Please mail or fax in your vote.

BALLOT FOR ELECTION OF CHAIR
Term: (January, 2000 - December, 2001)

Please vote for one candidate.

☐ Betty Salzberg

☐ ___________
(write in)

Your Signature:____

Your Name:____

IEEE CS Membership No.:________________________________
(Note: You must provide your member number. Only TCDE members who are Computer Society members are eligible to vote.)

Please fax or mail the ballot to arrive by January 25, 2000 to:

nschoultz@computer.org
Fax: +1-202-728-9614

IEEE Computer Society
Attn: Nichelle Schoultz
1730 Massachusetts Avenue, NW
Washington, DC 20036-1992

RETURN BY January 25, 2000

Letter from the Special Issue Editor

In this special issue, we address some recent, state-of-the-art, approachs for querying, approximating, estimating and maintaining large data sets. In many of these cases, users would like to start with summary or approximate answers that would guide them in the direction of their quest. This approximate answer may be useful in itself, or may be a first step to more detailed and more specific drill down queries. Large data sets are often characterized by multiple attributes that refer to various characteristics of the data. As a result, large collections of data are often summarized in a data cube, which has frequently been used for OLAP applications. Many of the papers in this special issue specifically address the issue of approximating and maintaining information in large data cubes.

Poosala, Ganti and Ioannidis have been involved in an effort to build an efficient data analysis system called Aqua. In the first paper they describe how to use histograms to approximate queries when precise answers are not necessary and early feedback is helpful. Given the exploratory nature of many OLAP applications, precise answers often result in answers of no particular interest to the users. However, for approximate answers, users need to be provided with some measure of confidence in the answers. Hence, in this paper, histograms are enhanced to provide quality guarantees on the approximate answers. Furthermore, the authors provide an efficient technique for selecting histograms that result in the most accurate answers while using small amounts of space.

In the second paper, Barbara and Wu describe how Quasi-Cubes can be used to compress data cubes and provide approximate answers with guaranteed bounds on the errors. They also discuss a variety of applications that would benefit from such approximate answers inclulding data marts (light-weight data warehouses), and approximate data mining. In such data mining applications, exploratory data analysis tries to determine which factors have the most influence on the data values. Alternatively, decision trees can be used as a classification technique to forecast characteristics of various data items. In such applications fast approximate answers have the potential of resulting in significant benefits in efficiency.

Johnson and Shasha address the problem of maintaining large data cubes. In particular, they present a new storage structure called cube forests. They present algorithms for the querying and updating of cube forests and also describe how to perform batch updates. An implementation of cubes forests is described and a simulation study using the TPC-D benchmark shows the significant reduction of costs for batch updates.

Geffner, Riedewald, Agrawal and El Abbadi describe two alternative approaches for storing data cubes. These approaches present a tradeoff between the cost of querying versus the cost of updating. In the relative prefix sum approach query cost is kept constant but updates are proportional to the square root of the size of the cube. This approach is useful for applications where queries are the dominant application, but the data still needs to be updated. In the second approach, the dynamic data cubes, both queries and updates are have logarithmic cost, which makes this data structures especially appropriate for speculative, what-if analytical processing applications.

Chaudhari and Motwani propose using sampling as a method for estimating query results. In fact, they argue that for many data mining and OLAP applications, sampling must be supported on the result of an arbitrary query, not just on the data itself. For this purpose, they explore the major impediments to implementing sampling as a primitive relational operation. They demonstrate that auxiliary information such as histograms can be leveraged to facilitate efficient sampling.

The papers in this issue represent on-going research and implementation efforts. They pose several open and interesting problems and hence I hope this special issue will lead to more research and investigations in this increasingly significant area. Current applications range from traditional databases to OLAP and data mining and potentially are useful in diverse domains such as digital libraries, biological computing, as well as enviromental and economic applications.

Amr El Abbadi
UC Santa Barbara

Approximate Query Answering using Histograms

Viswanath Poosala
poosala@lucent.com
Bell Laboratories
Lucent Technologies
Murray Hill, NJ, USA

Venkatesh Ganti
vganti@cs.wisc.edu
Department of Computer Sciences
Univ. of Wisconsin-Madison
Madison, WI, USA

Yannis E. Ioannidis
yannis@cs.wisc.edu
Department of Informatics
Univ. of Athens
Athens, Greece

Abstract

Answering queries approximately has recently been proposed as a way to reduce query response times in on-line decision support systems, when the precise answer is not necessary or early feedback is helpful. In this article, we explore the use of precomputed histograms *for approximate answering of aggregate queries. Histograms are used by most database systems for selectivity estimation within their optimizers. However, the use of histograms for approximate query answering raises several novel issues, which are addressed in this article. We present a* histogram algebra *for efficiently executing complex SQL queries on histograms within a DBMS without requiring any changes to the DBMS internals. We enhance histograms to estimate the quality of the approximate answers. Finally, we present an efficient technique for selecting a provably* near-optimal *set of histograms on the data cube, which minimizes the space needed when an upper bound on errors is given.*

1 Introduction

The users of decision support applications pose very complex queries to Database Management Systems (DBMSs), which take a long time to execute. Given the exploratory nature of such applications, many of these queries end up producing no result of particular interest to the user. Much wasted time could have been saved if users were able to quickly see an *approximate answer* to their query, and only proceed with their complete execution if the approximate answer indicated something interesting.

Several techniques have been proposed for providing approximate answers using statistical summaries of the data, such as samples, histograms, and wavelets. In this article we focus on histograms, which have the advantages of being present in almost every commercial DBMS and being reasonably accurate (for selectivity estimation). There are also two different ways to present approximate answers: the *online aggregation* approach constantly refines the answer using larger and larger sets of statistics of the data until the accurate answer is obtained [8], whereas the *precomputation* approach presents a small number of discrete approximate answers (typically, just one) by using precomputed summaries of the data [1]. The precomputation approach has the advantage of being faster in providing an answer because only the small summary data has to be processed at run-time; on the other hand online aggregation has the flexibility of refining the answers. However, it is possible to deploy

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

a precomputation approach without requiring any changes to the DBMS, while online aggregation requires new query processing and data access strategies to be implemented in the DBMS. Based on these trade-offs, we use precomputed histograms in our work[1].

In this paper, we investigate the above approach to approximate query answering and present two different ways of using histograms for this purpose. Though histograms have been widely used in databases, their usage has been mostly restricted to selectivity estimation [15, 9, 19, 12, 14]. The use of histograms for approximate query answering brings up several novel issues to fore, which form the main focus of this article. Our contributions are summarized below.

- **Efficient Query Execution:** We propose storing histograms as regular relations in a relational DBMS and appropriately translating regular database queries into equivalent queries on the histograms so that approximate query answers can be obtained using the same mechanism as exact query answers. To this end, we define a *histogram algebra* that can be used for this translation.
- **Quality of Answers:** In order to provide any confidence to the user in using an approximate answer, a measure of the quality of the answer must also be provided. Unlike sampling-based techniques which often provide a confidence measure for this purpose, traditional histogram-based techniques do not offer any error measures. Hence, we enhance histograms to provide quality guarantees on the approximate answers.
- **Histogram Selection:** Given the limited resources for storing summaries, it is important to choose histograms that result in the most accurate answers to queries while using small amounts of space. Presumably, a common situation in an approximate query answering system is where the user specifies a bound on the errors in the answers and demands minimal space usage. We provide an efficient greedy technique for selecting a near-optimal set of histograms on the data cube when an error bound is given.

Due to space limitations, we omit our experimental study from this article, and refer the reader to other papers for those

This work constitutes a part of our efforts to build an efficient data analysis system called Aqua [1]. In this system, we store the statistics (histograms and samples) as relations in the DBMS and rewrite the user query posed on the original relations as a query on the statistics relations. The rewritten query is then submitted to the DBMS for execution. This middleware architecture, coupled with the provision of standard ODBC interfaces, enables Aqua to be easily deployed between almost any front-end querying system and backend DBMS. More details about the Aqua system can be found in [1].

Next, we provide a background on histogram techniques.

2 Histograms

In this section, we summarize standard histogram-based techniques for approximating the data in a database [19, 18]. First, we present some useful definitions.

The *value set* $\mathcal{V}_i$ of attribute X_i is the set of values of X_i that are present in R. Let $\mathcal{V}_i = \{\, v_i(k)\colon\ 1 \leq k \leq D_i \,\}$, where $v_i(k) < v_i(j)$ when $k < j$. The *spread* $s_i(k)$ of $v_i(k)$ is defined as $s_i(k) = v_i(k+1) - v_i(k)$, for $1 \leq i \leq D_i$. (We take $s_i(D_i) = 1$.) The *frequency* $f_i(k)$ of $v_i(k)$ is the number of tuples in R with $X_i = v_i(k)$. The *area* $a_i(k)$ of $v_i(k)$ is defined as $a_i(k) = f_i(k) \times s_i(k)$. The *data distribution* of X_i is the set of pairs $\mathcal{T}_i = \{\, (v_i(1), f_i(1)), (v_i(2), f_i(2)), \ldots, (v_i(D_i), f_i(D_i)) \,\}$. Typically, real-life attributes tend to have *skewed* data distributions, i.e., they may have unequal frequencies and/or unequal spreads.

A *histogram* on an attribute X is constructed by using a *partitioning rule* to partition its data distribution into β (≥ 1) mutually disjoint subsets called *buckets* and approximating the frequencies and values in each bucket in

[1]It may also be possible to perform online aggregation using histograms, however that is outside the scope of this article.

some common fashion. In particular, the most effective approach for values is the *uniform spread* assumption [19], under which the attribute values in a bucket are assumed to be placed at equal intervals between the lowest and highest values in the bucket. The most effective approach for frequencies is to approximate the frequencies in a bucket by their average (*uniform frequency assumption*). Example 1 illustrates the above concepts.

Example 1: The following table shows how each parameter defined above is instantiated for a hypothetical attribute.

Quantity	**Data Distribution Element**					
Value	10	60	70	80	85	100
Frequency	100	200	600	200	80	200
Spread	50	10	10	5	15	1
Area	5000	2000	6000	1000	1200	200

Consider a 3-bucket histogram on this attribute with the following bucketization of attribute values: $\{10\}, \{60, 70\}, \{80, 85, 100\}$. The approximate distribution captured by this histogram looks as follows.

Quantity	**Data Distribution Element**					
Approx. Value	10	60	70	80	*90*	100
Approx. Frequency	100	400	400	160	160	160

Conceptually, one can "expand" a histogram into a relation containing the approximate attribute values as its tuples, with each tuple appearing as many times as the approximate frequency of that value. We call this the *approximate relation* (*ApproxRel*) of that histogram.

Histograms can also be built on multiple attributes together, by partitioning the joint distribution of the attributes into multi-dimensional buckets and using extensions of the uniform frequency and spread assumptions. It is also possible to *combine* two histograms on different sets of attributes to obtain a single histogram on the union of those two sets by making the *attribute value independence assumption*. All of these details are given in [18]. In practice, there may be several one- and/or multi-dimensional histograms on a relation R. For simplicity of presentation, we assume in the rest of the paper that there is a single (multi-dimensional) histogram on a relation computed under the above assumption.

Given the mechanisms of approximation within a histogram, it is clear that the accuracy of the approximation is determined by which attribute values are grouped together into each bucket. Several partitioning rules have been proposed for this purpose. For example, in an *equi-width* histogram, all buckets are assigned value ranges of equal width; in an *equi-depth* histogram, all buckets are assigned the same total number of tuples. In earlier work, we have introduced several new classes of histograms and identified a particular class of histograms, that we call *V-Optima(V,A)*, which performs the best in estimating the selectivities of most kinds of queries. In a *V-Optimal(V,A)* histogram, buckets are formed such that the sum of the weighted variances of the areas within the buckets is minimized (the weights being the number of values in that bucket). Recall that *area* captures both the frequency and value domains. By trying to group together similar frequencies and spreads, the V-Optimal(V,A) histogram ensures that the uniform frequency and spread assumptions do not cause much errors.

With respect to storage, each bucket in a histogram keeps the following information: the total number of tuples that fall in the bucket (tot), and for each dimension i, the low and high values (lo_i, hi_i) and the number of distinct values ($count_i$) in that dimension (the subscripts are dropped for single-dimensional histograms). For the purpose of this work, we store histograms as regular relations in the database with each bucket forming a tuple. For ease of explanation in later sections, we also include additional set of columns: the average spreads along each dimension ($sp_i = \frac{hi_i - lo_i}{u_i - 1}$) and the average frequency for the bucket ($avg = \frac{tot}{u_1 u_2 \ldots u_d}$). In the rest of the paper, the term *histogram* refers to the histogram relation described here. For illustration, we present the histogram relation for the histogram given in Example 1:

lo	*hi*	*count*	*tot*	*sp*	*avg*
10	10	1	100	0	100
60	70	2	800	10	400
80	100	3	480	10	160

3 Approximate Query Answering using Histograms

There are essentially two different ways to use histograms for approximate query answering, roughly corresponding to the relational and multi-dimensional OLAP approaches. They are described below.

Relational Approximation: In this approach, histograms are used to approximate the data in the relation (as explained in the previous section). Queries on the data are answered based on the approximate relations captured by the histograms. The key issue in relational approximation is the efficient execution of queries. This is addressed in Section 4.

Data Cube Approximation: In this approach, histograms are used to approximate the OLAP *data cube* and aggregate queries on the data cube are answered from the "approximate data cube". A data cube is essentially a hierarchy of multi-dimensional *sub-cubes*, where each sub-cube describes the aggregate distribution of data in a subset of dimensions [6]. The axes of the sub-cube contain the distinct values in each of the *dimensional attributes*; and the cells contain the aggregate (sum, max, min, etc) value of one of the *measured attributes* in the relation. An *approximate sub-cube* has the same structure as the original sub-cube, but with approximate values on the cells and the axes. In particular, we obtain the approximation by building a histogram on the multi-dimensional distribution represented by the sub-cube, i.e., *the dimensions form the value domains and the cell values are treated as their frequencies*.

We illustrate this using the following example.

Example 2: Consider the `LINEITEM` table in the TPC-D database and three of its attributes `Supplier id (S)`, `Customer id (C)`, and `Part id (P)`. Figure 1 shows the tuples in an example `LINEITEM` table. Figures 2 and 3 show the sub-cubes corresponding to applying `SUM` aggregate on $\{$ `P,S` $\}$ and $\{$ `P` $\}$ respectively. In Figure 4, a trivial approximate sub-cube for $\{$ `P,S` $\}$ is shown, which is obtained by replacing all the measure values by their average (= total (54)/count (6)), i.e., using a single bucket histogram.

P	S	C	Q
100	200	300	1
100	200	301	7
100	201	300	3
100	201	301	9
101	200	302	11
101	202	300	23

Figure 1: $\{$ `P,S,C` $\}$

	S		
P	200	201	202
100	8	12	0
101	11	0	23

Figure 2: $\{$ `P,S` $\}$

P	
100	101
20	34

Figure 3: $\{$ `P` $\}$

	S		
P	200	201	202
100	9	9	9
101	9	9	9

Figure 4: Approx $\{$ `P,S` $\}$

Note that any query on the original sub-cube $\{$ `P,S` $\}$ can also be answered, albeit inaccurately, from the approximate sub-cube. In order to answer all queries on the data cube, one needs to approximate the entire data cube while using small amount of space. This is addressed in Section 6.

4 Query Execution on Histograms

In this section, we develop techniques for automatically translating SQL queries on original relations into SQL queries on histogram relations. These techniques are used in the relational approximation approach described above.

First, we define the notion of providing *valid approximate answers* to a query using histograms. Let *ApproxRel(H)* be the approximate relation corresponding to a histogram H on relation R (Section 2). Then, we believe that the following definition captures the intuition behind an approximate query answer based on histograms.

Definition 1: Consider a query Q operating on relations $R_1..R_n$, and let H_i be the histogram on R_i. The *valid approximate answer* for Q and $\{H_i\}$ is the result of executing Q on ApproxRel(H_i) in place of R_i, for $1 \leq i \leq n$.

An obvious way to derive the valid approximate answer is as follows: first, compute the approximate relations of all the histograms on the relations in the query; next, execute the query Q on these relations. For a 1-dimensional histogram H, its approximate relation can be computed using the following SQL query ,called `Expand.sql`[2].

> SELECT $(H.lo + I_C.idx * H.sp)$
> FROM H, I_C, I_A
> WHERE $I_C.idx \leq H.ct$ & $I_A.idx \leq H.avg$;

Here, H is the histogram stored as a relation and I_A, I_C are auxiliary relations, each with a single attribute idx. Relation I_A (resp., I_C) contains the integers $1, 2, .., A$ (resp., $1, 2, .., C$), where A (resp., C) is the largest *average frequency* (resp., *count*) in the buckets of H. Essentially, this query uses I_C to generate the positions of values within each bucket and then uses the *low* and *spread* values of the bucket to compute each of the approximate values, under the uniform spread assumption. Then, it uses I_A to replicate each value based on its frequency.

However, this approach is inefficient because ApproxRel(H_i) may have as many tuples as R_i itself, thus defeating the whole purpose of approximate query answering. Hence, we describe a far more efficient approach for computing valid approximate answers, which executes directly on histograms:

1. Obtain a *valid translation* Q' of Q, which would at the end of these three steps yield a *valid approximate answer* (described next).

2. Execute Q' on $\{H_i\}$ to obtain a result histogram H_{res}

3. Compute ApproxRel(H_{res}) using `Expand.sql`

Since most of the query processing takes place on small histogram relations, this approach is clearly very efficient.

4.1 Translations for Non-Aggregate Queries

These queries are equivalent to relational algebra expressions involving just *selection, projection*, and *join* operations. A query Q in this category is translated as follows:

1. Construct an operator tree T of *select, project*, and *join* operations that is equivalent to Q.
2. Replace all the base relations in T by their corresponding histograms to obtain another tree T'.
3. Starting from the bottom of T', translate each operator into an SQL query that takes one or two histograms from the operator's children and generates another histogram as output.

The translations for various query operators are described below.

- **Equality Selection** ($\sigma_{A=c}$): Equality selection is translated into the following query $Q_=$:

[2] It is straightforward to generalize it so that it works with a multi-dimensional histogram, but it becomes quite complex without offering any new insight, so we do not present it.

```
SELECT c, c, 1, avg
FROM H
WHERE (c ≥ lo) & (c ≤ hi) & (mod(c − lo, sp) = 0);
```

- **Range Selection** ($\sigma_{A \leq c}$): Range selection is translated into the following query Q_σ:

SELECT * FROM H WHERE $hi \leq c$; $\cup$ SELECT $lo, lo + sp * \lfloor \frac{c-lo}{sp} \rfloor, \lfloor \frac{c-lo}{sp} \rfloor, avg$ FROM H WHERE $(lo \leq c)$ & $(hi > c)$;

- **Projection** (π_A): Assuming duplicate elimination, projection is translated into the following query Q_π:

```
SELECT lo, hi, count, 1
FROM H;
```

Assuming no duplicate elimination, projection is just the identity query (i.e., selecting all tuples from the histogram relation with no changes).

- **Equi-Joins** ($R_1 \bowtie_{R_1.A=R_2.B} R_2$): Let H_i be the histogram on the joining attribute of R_i, and N_i be the largest count in the buckets of H_i. Join is translated into a sequence of two queries, $Q1_\bowtie$ and $Q2_\bowtie$[3]. The first query ($Q1_\bowtie$) computes the frequency distribution of the approximate join result by joining the approximate frequency distributions of H_1 and H_2. It assumes the existence of two auxiliary relations of integers I_{N_1} and I_{N_2} defined in the same fashion as I_C described earlier.

SELECT $(H_1.lo + I_{N_1}.idx * H_1.sp)$ as v, $H_1.lo$ as lo_1, $S.lo$ as lo_2, $H_1.avg * H_2.avg$ as $navg$
FROM $H_1, H_2, I_{N_1}, I_{N_2}$
WHERE $(H_1.lo + I_{N_1}.idx * H_1.sp = H_2.lo + I_{N_2}.idx * H_2.sp)$ &
$(I_{N_1}.idx \leq H_1.count)$ & $(I_{N_2}.idx \leq H_2.count)$;

The second query ($Q2_\bowtie$) converts the result of query Q (say, $Q1R$) into a histogram by appropriate grouping.

```
SELECT min(v), max(v), count(*), navg
FROM Q1R
Group By lo1, lo2, navg;
```

4.2 Aggregate Queries

In general, an aggregate query Q_{agg} can be viewed as computing aggregates over some of the attributes in the result of a non-aggregate query Q. Hence, a valid translation for Q_{agg} consists of a valid translation for Q producing a histogram H, followed by an aggregate-specific SQL query on H computing a single bucket histogram containing the aggregate value. These queries are given in Table 1 for the most common aggregate operators. Here, $bsum$ is the sum of all the values in a bucket, i.e., $(avg * count * (lo + \frac{sp*(count-1)}{2}))$.

It has been shown that the techniques presented here result in orders of magnitude faster execution than obtaining the exact answer [10].

[3]In our implementation, we make this scheme more efficient by running another simple query in the beginning to identify overlapping buckets in the histograms and then executing $Q1_\bowtie$ and $Q2_\bowtie$ for each pair of overlapping buckets.

distinct COUNT	SUM	AVG	MAX	MIN
SELECT SUM($count$) FROM H;	SELECT SUM($bsum$) FROM H;	SELECT $\frac{\texttt{SUM}(bsum)}{\texttt{SUM}(count)}$ FROM H;	SELECT MAX(hi) FROM H;	SELECT MIN(lo) FROM H;

Table 1: Queries to Compute Aggregate Values from Histograms

5 Quality of Answers

An important requirement of any system providing approximate answers is a quality measure on the answers and a way to estimate this measure when the actual answer is not available. Histograms can be supplemented with additional information in order to estimate the error in an approximate answer. Here, we describe the estimation of errors for data cube approximation; similar approach can be taken for relational approximation.

Consider a group-by query applying an aggregate operator on a sub-cube with dimensions $\mathcal{S}$. The error in estimating the result over a single group in the answer is the absolute difference between the actual and approximate aggregate values over that group. And, when the actual and approximate answers have the same set of groups, the error in answering the entire query is the sum of errors over all the groups in the result[4]. We describe a simple way to compute an *upper bound* on this error for the *sum* operator below. Similar techniques can be derived for other aggregate operations as well.

Let $\{ b_1, .., b_n \}$ be the set of buckets in the histogram used in answering the query. With each bucket b_i, we also maintain the maximum difference (m_i) between the actual and the average measured values aggregated on $\mathcal{S}$ in that bucket. Then, the maximum error contributed by bucket b_i to the query is simply $m_i \times MIN(n_i, t_i - n_i)$, where n_i is the number of values in the bucket that satisfy the query predicate and t_i is total number of values in the bucket. The total error in answering the query is computed by adding the contributions from all the buckets.

6 Histogram Selection for Data Cube Approximation

Recall that the data cube consists of a set of sub-cubes. An obvious approach for approximating the data cube is to build a histogram on each sub-cube. We call this the *direct* approach. However, this requires considerable amount of space because the number of histograms needed is exponential in the number of dimensions of the fact relation. Fortunately, since a histogram on a sub-cube $\mathcal{S}$ also contains information on any of its sub-sets, say $\mathcal{P}$, we can use a histogram on $\mathcal{S}$ to provide an approximation for $\mathcal{P}$ by projecting it onto the attributes in $\mathcal{P}$. We call this the *slicing* approach.

In view of the above alternatives for approximating a sub-cube, two natural questions arise: i) which histograms should be built and how much space should be allocated for each? and, ii) given a set of histograms computed on the data cube, which histogram should be used to answer a query on a sub-cube? First, we define two error metrics which are useful in answering these questions.

Definition 2: Let $\mathcal{S}'$ be an approximate sub-cube of the sub-cube $\mathcal{S}$. For a tuple t, let $A(t)$ be the measure value from $\mathcal{S}$, and $A'(t)$ its measure value from $\mathcal{S}'$. We define the *sub-cube error (E_S)* of $\mathcal{S}'$ as the average relative error between all the actual and the approximate measure values present in the sub-cube. That is[5],

$$E_S = \Sigma_{t \in \mathcal{S}} \frac{A(t) - A'(t)}{MAX(A(t), 1) \times |S|}$$

[4]When the answers contain different sets of groups, one can use the *MAC error* that we devised in [10] for capturing the difference between any two sets of numbers - in this case the actual and approximate answers. However, we are still working on techniques for estimating this error when the actual answer is not known.

[5]we divide by the maximum of 1 and the actual value to handle the case where actual result is 0.

Definition 3: The *data cube error* (E) is the *maximum* of the sub-cube errors of all the sub-cubes in that data cube.

$$E = Max_{\mathcal{S} \in 2^{\mathcal{U}}}(E_S)$$

Then, for a given set of histograms on a data cube, we use the histogram that approximates a sub-cube with the least sub-cube error for answering queries on that sub-cube. The allocation of space among the histograms is addressed next.

We refer to a particular allocation of space among histograms as a *histogram configuration*. We developed techniques to identify histogram configurations that minimize the space needed when the error is upper-bounded.

Definition 4: For a given upper bound ϵ on the data cube error E (Definition 3), the ϵ*-optimal* configuration is the configuration that requires the least amount of space while resulting in an error of at most ϵ.

Earlier research on optimization problems similar in nature to this problem has shown many of them to be NP-Hard [13, 7]. Though the complexity of our specific problem has not yet been established, there is a strong likelihood that it may not have an efficient solution. However, we have shown in [16] that the identification of the ϵ*-optimal* configuration is upper-bounded by the *minimum weighted set cover* problem (MWSC). There, we adapted the standard greedy algorithm used for computing an approximation to the minimum weighted set cover to provide an efficient heuristic for the current problem. The configuration generated by this algorithm requires an amount of space that is bounded within a factor of that required by the optimal configuration. This algorithm is described next.

We use the notion of a data cube lattice [7], which contains a vertex for each sub-cube and an edge from u to v if v is a subset of u. Each node v in the lattice is associated with a "marking;" v is marked to indicate that the sub-cube associated with it is not yet approximated with the required accuracy ϵ. To start with, no histogram is built on any sub-cube, and all nodes are marked. Next, we define some terms used in the algorithm. Let u and v be two nodes in the lattice and $e = \vec{uv}$ be a directed edge in the lattice.

- *ApproxError(*e, β*)*: This is the sub-cube error in approximating the sub-cube v using a histogram of β buckets on u. It is calculated by actually building a histogram with β buckets on the sub-cube u and computing the sub-cube error (Definition 2).
- *weight(e)*: This is the least value of β such that ApproxError(e, β) is less than or equal to ϵ.
- *benefit(e)*: This is the number of *marked* children of u that can be answered within an error ϵ by allocating $weight(e)$ buckets to the histogram on u. That is, the additional number of sub-cubes that can be approximated within ϵ due to the allocation of additional buckets.

The algorithm is given below.

Algorithm 6.1: GREEDY(double ϵ**)** $\{$
/* ϵ is the bound on the data cube error */
    for every edge e in the lattice **do**
        compute weight(e) by computing ApproxError(e, β) for increasing values of β.
    while (some of the vertices are still *marked*) **do** $\{$
        Pick an edge $e = \vec{uv}$ with the largest *benefit to weight ratio* and allocate a space of $weight(e)$ to u.
        Unmark all the children w of u with $ApproxError(uw, weights(e)) \leq \epsilon$.
        Update the benefits of the affected edges. /* Only those edges incident on v are affected by selecting e */
    $\}$
$\}$

---

7 Related Work

There has been significant amount of recent work on providing approximate answers to *aggregate queries* using precomputed statistics, such as, samples [1], histograms [17, 16], and wavelets [20]. Much of the work presented here on data cube approximation and histogram selection first appeared in [16]. Approximate answering of non-aggregate queries was addressed in [10], where a numerical measure for comparing set-valued answers was defined. Online aggregation [8], described earlier, constitutes another style of sampling-based approximate query answering wherein the answers are continuously refined till the exact answer is computed.

Histograms have been studied extensively for application in selectivity estimation in query optimizers [12, 14, 15]. In our earlier work, we have identified several novel classes of histograms to build on one or more attributes [19, 18] and also proposed techniques for their efficient computation [11] and incremental maintenance [5]. We recently extended histograms for selectivity estimation in spatial databases [2]. Much of the work presented in this article on query algebra for histograms has appeared in [10].

8 Conclusions

In this article, we have described various histogram-based techniques for providing approximate answers to aggregate queries. Histograms have been traditionally used for selectivity estimation. Their use for approximate query answering brings up several issues: efficient query processing, quality guarantees, and histogram selection being the three key issues addressed here. We have presented general solutions for all three issues, thus establishing the usability of histograms for this new problem.

References

[1] S. Acharya, P. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for improving approximate query answers. *Proc. of ACM SIGMOD Conf*, 1999.

[2] S. Acharya, V. Poosala, and S. Ramaswamy. Selectivity estimation in spatial databases. *Proc. of ACM SIGMOD Conf*, 1999.

[3] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA., 1989.

[4] M. Garey and D. Johnson. *Computers and intractability*. W. H. Freeman and Co., 1979.

[5] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. *Proc. of the 23rd Int. Conf. on Very Large Databases*, August 1997.

[6] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tabs, and sub-totals. *Proc. of IEEE Conf. on Data Engineering*, pages 152–159, 1996.

[7] V. Harinarayanan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. *Proc. of ACM SIGMOD Conf*, pages 205–216, 1996.

[8] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. *Proc. of ACM SIGMOD Conf*, 1996.

[9] Y. Ioannidis and V. Poosala. Balancing histogram optimality and practicality for query result size estimation. *Proc. of ACM SIGMOD Conf*, pages 233–244, May 1995.

[10] Y. Ioannidis and V. Poosala. Histogram-based techniques for approximating set-valued query-answers. *Proc. of the 25th Int. Conf. on Very Large Databases*, 1999.

[11] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. Sevcik, and T. Suel. Optimal histograms with quality guarantees. *Proc. of ACM SIGMOD Conf*, 1998.

[12] R. P. Kooi. *The optimization of queries in relational databases*. PhD thesis, Case Western Reserve University, Sept 1980.

[13] S. Muthukrishnan, V. Poosala, and T. Suel. On rectangular partitionings in two dimensions: Algorithms, complexity, and applications. *7th International Conference on Database Theory*, January, 1999.

[14] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. *Proc. of ACM SIGMOD Conf*, pages 256–276, 1984.

[15] V. Poosala. *Histogram-based estimation techniques in databases*. PhD thesis, Univ. of Wisconsin-Madison, 1997.

[16] V. Poosala and V. Ganti. Fast approximate answers to aggregate queries on a data cube. *International working conference on scientific and statistical database management*, 1999.

[17] V. Poosala and V. Ganti. Fast approximate query answering using precomputed statistics. *Proc. of IEEE Conf. on Data Engineering*, 1999.

[18] V. Poosala and Y. Ioannidis. Selectivity estimation without the attribute value independence assumption. *Proc. of the 23rd Int. Conf. on Very Large Databases*, August 1997.

[19] V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita. Improved histograms for selectivity estimation of range predicates. *Proc. of ACM SIGMOD Conf*, pages 294–305, June 1996.

[20] J. S. Vitter, M. Wang, and B. R. Iyer. Data cube approximation and histograms via wavelets. *CIKM*, pages 96–104, 1998.

The Role of Approximations in Maintaining and Using Aggregate Views

Daniel Barbará and Xintao Wu
George Mason University [||]
Information and Software Engineering Department
Fairfax, VA 22030 Email: {dbarbara,xwu}@gmu.com

Abstract

Techniques for approximating the data cube have been the object of much research lately. In this paper, we present a summary of Quasi-Cubes, a technique that allows to compress data cubes while keeping guaranteed bounds for the errors made when querying the approximate versions, regardless of the distribution of the underlying data. We also present a suite of applications that can benefit from using Quasi-Cubes, among them some in the area of data mining.

1 Introduction

Data cubes [9] are a widely used data abstraction for aggregates of multidimensional data. A cube is simply a multidimensional structure that contains in each cell an aggregate value, i.e., the result of applying an aggregate function to an underlying relation that consists of a series of attributes known as dimensions and one or more attributes known as measures. For instance, for a retail sales dataset with dimensions day, $store$ and $product$ and measure $sales$, each cell of the data cube would contain the total sales for a combination of the values of the dimensions. At the finest level of aggregation (known as the core cuboid), cells would contain sales for specific days, stores and product values, e.g., $January\ 1st$, $NewYork$, and $shoes$. At other levels of aggregation cells would contain combinations of proper subsets of the dimensions, e.g., total sales of $shoes$ in the $NewYork$ store (for all the "days"). The different levels of aggregation, or cuboid, form a lattice, such as the one presented in Figure 1, with the core cuboid being the one at the bottom of the lattice and the cuboid with the coarser level of aggregation (i.e., the one that contains a single cell with the total value of sales) at the top (ALL). Tools that implement data cubes allow users to ask a variety of queries whose answers are summaries of the data for a subset of the dimensions of the dataset (e.g., "what were the total sales of $shoes$ for the first quarter of 1999?").

As it is the case in most database systems, the tradeoff between space and time plays a crucial role in data cube implementations. If one materializes all the possible cuboids, in order to be ready for any query posed by the users, the space needed to store the materializations grows enormously, but the queries are answered very fast. If one does not materialize anything, but takes the "lazy" approach of computing the aggregates on demand, the

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[||]This work has been supported by NSF grant IIS-9732113

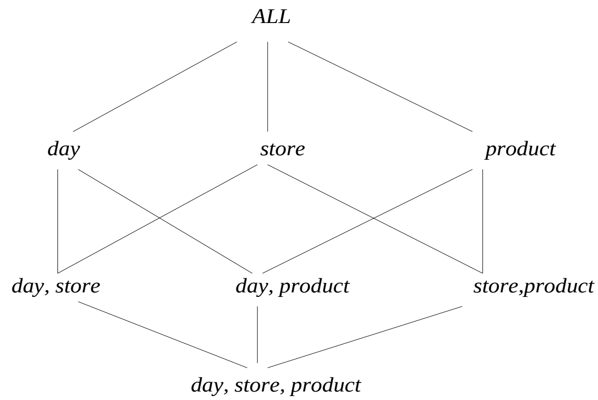

Figure 1: Lattice for the data cube with dimensions $day, store, product$.

demand for space is reduced considerably at the expense of long delays to get the answers. Data cubes can be prohibitively large: consider the previous example with 1,000 stores, 10 years (3,650 days) and 20,000 products. Materializing just the core cuboid would require storage for 73 billion aggregate values. Even if the data is sparse (not all aggregates are present) and only 1 % of the cells have values, we would still need space for 0.73 billion cells. If we add the cells for other cuboid and the fact that in many cases users want to see aggregates for all the elements on the hierarchy of dimensions (e.g., the values on the dimension day might be viewed by $month$, $quarter$, $year$ and so on), the space demands grow considerably. On the other hand, users of data cube technology demand fast answers to their queries, so they can quickly find areas of interest within the cube space and investigate trends and patterns in them. This tradeoff forces the designer to choose carefully which cuboids provide the most efficient choices for materialization. Strategies to select a subset of cuboids are presented in [11]. These techniques aim to control the space used in materialized cuboids while keeping the response time within acceptable limits.

Alternatively, researchers have begun to investigate ways to compress the data cube in such a way that only a fraction of the space taken by the whole cube is needed [1, 3, 4, 7, 20, 22]. Since the compression techniques are lossy, one can only provide approximate answers to the queries posed to the data cube. On the other hand, the queries can be answered without incurring into much disk I/O, so the response time is considerably smaller than the one experienced in uncompressed data cubes. Of all these techniques, ours [3, 4, 7] is the only one that allows the designer to establish guarantees for the errors incurred in the answers in such a way that the errors are kept below a predefined threshold, *regardless of the distribution of the data*. (All the other techniques can guarantee error levels, but they vary with the underlying data distributions and cannot be fixed by the designer.) We achieve that by insisting in retaining cells of the cube whose values, when recreated by the decompression algorithms differ from the real values in more than the tolerated threshold. That way, our compressed cubes, or *Quasi-Cubes* are composed by models used to estimate cells and a set of "outlier" values, or retained cells, that if estimated would incur in errors whose magnitude exceeds the predefined threshold.

In this paper we present a brief summary of our Quasi-Cube technique along with a list of possible applications that go beyond simply providing approximate answers to aggregate queries. Information about the project in which this paper is based can be found in [5]. The paper is organized as follows. In Section 2 we briefly summarize how Quasi-Cubes are implemented (more details can be found in [3, 4, 7]). Section 3 lists the applications enabled by this technology. Finally, Section 4 offers some conclusions.

2 Quasi-Cubes

We model regions of the core cuboid and employ these models to estimate the values of the individual cells. The reason to focus on the core cuboid is simple: the error guarantees for queries to the core cuboid hold for any other cuboid in the lattice. (In practice, the errors incurred when aggregating estimated cells of the core cuboid decrease dramatically because they tend to cancel each other.) As we stated in the introduction, to avoid incurring in large errors by the estimation, we retain all the cell values whose estimations are farther away from the real value by more than a pre-established threshold. This threshold becomes the guarantee of the approximate answer has. (As we will show later, many answers are, in reality closer to the real answer than what the threshold predicts.) We store the model parameters (for each modeled region of the cuboid) along with the retained cells to process the queries. The choice of models, although important because some models produce a better fit than others, is orthogonal to the Quasi-Cubes idea. We have experimented with two types of models: linear regression [4] and loglinear models [7]. (For a review of data reduction models see [2].)

The issues involved in compressing the cube, giving approximate answers to the queries and evaluating our technique can be summarized as follows:

- Selecting chunks of the core cuboid that will be described by models (regions of the core cuboid that are sufficiently dense). We do this by dividing the core cuboid space, usually starting with a regular grid and classifying the chunks into three types:
  1. Dense chunks, which are candidates for modeling.
  2. Empty chunks, i.e., sub-spaces that do not contain any data cells and can be discarded.
  3. Sparse chunks, which contain very few cells. Cells in these types of chunks are retained without attempting to model them.
- For each chunk to be modeled, computing the model parameters based on the data contained in the chunk and then, based on the estimated values compute for each non-zero cell in the chunk the estimation error and determine if the cell needs to be retained.
- Organizing the model parameters and retained cells to efficiently access them when processing queries.

Dividing the core cuboid in dense chunks is of outmost importance. Dense chunks render better-fitting models which in turn results in a higher compression rate. Moreover, finding enough dense chunks is the key to a good compression rate. Fortunately, most datasets in practice are skewed, i.e., the non-empty cells are clustered in regions of the core cuboid. However, finding these sub-spaces is far from trivial. So far, we have taken the approach of dividing the core cuboid in a regular grid, identifying potentially dense chunks, and subdividing those as needed until a sufficiently dense region is found. We are currently experimenting with techniques to "re-shuffle" the order of the rows in some (or all) dimensions to increase the density of regions. (For some type of dimensions, e.g., categorical attributes such as $product$ in our example, it is irrelevant in which order the values are kept, while for some others, such as $time$, the original order must be preserved.)

Computing the models is a step whose implementation depends, of course, on the choice of models. (Details of how to efficiently model chunks of data using linear regression can be found in [4]; for loglinear models, see [7]; for kernel estimations see [20], and for wavelets [22].)

The dense chunk descriptions are organized as blocks of data containing the model description, parameter values, outliers and and index (we index either non-empty cells or empty cells, depending on which set is smaller). These descriptions are smaller than the data they characterize and thus, many of them can be brought to main memory at once. Indexing of the chunks can help in quickly finding which chunks are needed to answer a query.

3 Applications

In this section we list the applications that are enabled by Quasi-Cubes.

3.1 Approximate query answering

Of course, this is the idea that motivated cube compression in the first place. A range query over the cuboid can be decomposed as the union of several disjoint queries, each spanning a chunk in the cuboid. That being the case, for each one of the disjoint sub-queries there are two possibilities:

- The sub-query completely includes its respective chunk. In this case, the answer of the sub-query can be immediately obtained from the chunk description, which also contains the aggregated value of all the cells in the chunk. Notice that this value is free of error and that the answer to this sub-query does not require retrieving any of the values in the retained list of cells for the chunk or estimating any of the cells values.
- The sub-query covers the respective chunk only partially. In this case, we need to estimate or retrieve (from the list of retained cells) the individual cell values and aggregate them.

In larger queries, i.e., aggregates over large portions of the core cuboid, many sub-queries span complete chunks and therefore, the running time is sped up considerably. Our experiments [7] show a considerable gain in execution time for approximate queries when compared with the same queries posed over the uncompressed cube.

3.2 On-line aggregation

On-line aggregation [10], is the process by which one can streamline the computation of more refined answers to the queries as the user is looking at the current estimate. The aim of this method is to reduce the latency of the query, by offering answers long before the final, correct answer is computed. With Quasi-Cubes, there is a relatively easy way to implement this: using different error thresholds, classify the data into *error bins*, each bin corresponding to an error level. This leads to a series of chunk descriptions, corresponding to each error lever. To produce the first estimate, only the data in the highest-error chunk description needs to be brought from disk; to refine the answer, successive bins are brought to memory.

Hellerstein et al. in [10], proposed to implement on-line aggregation by incrementally sampling the base relations that underlie the data cube. Other researchers [1] have shown that taking samples on the base relations leads to problems such as the lack of uniformity in the likelihood of cells appearing in the answer. The work in Join Synopses [1] attempts to remedy this by sampling the actual joins. However, a common problem with techniques that use sampling is that the samples must be refreshed periodically to avoid showing bias in the query answers. Our approach, on the other hand, does not exhibit any of these problems.

3.3 Supporting query answering in update periods.

While the data warehouse is being updated, the system is usually made unavailable for query answering. (Doing otherwise may create consistency problems in the answers or slow the updating considerably.) For that reasons, researchers [14, 17] have been investigating ways of reducing the window of time needed for performing updates in the warehouse. However, as organizations become more global, shutting down the querying system at any time of the day becomes a difficult proposition.

Quasi-Cubes can help in enabling approximate query answering while the actual data cube is being updated. The answers will, of course, be approximations of the state of the cube before the update, but for many cases this will be an acceptable tradeoff. Notice that since the Quasi-Cube description (chunk descriptions) can be maintained decoupled from the actual data in the cube, the users will be accessing a separate database while the warehouse is being updated. (Of course, this application would not support data cube compression and it assumes that space is not a problem.) The Quasi-Cube descriptions need, of course, to be updated (we have shown inexpensive ways to do this in [7]), but this can be done after the warehouse update has finished (then the query processing can be redirected back to the real cube).

3.4 Lighter Data Marts

The chunk descriptions can be made to map areas of the core cuboid that correspond to data which is of interest to a particular division in the organization. These sub-cubes are commonly known as Data Marts, and many commercial tools made provisions to export the Data Mart to remote clients. Using the Quasi-Cube technology, these Data Marts can be made extremely "light," with respect to the space they occupy in the remote client at the expense of providing only approximate views of the relevant data. (More refined views can always be obtained by going to the data warehouse.) This idea is very much in line with the opinion of some data mining experts who claim that the trend should be in exporting models and not data to the clients [8].

3.5 Approximate Data Mining

The ultimate goal of building a data cube is to extract knowledge from the data stored in it. To this end, we are currently investigating data mining techniques that can benefit from the usage of Quasi-Cubes. The idea is to perform mining in the compressed form of the data, thereby getting the mining task to complete quickly at the expense of having approximate results. We present here two examples of techniques that we are actively investigating.

3.5.1 Exploratory Data Analysis

Exploratory Data Analysis (EDA) [12, 13, 21] is a suite of widely used techniques that aim to determine which factors have the most influence on data values in a multi-way table, or which cells in the table can be considered anomalous with respect to the other cells. EDA is performed without any a-priori hypothesis in mind: rather it searches for "exceptions" of the data values relative to what those values would have been if anticipated by an statistical model. This statistical model fits the rest of the data values rather well and can be accepted as a description of the dataset. When applied to multidimensional tables, EDA also uncovers the attribute values that have the greatest effect on the data points. As pointed out by [19], these exceptions can be used by an analyst as starting points in the search for anomalies, guiding the analyst work across a search space that can be a very large.

At any level of aggregation, data cubes can be viewed as *multi-way tables*, that can be subjected to EDA. Two traditional ways of performing EDA in tables are *median polish* and *mean polish* [12]. Both methods try to fit a model (additive or multiplicative) by operating on the data table, finding and subtracting medians (means) along each dimension of the table. Starting with one dimension, the method calculates the median (mean) of each "row" [1] and subtracts this value from every observation in the row. Then, the method moves to the next dimension and uses the table resulting from the previous operation to find medians (means) in each row and subtract them from the entries in this row. Of course, doing that changes the medians (means) on the previous dimension rows. The process continues with each dimension and iteratively cycles through the set of dimensions. In principle, the process continues until all the rows in each dimension have zero median (mean), hence the number median (mean) polish. (In practice, the process stops when it gets sufficiently close to that goal.) One gets two kinds of information from this process. First, one gets the *effect* that each row in each dimension has on the model, given by the algebraic sum of the medians (means) that have been subtracted in that row at every step. Secondly, one

[1] we use the term "row" to refer to the set of cells in the hypercube which share the same attribute value for one of the dimensions

gets *residuals* in each cell of the table, which tell us how far apart that particular cell is from the value that would have been predicted by the model being fit.

It has been pointed out that the main drawback of the mean polish method is its lack of resistance to outliers (i.e., cells that do not fit the model well). This lack of resistance manifests itself specially when “holes,” e.g., missing cells, are present [12]. This is particularly troublesome for data cubes, which are usually sparse (i.e., not every cell has a value). On the other hand, median polish, being more resistant to outliers and holes, is very adversely affected by holes which increase substantially the number of iterations needed by the process [12]. Increasing the number of iterations can drastically impose enormous I/O demands (by requesting many passes over a large dataset) and therefore render the process impractical for large data cubes. Previous work in using EDA for data cubes [19] has chosen to use mean polish, precisely for being less demanding on the number of iterations needed to finish.

We have implemented a method of performing median polish in a Quasi-Cube which uses the chunk descriptions instead of the cube data to perform the polishing of the medians. The results, reported in [6] show that the quality of the results obtained by this method is extremely good, while at the same time the method outperforms the running time of doing median polish on the data itself, allowing the user to perform EDA in cases where the cost of using the standard procedure (over the real data) would be prohibitive.

We are currently investigating extensions of this work such as the obtaining joint influences of more than one attribute value and performing non-parametric tests in the data [23].

3.5.2 Decision Trees

Decision trees [15] are a classification technique, used to forecast membership for a multidimensional data point. For instance, a decision tree could be trained to identify “risky” and “good” customers from multidimensional descriptions of people who apply for loans. Nodes in a decision tree involve testing a particular attribute, usually comparing the attribute value with a constant. The decision tree needs to be trained by using a data set in which each multivariate record has a label indicating the class it belongs to. The classical implementation of decision trees (C4.5 [16]) is meant to be used with memory-resident data sets. An implementation, named SPRINT, designed for large data sets is reported in [18]. SPRINT uses a combination of clever data structures that make it possible to manage disk-resident data sets. These data structures have to be recreated at every step of the training phase, using the disk-resident data set. SPRINT can still experience a large I/O activity for large data sets, due to the need of swapping the tables in and out of main memory. We are currently investigating the use of our compression technique to make the decision tree training step scale with large sets. The idea is to drive algorithms such as SPRINT not with the data set itself, but with compressed versions of it. The class or label attribute becomes a dimension of the Quasi-Cube. Once that is done, the decision of what is the question to be asked in the next node of the tree can be answered (approximately) by using the chunk descriptions, instead of going to the data itself (like SPRINT does). A caveat, though, is that as one goes deeper in the tree, the queries that are needed to fill the tables become more and more fragmented, using a larger number of chunk descriptions (i.e., they partially intersect a large number of chunks in the core cuboid). When the time to bring lots of chunk descriptions (I/O time) and estimate cells (CPU time) exceeds that of using the data in the cube (mostly I/O), we need to switch to use the cube data. So, at some level in the creation of the tree, one needs to switch from driving SPRINT from the Quasi-Cubes to driving it from the actual data, and the decision must be done solely on the basis of the increase in chunk processing time. Nevertheless, we expect a significant improvement on the running time of the algorithm due to the effect of the upper levels of the tree (which actually need the bulk of the data to be fetched, since as we go down the tree, the queries become more and more restrictive).

4 Conclusions

We have presented in this paper an overview of Quasi-Cubes, a technique that allows the designer of data cubes to compress the data and still maintain guaranteed bounds for the errors made in querying the cube. We have also presented a list of applications that can substantially benefit from the use of approximations.

References

[1] S. Acharya, P.B. Gibbons, V. Poosala, and S. Ramaswamy. Join Synopses for Approximate Query Answering. In *Proceedings of the ACM SIGMOD Conference, Philadelphia, PA*, June 1999.

[2] D. Barbará, W. DuMouchel, C. Faloutsos, P.J. Haas, J.M. Hellerstein, Y. Ioannidis, H.V. Jagadish, T. Johnson, R. Ng, V. Poosala, K.A. Ross, and K.G. Sevcik. The New Jersey Data Reduction Report. *Data Engineering Bulletin*, 20(4):3–45, December 1997.

[3] D. Barbará and M. Sullivan. Quasi-Cubes: A space-efficient way to support approximate multidimensional databases. Tech. Report, Dept. of Information and Software Systems Engineering, George Mason University, 1997.

[4] D. Barbará and M. Sullivan. Quasi-cubes: Exploiting approximations in multidimensional databases. *SIGMOD Record*, 26(3), September 1997.

[5] D. Barbará. Quasi-Cubes: Exploiting Approximations in Multidimensional Data Sets. http://www.ise.gmu.edu/ dbarbara/quasi.html.

[6] D. Barbará and X. Wu. Using Approximations to Scale Exploratory Data Analysis in Datacubes. In *Proceedings of the 1999 ACM SIGKDD Conference, San Diego, CA*, August 1999.

[7] D. Barbará and X. Wu. Using loglinear models to compress data cubes. Tech. Report, Dept. of Information and Software Systems Engineering, George Mason University, 1999.

[8] U. Fayyad. Data mining techniques. Tutorial at Very Large Databases Conference, New York, 1998.

[9] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. In *Proceedings of the Intl. Conference on Data Engineering, New Orleans*, 1996.

[10] J.M. Hellerstein, P.J. Haas, and H.J. Wang. Online Aggregation. In *Proceedings of the ACM SIGMOD Conference, Tucson, Arizona*, May 1997.

[11] V. Harinarayan, A. Rajaraman, and J.D. Ullman. Implementing Data Cubes Efficiently. In *Proceedings of the ACM-SIGMOD Conference, Montreal, Canada*, 1996.

[12] D.C. Hoaglin, F. Mosteller, and J.W. Tukey. *Exploring Data Tables, Trends and Shapes*. Wiley, 1985.

[13] D.C. Hoaglin, F. Mosteller, and J.W. Tukey. *Understanding Robust and Exploratory Data Analysis*. Wiley, 1986.

[14] W. Labio, R. Yerneni, and H. Garcia-Molina. Shrinking the Warehouse Update Window. Tech. Report, Computer Science Dept., Stanford University.

[15] J.R. Quinlan. Introduction of decision trees. *Machine Learning 1(1): 81-106*, 1986.

[16] J.R. Quinlan. *C4.5: Programs for machine learning.* Morgan Kaufmann, San Francisco, 1993.

[17] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and Bulk Incremental Updates on the Data Cube. In *Proceedings of the ACM SIGMOD Conference, Tucson, Arizona*, May 1997.

[18] J.C. Shafer, R. Agrawal, and M. Mehta, SPRINT: A Scalable Parallel Classifier for Data Mining. In *Proceedings of the Very Large Databases Conference, Mumbai (Bombay), India*, Sept. 1996.

[19] S. Sarawagi, R. Agrawal, and N. Meggido. Discovery-driven Exploration of OLAP Data Cubes. In *Proceedings of Conference on Extending Data Base Technology*, pages 168–182, 1998.

[20] J. Shanmugasundaram, U. Fayyad, and P.S. Bradley. Compressed Data Cubes for OLAP Aggregate Query Approximation on Continuous Dimensions. In *Proceedings of the ACM-SIGKDD Conference, San Diego, CA*, August 1999.

[21] J.W. Tukey. *Exploratory Data Analysis*. Addison Wesley, 1977.

[22] J.S. Vitter and M. Wang. Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets. In *Proceedings of the 1999 ACM-SIGMOD Conference, Philadelphia, PA*, June 1999.

[23] T.H. Wonnacott and R.J. Wonnacott. *Introductory Statistics*. Wiley, 1990.

Some Approaches to Index Design for Cube Forests

Theodore Johnson
johnsont@research.att.com
AT&T Labs – Research

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

Abstract

Data cubes greatly facilitate data analysis by allowing the simple expression and fast evaluation of common data analysis queries (roll up, drill down, slice by, etc.). Data cubes are most useful as an interactive data analysis tool, a fact that has motivated considerable data cube storage research. However, loading new data into some of these storage structures can be very expensive. We have developed a storage structure, cube forests*, that stores the data cube in one or more indices. In previous papers, we have discussed the design and querying of a cube forest. In this paper, we describe an implementation and a technique for batch updates.*

1 Introduction

Data in a warehouse is often viewed as multidimensional [2]. The attributes of a relation are classified as either dimension or measure attributes. The dimension attributes classify the object that the tuple represents, while the measure attributes represent the unique properties of the object. Often, several attributes represent successively refined classifications of the object. These attributes are considered to be a single hierarchical dimension.

A common relational representation of multidimensional data is the star schema. Each dimension is represented by a separate dimension table, and contains the unique values of the dimension. The warehoused data is represented by a central fact table, which contains foreign keys that link the fact table to the dimension tables, and also the measure attributes.

Star and snowflake schemas are useful for extracting individual tuples from a data warehouse, but often users prefer to browse and analyze summaries of the date. A convenient method for representing these summaries is by a *data cube*, i.e., a collection of aggregates at all levels of granularity [6]. The name "data cube" is derived from a way to visualize the multidimensional aggregate, e.g. a 3-dimensional data cube.

A data cube on d dimensions is defined by 2^d group-by queries. A special value, ALL, is used to represent the full cube and all subcubes in a single table. A typical query on a cube specifies a subcube to query, and a restriction on the range on the dimensional attributes. For example, one might ask for weekly sales in New Jersey of clothing during 1998. The cube structure encourages users to explore interesting features of the data. For example, a user might *roll up* the query by asking for sales in the United States, or *drill down* by asking for daily sales.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

† Work partly supported by NSF grants IRI-9224601 and IRI-9531554

1.1 Data Cube Storage

Data cubes have become popular in part because they promise fast interactive results. As we have defined the data cube, all results have been computed, so a query is often just a matter of retrieving the data from the cube. However, the data cube might become very large, increasing storage costs and also data retrieval times. Alternative data cube storage architectures have been proposed to speed up access times and/or reduce storage costs.

1.1.1 ROLAP

One approach to storing a data cube is to use a conventional relational DBMS. This type of database is termed a *ROLAP* database (Relational On-Line Analytical Processing). The multidimensional view of the data may be on the fact table (which is often stored in order to answer ad-hoc queries), or on the most refined collection of aggregates in the data cube (i.e., grouped by every dimension), the *base subcube*. These tables usually have many indices built on them, e.g., one for each dimension.

One problem with a ROLAP approach is that the base table can be very large, and user queries are often at a relatively high level of aggregation. In [3], the author gives an example of a six dimensional database whose base table contains 122 million rows and occupies 17 Gbytes. Many aggregation queries would require accessing the entire table (because the indices are not clustered), and take a very long time to complete.

1.1.2 Molap

The highly structured nature of a multidimensional database suggests that a specialized storage structure might have better performance than a ROLAP database. In [6], the authors suggest storing base subcube data in a multidimensional array. However, the multidimensional array representation of a data cube has two weaknesses. First, good performance generally depends on the array being main-memory resident. Sarawagi and Stonebraker [12] evaluate a collection of techniques for efficient secondary storage of large multidimensional arrays. The basic problem is to chop the array into *chunks*, which are small subarrays that can fit into a block of memory.

Another weakness of the multidimensional array representation of a data cube is that the base subcube is likely to be *sparse*. One approach is to create a hybrid storage structure that uses multidimensional arrays wherever the data is dense. In [3], Colliat described the storage structure of the Essbase multidimensional database. Essbase engineers observed that in many data sets, some collections of dimensions are dense (every cell is filled) while other collections of dimensions are sparse (only a few combinations of the sparse dimensions exist in the fact table). Essbase uses the sparse attributes for indexing, and stores the dense attributes and the aggregates values in a multidimensional array. The index on the sparse dimensions is usually small enough to be memory resident, and the multidimensional arrays are usually compact enough to be fetched with one disk read. Another hybrid array based storage structure divides the multidimensional array into chunks, but stores the chunks differently depending on whether they are dense (e.g., 40% or more of the cells are filled) or sparse [5, 13].

1.1.3 Hierarchical Storage

In addition to ROLAP and MOLAP data base storage, alternative structures have been proposed, which we summarize here. Each makes use of hierarchical decompositions of the data in some way.

In [8], Ho et al. propose structures for computing range aggregates over multidimensional data . They assume that the dimensional attributes are (or can be mapped to) contiguous integers, and that the base subcube is dense. At every cell in the multidimensional array, they compute the prefix sum over the base subcube. By the principle of inclusion-exclusion, any range sum can be computed in $O(2^d)$ time, for d dimensions.

A disadvantage of the prefix sum approach is that a single update to the data cube requires that a very large number of prefix sums be updated. In [8], a hierarchical structure is proposed to reduce the extent of this problem. More aggressive approaches are presented in [1, 4].

Roussoupolos et al. propose an R-tree based method for data cube storage, which they term the *cubetree* [11, 10]. They extend the attribute range of each dimension with an ALL attribute, and store the aggregates in an R-tree. The method of packing an R-tree can have a significant effect on performance.

2 Cube Forests

In [8, 9], Johnson and Shasha describe an index structure that stores the full data cube directly in a search structure. To simplify the discussion, suppose our data is a single denormalized table whose attributes come from d dimensions denoting orthogonal properties (e.g., as above, product, location, time, organization, and so on). Each dimension c is organized hierarchically into n_c *dimensional attributes* $A_{c1}, A_{c2}, ..., A_{cn_c}$ where A_{c1} is the most general attribute (e.g., continent) and A_{cn_c} is the most specific (e.g., school district). Thus, the denormalized relation looks like this: $R(A_{11}, A_{12}, ..., A_{1n_1}, A_{21}, A_{22}, ..., A_{2n_2}, \ldots, A_{d1}, A_{d2}, ..., A_{dn_d}, value\_attributes)$. Here the value attributes are those to be aggregated, e.g., sale price, cost, value added, or whatever. This relation is denormalized because $A_{ij} \rightarrow A_{ik}$ when $j > k$, thus violating third normal form. (The key is $A_{1n_1} A_{2n_2} ... A_{dn_d}$.)

We first present *cube forests*, in which every dimension consists of a single attribute, and then present the *hierarchically split cube forest*, in which dimensions are hierarchies of attributes. While our presentation uses a single aggregate value, our structure applies unchanged to multiple aggregates over multiple values.

2.1 The Basic Structure

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

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

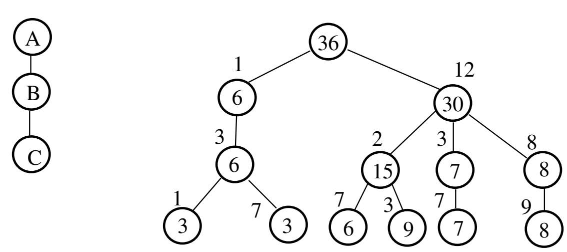

Figure 1: A cube tree template and its instantiation. Circled numbers represent the sum of V (i) over the entire relation (at the root, depth 0), (ii) for particular A values (depth 1), (iii) for particular AB combinations (depth 2), (iv) for particular ABC combinations (depth 3 or leaf level). Note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

We note that the index in our example is "cooked" to simplify its presentation. Since the relation is small, we can represent an attribute in the template as one level in the index. For a large relation, we need to define a strategy for implementing an index structure based on the template. For this section, we will assume that each attribute in the template corresponds to a separate index (for example, a B-tree). In Section 4, we present details of an implementation.

An interior node may have several children. In this case, each entry in a leaf of an index in the instantiation corresponding to the node has a pointer to a subindex for each template child of the node. This second feature leads to a tree topology for the template as shown in the left side of Figure 2.

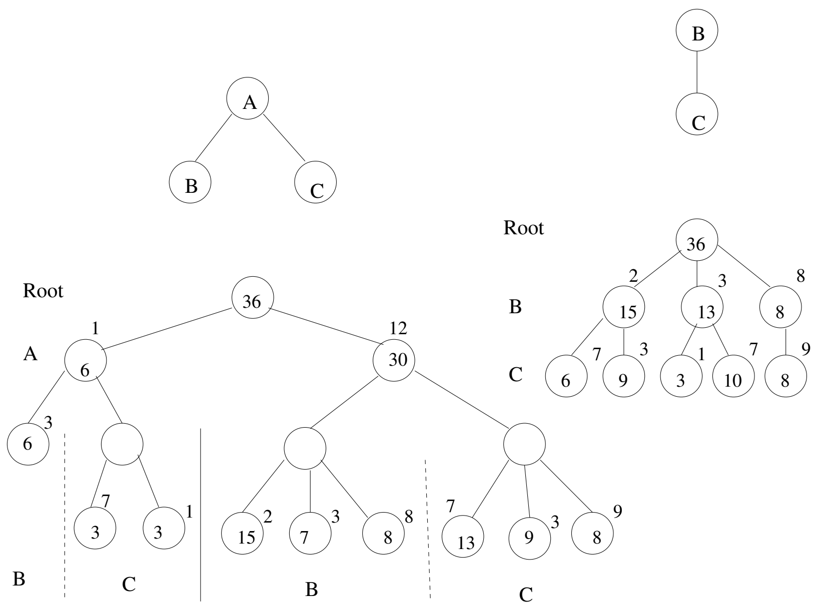

Figure 2: A cube forest template and its instantiation. Note that each A value in the instantiation of the left template has both B children and C children.

2.2 Full Cube Forests

We would like to be able to construct a cube forest in which any point query can be answered by searching for a single node. A *full cube forest* for d dimensions is defined recursively, using two copies of the full cube forest for $d-1$ dimensions. An example of a full cube forest is shown in Figure 3.

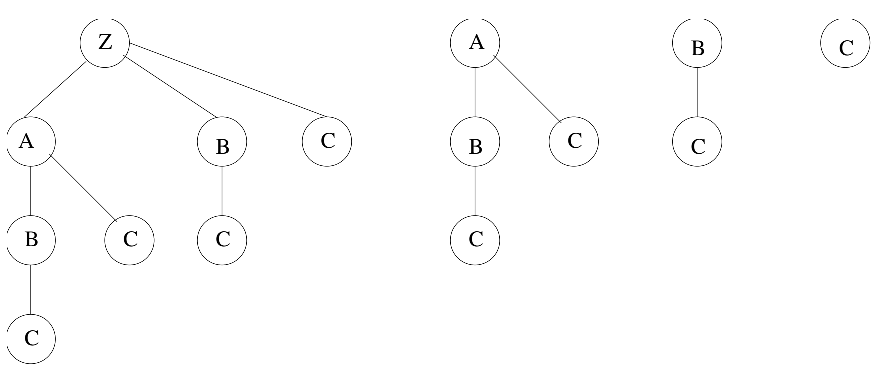

Figure 3: A full cube forest on the four attributes Z, A, B, C.

3 Hierarchically Split Cube Forests

Each dimension in the data cube can be hierarchical. For example, the data dimension For example, the date dimension can be specified by year, month, or day with a one-to-many relationship from year to month and month to day. Instead of treating each attribute as a separate dimension, we define a full *h-split forest* by "splitting" the dimension nodes the full cube forest. An example is shown in Figure 4. This construction may seem to be inefficient, but the full h-split forest has far fewer nodes and takes much less space than a full cube forest that treats all attributes as orthogonal.

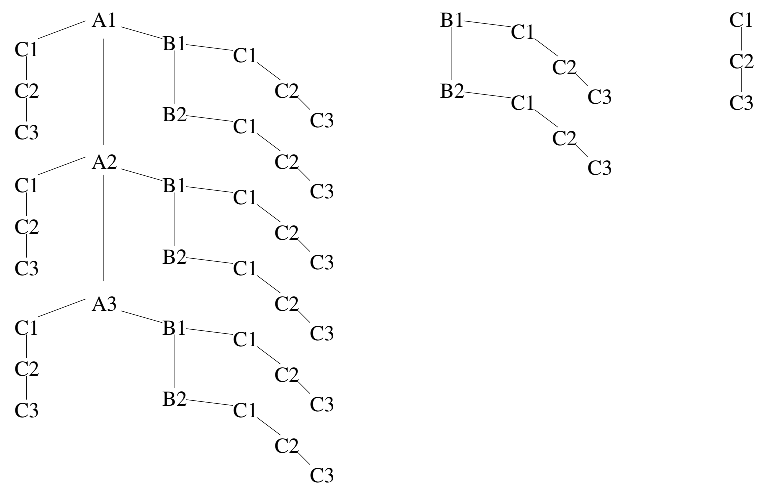

Figure 4: Template for a hierarchically split cube forest on 3 dimensions (A, B, and C).

Given a collection of d dimensions, there are $d!$ possible cube forests. Furthermore, these cube forests can be *pruned* (eliminating branches of easily computed aggregates) to reduce storage and update costs. Cube forest design is discussed in detail in [8, 9].

4 Cube Forest Data Structures

An efficient implementation of a cube forest requires tight integration between the forest structure and the update and query algorithms. In this section, we describe the data structures used to implement a cube forest.

The design of the cube forest data structures is as follows. Suppose that we are given cube forest template F. We design an index on each tree $T \in F$. Given a h-split tree template T, we choose a path from the root of T to be the *spine* of the tree. The spine defines a composite index, the keys of which are the attributes of the nodes in the spine concatenated together. In our examples, the spine is the longest root-to-leaf path in the tree. The spine partitions the h-split template, creating several subtrees. A spine is found for each subtree, and the process continues until all template nodes are in some spine.

Suppose that an index instantiates a spine on attributes $(A_1, A_2, \ldots, A_n)$. For every key $(a_1, a_2, \ldots, a_n)$ that is inserted into the tree, let us define the ith subkey, denoted sk_i, to be the prefix consisting of $(a_1, a_2, \ldots, a_i)$. If the template node corresponding to A_i has children other than A_{i+1}, then we need to associate a set of subtree pointers with subkey sk_i. If the node corresponding to A_i is not aggregate pruned, we need to associate an aggregate value with sk_i. We define an *effective leaf* for subkey $sk = (a_1, \ldots, a_i)$ to be the place in the index where information associated with the subkey (subtree pointers and aggregate values) is stored. The index invariant that defines the location of an effective leaf for sk should hold true at a single place in the index, and this place should be on the search path to any key k whose subkey is sk.

We build our spine index from a B-tree. We place an effective leaf for subkey sk at the highest level in the B-tree (closest to the root) where sk is a subkey of a separator[1] in a node, or a key in a leaf (to simplify the discussion, we will regard keys in leaves as separators). If there is more than one such separator, we place the effective leaf at the rightmost separator whose prefix is sk (one can also use the leftmost separator, the logic will be symmetric). This definition of the location of an effective leaf ensures that any insert operation for a key whose prefix is sk will encounter the effective leaf on the path to the leaf. Effective leaves might move during restructuring (as we will describe), but all such movement is local (i.e., some effective leaves will migrate from the split node and its new sibling to the parent, and some effective leaves in the parent might move). So, we are assured that inserts can be performed efficiently.

An example is shown in Figure 5. The tree template contains attributes A, B, C, and D. We create a spine (indicated by the solid line connecting template nodes) on (A, B, C). The template node D is in a separate partition, and its spine is simply (D). Since the edge $A - D$ is not a spine edge, we draw it as a dashed line. To the left, we show the resulting index on a small sample of data. Note that the key for the index consists of the values of the attributes in the spine catenated together.

To avoid clutter, we do not show where the aggregates are stored, but we illustrate the location of effective leaves by identifying where the pointers to indices on D are stored. The two separator keys in the root are effective leaves for the subkeys $a = 2$ and $a = 4$, respectively. We note that these separator keys are also effective leaves for the subkeys ($a = 2, b = 7$), ($a = 2, b = 7, c = 3$), ($a = 4, b = 3$), ($a = 4, b = 3, c = 14$), and the required aggregate information is stored in those effective leaves. The leftmost child of the root has an effective leaf for $a = 1$ in its middle entry (i.e., the rightmost separator key with subkey $a = 1$). The rightmost entry in this leaf is not an effective leaf for the subkey $a = 2$, because this effective leaf can be found in the parent. Note that an insert of a key with subkey $a = 2$ might be directed to this leaf, or to its right sibling. In either case, the insert operation will encounter the effective leaf for $a = 2$, which is stored in the parent node.

The reader will notice that even in Figure 5, there is a significant amount of redundancy in the keys. Simple prefix compression techniques can lead to significant space savings in the spine trees.

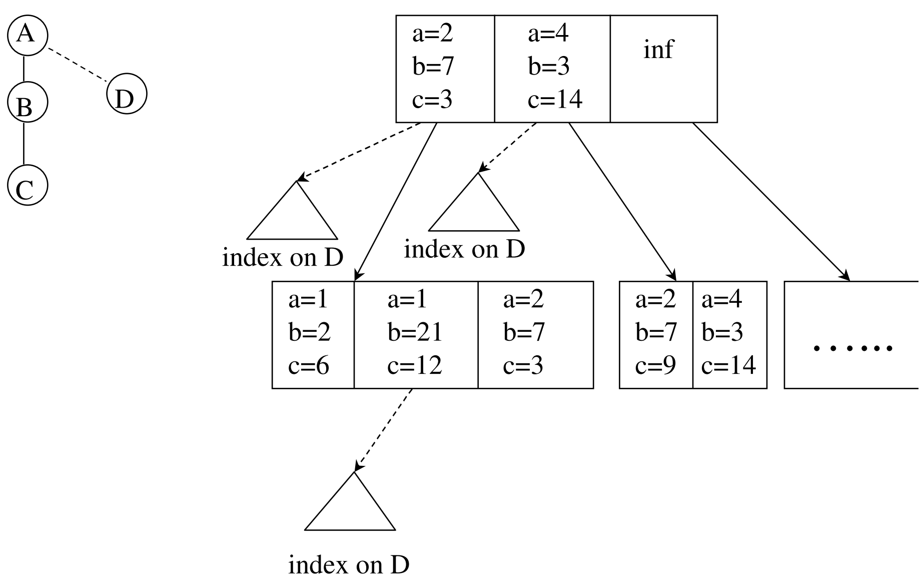

Figure 5: A B-tree index that supports a h-split tree.

[1] The ith separator in a B-tree node is a key that indicates which keys can be found the $i - 1$st subtree as opposed to the ith subkey.

4.1 Insertion Algorithm

The algorithm for inserting a key value into the index uses the B-tree insertion algorithm. However, two additional tasks must be performed – updating aggregate values at all effective leaves whose subkeys are prefixes of the insertion key, and preserving the effective leaf location invariant. We will assume that an effective leaf for subkey sk is located in the highest node n in the tree where a separator key k with subkey sk appears, and at the rightmost such separator if more than one separator has subkey sk in n.

The first task is to update all aggregates associated with a subkey on the descent into the index. We define the following variables:

n	Number of key values that comprise the aggregate key.
curr_eff_leaf	Length of the subkey of the next effective leaf to match
r	record that is being inserted into the cube forest.
key	Aggregate key derived from r for the index.
v	Value attributes of r.
node	Index node currently under examination.
node.num_children	Number of separator keys in this node.
node.separator[i]	key value that separates key values in subtrees i and $i+$.
node.eff_leaf[i][j]	Data associated with the effective leaf whose subkey is the length j prefix of node.separator[i]. This includes aggregate values and subtrees corresponding to subtree branches of the jth node of the spine (if any).
node.child[i]	Child node of node.

We assume that all keys stored in subtree node.child[i] have key values less than or equal to node.separator[i] (leaf nodes do not include the array node.child[..]). In the code fragment, we assume that we have already determined that node.separator[k] is the separator with the smallest value larger than or equal to key. If node is a non-leaf node we navigate to node.child[i], otherwise we insert key into the index if key is not equal to node.separator[k].

The code fragment can be summarized as follows. Let sk_i be the subkey of key of length i. To ensure that we find the highest node in the index where sk_i occurs, we search for as many subkeys of key as possible at every node in the search path. We do not need to match previously found subkeys, and if sk_i does not appear in the node, then sk_{i+1} does not appear in the node. If $\mathsf{sk}_{\mathsf{curr\_eff\_leaf}}$ appears in the node, then it will be a subkey of at least one of node.separator[k-1] and node.separator[k]. If the prefix of node.separator[k] is $\mathsf{sk}_{\mathsf{curr\_eff\_leaf}}$, then the effective leaf curr_eff_leaf is located at the highest index j such that the prefix of node.separator[j] is $\mathsf{sk}_{\mathsf{curr\_eff\_leaf}}$. Otherwise, if the prefix of node.separator[k-1] is $\mathsf{sk}_{\mathsf{curr\_eff\_leaf}}$, then the index of the effective leaf curr_eff_leaf is k-1. If we find the effective leaf, we update associated aggregate with the value attributes, and we insert r into each subtree associated with the effective leaf.

Because of pruning, a subkey of length i might not have any information associated with it, and thus will have no effective leaf. In this section, we will search for the effective leaf of sk_i, but any operations on the effective leaf are null, and no effective leaf is actually stored. This convention simplifies the presentation of the algorithm.

When we reach the leaf, l, we insert an entry for key into l only if key is not already present in l. If we insert an entry for key in l, we need to attach all effective leaves located at the entry for key. These are all of the effective leaves numbered curr_eff_leaf through n. Initializing the effective leaf consists of initializing the aggregate value with v (if any), creating all subindices of the effective leaf (if any) and inserting r into them. Finally, we must preserve the invariant that effective leaves are located at the rightmost position where the separator contains the subkey. So for all subkeys of key of length less than curr_eff_subkey, we need to search for effective leaves to the left of the position where key was inserted. If we find effective leaves for subkeys of key, we transfer their location to the location of key. We can speed up the search process by observing that the effective leaf for sk_{i-1}

```
done = false;
while(not done){
    found_effleaf = false
    if( key matches node.separator[k] on the subkey of length curr_eff_leaf){
        found_effleaf = true
        set efl_idx to be the largest index j such that key matches node.separator[j] on the
         subkey of length curr_eff_leaf
    }else{
        if( (k > 0) and (key matches node.separator[k-1] on the subkey of length curr_eff_leaf){
            found_effleaf = true
            efl_idx = k-1
        }
    }
    if(found_effleaf is true){
        add v to the aggregate value stored in node.eff_leaf[efl_idx] (if any)
        For every subtree st in node.eff_leaf[efl_idx]
            insert r into st
        curr_eff_leaf ++
        if( curr_eff_leaf > n){
            done = true
        }
    }else{
        done = true
    }
}
```

Figure 6: Find all effective leaves that match key in this node.

must be located at or to the left of the location of the effective leaf for subkey sk_i, and if the effective leaf for sk_i does not exist in l, then the effective leaf for sk_{i-1} does not exist in l.

If a key insertion causes a node to split, the usual restructuring operations are performed. In addition, the effective leaf location invariant must be maintained. The key value that is promoted to the parent is the rightmost separator in the left sibling of the pair of nodes that results from the split. All effective leaves are promoted along with the separator (key) value. If there is an effective leaf for subkey sk_i of the promoted separator in the right sibling, then it should be promoted to the newly inserted separator. The optimizations for finding these separators are the opposite of those described in the previous paragraph (i.e., substitute "right" for "left"). The separator that is inserted into the parent node might now contain a rightmost sk_i for an effective leaf in the parent. These effective leaves must be moved to the newly inserted key, and the optimizations for performing this migration are the same as that described in the previous paragraph.

5 Cube Forest Batch Update Algorithms

Our batch update algorithm for inserting tuples into the forest works as follows. Given a set of tuples to insert, we partition the tuples into subsets that can be processed in main memory (perhaps only one partition). For each partition, we insert the tuples of the partition into each index that comprises the forest.

To insert a batch of tuples into an index, we first sort it on the spine attributes of the index using an in-memory sort. For each tuple in the batch, we descend the index until the last effective leaf is found. We do not perform any processing with the effective leaves that are found, instead we make note of the location of the effective leaves. If necessary, we insert the spine key (and initialize any new effective leaves) and perform restructing. If an effective leaf for the recently inserted key changes location during restructuring, we note its new location. We compare every subkey of the recently inserted tuple against the corresponding subkey of the next tuple in the batch. If a subkey of the current tuple matches the subkey of the next tuple, we defer the update of the aggregates and the non-spine subtrees (if any). Otherwise, we perform all deferred aggregate updates for the associated effective leaf, and for all of its subtrees. We perform the subtree updates by a recursive call to the batch insert algorithm.

6 Experimental Results

We implemented a prototype h-split forest to test the practical value of our ideas. We tested the implementation using a 600,000 tuple database derived from the TPC-D benchmark. Building the unpruned largest index in the cube forest index by inserting tuples one at a time requires more that 5,000,000 I/Os. However, the batch insert algorithm (using 30 memory buffers and a page size of 1K bytes) needs only 960,000 I/Os with a batch size of 10,000 tuples (400,000 I/Os with a batch size of 100,000 tuples), and occupies 62 Mbytes of storage.

7 Conclusions

Cube forests are an attractive alternative for the storage and indexing of a data cube. In a full cube forest, point cube queries of the form "Find the sales of all Mustangs in New England in the first quarter of last year" is answered in a single index lookup yielding sub-second response time. We have presented an implementation for cube forests based on B-tree indexing. We showed that the implementation has efficient batch update algorithms.

References

[1] C.-Y. Chan and Y. Ioannidis. Hierarchical cubes for range-sum queries. In *Proc. 25th Intl. Conf. Very Large Data Bases*, pages 685–686, 1999.

[2] S. Chaudhuri and U. Dayal. On overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, 1997.

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

[4] S. Geffner, D. Agrawal, A. El Abbadi, and T. Smith. An efficient approach for querying dynamic data cubes. In *Proc. IEEE Intl. Conf. Data Engineering*, pages 328–335, 1999.

[5] S. Goil and A. Choudhary. An infrastructure for scalable parallel multidimensional analysis. In *Proc. 11th Intl. Conf. Scientific and Statistical Database Management*, pages 102–111, 1999.

[6] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proc. IEEE Intl. Conf. on Data Engineering*, pages 152–159, 1996.

[7] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *Proc. ACM SIGMOD Conf.*, pages 73–88, 1997.

[8] T. Johnson and D. Shasha. Hierarchically split cube forests for decision support: description and tuned design. Dept. of Computer Science tr727, New York University, www.cs.nyu.edu, Nov. 1996.

[9] T. Johnson and D. Shasha. Some approaches to index design for cube forest. *IEEE Data Engineering Bulliten*, 20(1):27–35, 1997.

[10] Y. Kotidis and N. Roussopoulos. An alternative storage organization for ROLAP aggregate views based on cubetrees. In *Proc. ACM SIGMOD Conf.*, pages 249–258, 1998.

[11] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and bulk incremental updates on the data cube. In *Proc. ACM SIGMOD conf.*, pages 89–99, 1997.

[12] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *Proc. IEEE Intl. Conf. Data Engineering*, pages 328–336, 1994.

[13] Y. Zhao, P. Deshpande, and J. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. ACM SIGMOD Conf.*, pages 159–170, 1997.

Data Cubes in Dynamic Environments

Steven P. Geffner **Mirek Riedewald** **Divyakant Agrawal** **Amr El Abbadi**

Department of Computer Science
University of California, Santa Barbara, CA 93106 †

Abstract

The data cube, also known in the OLAP community as the multidimensional database, is designed to provide aggregate information that can be used to analyze the contents of databases and data warehouses. Previous research mainly focussed on strategies for supporting queries, assuming that updates do not play an important role and can be propagated to the data cube in batches. While this might be sufficient for most of today's applications, there is growing evidence that modern interactive data analysis applications will have to balance update and query costs. Two techniques for maintaining data cubes in dynamic environments are described here. The first, Relative Prefix Sums *(RPS), supports a constant response time for ad-hoc range sum queries on the data cube, while at the same time greatly reducing the update costs compared to prior approaches. The second, the* Dynamic Data Cube *(DDC), guarantees a sub-linear cost for both range sum queries and updates.*

1 Introduction

A data cube or multidimensional database ([7] [4] [1]) is constructed from a subset of attributes in the database. Certain attributes are chosen to be *measure attributes*, i.e., the attributes whose values are of interest. Other attributes are selected as *dimensions* or *functional attributes*. The measure attributes are aggregated according to the dimensions. For example, consider a hypothetical database maintained by an insurance company. One may construct a data cube from the database with SALES as a measure attribute, and CUSTOMER_AGE and DATE_OF_SALE as dimensions. Such a data cube provides aggregated total sales figures for all combinations of age and date. Range sum queries are useful analysis tools when applied to data cubes. A range sum query sums the measure attribute within the range of the query. An example is to "Find the total sales for customers with an age from 37 to 52, over the past three months". Queries of this form can be very useful in finding trends and in discovering relationships between attributes in the database. Efficient range-sum querying is becoming more important with the growing interest in database analysis, particularly in On-Line Analytical Processing (OLAP) [3].

Ho et al. [8] have presented an elegant algorithm for computing range sum queries in data cubes which we call the *Prefix Sum* (PS) approach. The essential idea is to precompute many prefix sums of the data cube, which can

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

†This work was partially supported by NSF grants IIS 98-17432 and IIS 99-70700

A	0	1	2	3	4	5	6	7	8
0	3	5	1	2	2	4	6	3	3
1	7	3	2	6	8	7	1	2	4
2	2	4	2	3	3	3	4	5	7
3	3	2	1	5	3	5	2	8	2
4	4	2	1	3	3	4	7	1	3
5	2	3	3	6	1	8	5	1	1
6	4	5	2	7	1	9	3	3	4
7	2	4	2	2	3	1	9	1	3
8	5	4	3	1	3	2	1	9	6

P	0	1	2	3	4	5	6	7	8
0	3	8	9	11	13	17	23	26	29
1	10	18	21	29	39	50	57	62	69
2	12	24	29	40	53	67	78	88	102
3	15	29	35	51	67	86	99	117	133
4	19	35	42	61	80	103	123	142	161
5	21	40	50	75	95	126	151	171	191
6	25	49	61	93	114	154	182	205	229
7	27	55	69	103	127	168	205	229	256
8	32	64	81	116	143	186	224	257	290

Figure 1: The original array (A, left) and the cumulative array used for the Prefix Sum method (P, right)

then be used to answer ad hoc queries at run-time (see Figure 1). The Prefix Sum method permits the evaluation of any range-sum query on a data cube in constant time. The approach is mainly hampered by its update cost, which in the worst case requires rebuilding an array of the same size as the entire data cube. [6] describes a technique that considerably improves the update performance compared to PS, but still provides a constant range sum query cost. The main idea is to control the cascading updates. The Hierarchical Cubes techniques [2] generalize this idea by offering the user different tradeoffs between update and query cost. The only technique that provides guaranteed sub-linear update and range query cost is the Dynamic Data Cube [5].

In some problem instances, update cost is not a significant consideration. There are, however, many current and emerging applications for which reasonable update cost becomes important. In Section 2 we discuss some dynamic scenarios. Then in Section 3 the Relative Prefix Sum and the Dynamic Data Cube techniques are presented. Section 4 concludes this article.

2 Why Do Updates Matter?

Update complexity is often considered to be unimportant in current-day data analysis applications. These systems are oriented towards batch updates, and for a wide variety of business applications this is considered sufficient. Nevertheless, the batch updating paradigm, a holdover from the computing environment of the 1960's, is tremendously limiting to the field. The Prefix Sum method is a good example of present-day cutting-edge data cube technology. Any range sum query can be answered in constant time. During updates, however, it requires in the worst case updating an array whose size is equal to the size of the entire data cube. It is easy to see that, even under batch update conditions, this model is not workable for many emerging applications (e.g., what if the size of the data cube were a terabyte?).

There is no doubt that OLAP applications typically have to deal with updates. Data warehouses collect constantly changing data from a company's databases; digital libraries and data collections grow with an increasing rate. But isn't it good enough if those updates can be efficiently processed in batches? Why instantly propagating each update to the data cube? Why not just collect all updates during the day and then apply them overnight when nobody uses the data collection? There are several arguments.

- For some applications, it is desirable to incorporate updates as soon as possible. Batch updating unnecessarily limits the range of choices. While some applications do not suffer in the presence of stale data, in many emerging applications, e.g. decision support and stock trading, the instant availability of the latest information plays a crucial role.

- OLAP means *interactive* data analysis. For instance, business leaders will want to construct interactive what-if scenarios using their data cubes, in much the same way that they construct what-if scenarios using spreadsheets now. These applications require real-time (or even hypothetical) data to be integrated with

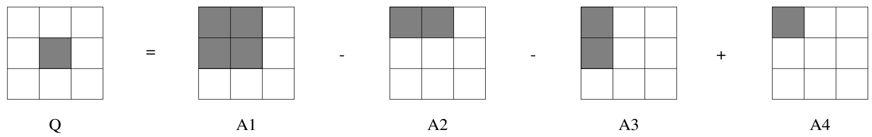

Figure 2: A geometric illustration of the two-dimensional case: SUM(Q) = SUM(A1) - SUM(A2) - SUM(A3) + SUM(A4)

historical data for the purpose of instantaneous analysis and subsequent action. The fact that there are significant impediments to updates in popular data cube techniques prevents these and many other emerging applications from being deployed.

- Batch updates incur another serious handicap. Even though the *average* cost per update might be small, performing the complete batch of updates takes a considerable amount of time. During this time the data in the data cube is generally not accessible to an analyst. The greater the amount of updates, the worse the situation can get. Finding a suitable time slot for this *update window* becomes increasingly harder when businesses demand flexible work hours and 24 hour availability of their data. Also, data collections that are accessible from all over the world (e.g., for multinational companies) do not follow the simple "many accesses during daytime, no accesses at nighttime" pattern.

By reducing the barriers to frequent updates in very large data cubes, new and interesting applications become possible; traditional applications can profit from greater flexibility and 24 hour availability of the data. With growing data collections and a growing demand for interactive analysis of up to date data, traditional approaches like batch updates and re-computation of the complete data cube can not be regarded as sufficient any more for an increasing number of applications.

3 Two Dynamic Data Cube Approaches

In the following, two data cube techniques for dynamic environments are presented. Compared to the Prefix Sum technique [8] they trade query efficiency for faster updates. Both make use of the inverse property of addition by adding and subtracting region sums to obtain the complete sum of the query region, in the same manner as the PS method. As Ho et al. point out, the technique can be applied to any operator $\oplus$ for which there exists an inverse operator $\ominus$ such that $a \oplus b \ominus b = a$ (e.g., COUNT, AVERAGE, ROLLING SUM, ROLLING AVERAGE).

Let A denote the original array, d its dimensionality and n the number of possible indexes (attribute values) for each dimension[1]. $(y_1, \ldots, y_d)$ describes a single *cell*, i.e., a point of the multidimensional data space. Without loss of generality we assume that $(0, \ldots, 0)$ is the point of the array with the smallest index in each dimension. Then the sum for an arbitrary range query can be obtained as the result of combining (adding/subtracting) up to 2^d range sums of the form $\mathrm{SUM}(A[0, \ldots, 0] : A[x_1, \ldots, x_d])$. Figure 2 illustrates the calculations for a two-dimensional data cube. With $\mathrm{SUM}(A[y_1, \ldots, y_d] : A[z_1, \ldots, z_d])$ we refer to the aggregate sum of all cells enclosed in the bounding box described by $(y_1, \ldots, y_d)$ ("upper left" corner) and $(z_1, \ldots, z_d)$ ("lower right" corner). The "upper left" corner of a hyper-rectangle, i.e., the point of the rectangle with the smallest index in each dimension, will be referred to as the *anchor* of that hyper-rectangle. Since an arbitrary range sum can be obtained as the combination of up to 2^d (which is independent of the size of the range) range sums for ranges anchored at $(0, \ldots, 0)$, we will only focus on handling those ranges.

[1]Choosing a single parameter n is done for the sake of clarity and results in simpler formulas. Our techniques, however, are not restricted to data cubes with domains of equal sizes.

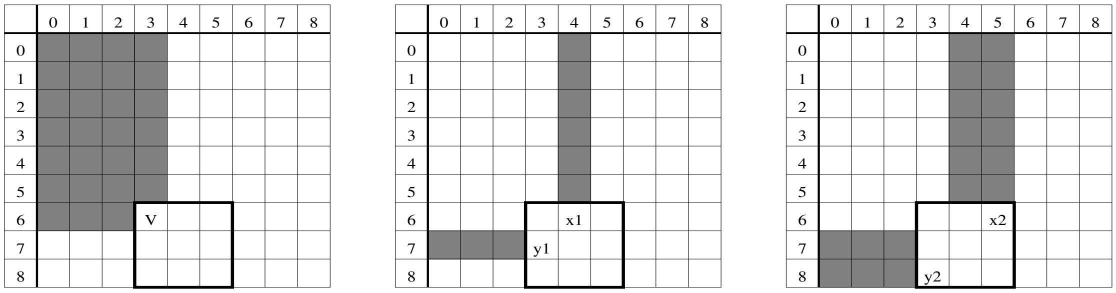

Figure 3: Calculation of overlay values as the sum of the cells in the shaded area on array A

3.1 The Relative Prefix Sum Approach

The Relative Prefix Sum (RPS) method [6] provides constant-time queries with reduced update complexity (compared to the Prefix Sum technique), and is suitable for applications where constant time queries are vital but updates are more frequent than the Prefix Sum method will allow. The main idea behind RPS is to control the cascading updates that lead to poor update behavior.

The RPS method makes use of two components: an *overlay* (*OL*) and a *relative-prefix* (*RP*) array. The overlay partitions array A into fixed size regions called overlay boxes. Overlay boxes store information regarding the sums of regions of array A preceding them. *RP* contains relative prefix sums within regions defined by the overlay. Using the two components in concert, we construct prefix sums "on the fly". We first describe the overlay, then describe *RP*.

3.1.1 Overlay

We define an overlay as a set of disjoint hyper-rectangles of equal size, further on called *overlay boxes*, that completely partition array A into regions of cells. For clarity, and without loss of generality, let the length of the overlay box in each dimension be k. The size of array A is n^d, thus the total number of overlay boxes is $\lceil n/k \rceil^d$. The first overlay box is anchored at $(0, \ldots, 0)$. Let overlay box B be anchored at $(b_1, \ldots, b_d)$. B is said to *cover* a cell $(x_1, x_2, \ldots, x_d)$ in array A if the cell falls within the boundaries of the overlay box, i.e., if for all i: $b_i \leq x_i < b_i + k$. A single cell o of an overlay box *aggregates* a cell a of array A outside the overlay box, if the value of o depends on a's value. Each overlay box corresponds to an area of array A of size k^d cells. The values stored in an overlay box provide sums of regions outside the box. In the two-dimensional example in Figure 3 the cells in the top row and the leftmost column contain the sums of the values in the corresponding shaded cells of array A (those overlay cells aggregate the respective cells in the shaded area). The other cells covered by the overlay box are not needed in the overlay, and would not be stored.

In general only overlay cells in the "upper left" surfaces are needed, i.e., in those surfaces that contain the anchor cell. More formally, overlay box B anchored at $(b_1, \ldots, b_d)$ aggregates k^d overlay cells $O = (o_1, \ldots, o_i, \ldots, o_d)$, namely those cells that satisfy for each dimension i: $b_i \leq o_i < b_i + k$. Among those overlay cells, only $k^d - (k-1)^d$ are used, namely those where a dimension j exists, such that $o_j = b_j$ (compare to the two-dimensional example). The anchor cell $(b_1, \ldots, b_d)$ stores the value

$$(\sum_{a_1=0}^{b_1} \cdots \sum_{a_d=0}^{b_d} A[a_1, \ldots, a_d]) - (\sum_{a_1=b_1}^{b_1} \cdots \sum_{a_d=b_d}^{b_d} A[a_1, \ldots, a_d])$$

(the sum of all cells in $A[0, \ldots, 0] : A[b_1, \ldots, b_d]$ excluding $(b_1, \ldots, b_d)$). For an overlay cell O where for exactly one dimension j $o_j > b_j$ and for all other dimensions i $o_i = b_i$ the value is calculated as

$$(\sum_{a_1=0}^{b_1} \cdots \sum_{a_{j-1}=0}^{b_{j-1}} \sum_{a_j=b_j+1}^{o_j} \sum_{a_{j+1}=0}^{b_{j+1}} \cdots \sum_{a_d=0}^{b_d} A[a_1, \ldots, a_d])$$

$$-(\sum_{a_1=b_1}^{b_1} \cdots \sum_{a_{j-1}=b_{j-1}}^{b_{j-1}} \sum_{a_j=b_j+1}^{o_j} \sum_{a_{j+1}=b_{j+1}}^{b_{j+1}} \cdots \sum_{a_d=b_d}^{b_d} A[a_1, \ldots, a_d])$$

In general the value stored in a used overlay cell $O = (o_1, \ldots, o_d)$ is

$$(\sum_{a_1=l_1}^{u_1} \cdots \sum_{a_d=l_d}^{u_d} A[a_1, \ldots, a_d]) - (\sum_{a_1=m_1}^{v_1} \cdots \sum_{a_d=m_d}^{v_d} A[a_1, \ldots, a_d])$$

where for all dimensions i:

- if $o_i = b_i$: $l_i = 0, u_i = b_i, m_i = b_i, v_i = b_i$
- if $o_i > b_i$: $l_i = b_i + 1, u_i = o_i, m_i = b_i + 1, v_i = o_i$

Intuitively the first sum includes all cells that fall into the hyper-rectangle $A[0, \ldots, 0] : A[o_1, \ldots, o_d]$, and that are not aggregated by another overlay cell whose coordinates can be obtained by replacing some of the o_i that are greater than b_i by b_i (e.g., in Figure 3 V's coordinates $(3, 6)$ can be obtained by replacing x_2's x-coordinate 5 with the corresponding overlay anchor coordinate 3, therefore x_2's summation in x-direction has to range from (3+1) to 5). The second term subtracts the values of those cells that fall into overlay box B and should therefore not be included in the summation (in Figure 3 for x_2 the values in cells $(4, 6)$ and $(5, 6)$ have to be subtracted).

3.1.2 Relative Prefix Array (*RP*)

The relative prefix array (*RP*) is of the same size as array A. It is partitioned into regions of cells that correspond to overlay boxes. Each region in *RP* contains prefix sums that are relative to the area enclosed by the box, i.e., it is independent of other regions. More formally, given a cell *RP*$[i_1, \ldots, i_d]$ and the anchor cell location $(v_1, \ldots, v_d)$ of the overlay box covering this cell, the value stored in *RP*$[i_1, \ldots, i_d]$ is SUM$(A[v_1, \ldots, v_d] : A[i_1, \ldots, i_d])$.

3.1.3 Query and Update Operations

The range sum for any query anchored at $(0, \ldots, 0)$ can be obtained by adding the corresponding values stored in the overlay and in *RP*. The overlay values and *RP* are therefore sufficient to provide the region sums required by the method illustrated in Figure 2. Figure 4 shows the two data structures for our example array. For instance to calculate the value for SUM$(A[0, 0] : A[7, 4])$ we have to add *OL*$[6, 3]$, *OL*$[7, 3]$, *OL*$[6, 4]$ and *RP*$[7, 4]$, resulting in 4 accesses and returning 142.

In general, a query for SUM$(A[0, \ldots, 0] : A[z_1, \ldots, z_d])$, i.e., a query that sums the values of all array cells up to $Z = (z_1, \ldots, z_d)$, accesses exactly one value in *RP* and some values in the overlay box B_Z that covers Z. Let B_Z be anchored at $(b_1, \ldots, b_d)$. Then all overlay cells in B_Z that together aggregate the range $A[0, \ldots, 0] : A[z_1, \ldots, z_d]$, excluding the cells covered by overlay box B_Z, must be accessed. These are all cells $X = (x_1, \ldots, x_d)$ that satisfy for all dimensions i: $x_i = b_i$ or $x_i = z_i$, excluding cell $(z_1, \ldots, z_d)$ which is not a used overlay cell. Intuitively all overlay cells have to be accessed whose coordinates can be obtained from Z by replacing one or more of the z_i with the corresponding b_i. In total $2^d - 1$ overlay cells must be accessed. The overall cost for a query anchored at $(0, \ldots, 0)$ therefore sums to 2^d. Since an arbitrary range sum query can be computed by adding/subtracting the results of at most 2^d of these queries, the overall worst case query cost

OL	0	1	2	3	4	5	6	7	8
0	0	0	0	9	0	0	17	0	0
1	0			12			33		
2	0			20			50		
3	12	12	17	46	13	27	97	10	24
4	0			7			17		
5	0			15			40		
6	21	19	29	86	20	51	179	20	40
7	0			8			14		
8	0			20			32		

RP	0	1	2	3	4	5	6	7	8
0	3	8	9	2	4	8	6	9	12
1	10	18	21	8	18	29	7	12	19
2	12	24	29	11	24	38	11	21	35
3	3	5	6	5	8	13	2	10	12
4	7	11	13	8	14	23	9	18	23
5	9	16	21	14	21	38	14	24	30
6	4	9	11	7	8	17	3	6	10
7	6	15	19	9	13	23	12	16	23
8	11	24	31	10	17	29	13	26	39

Figure 4: Overlay *OL* and array *RP* with overlay boxes drawn for reference (computed for array A)

OL	0	1	2	3	4	5	6	7	8
0	0	0	0	9	0	0	17	0	0
1	0			12			33		
2	0			20			50		
3	12	12	17	46	13	27	97	10	24
4	0			7			17		
5	0	*		17			42		
6	21	21	31	88	20	51	181	20	40
7	0			8			14		
8	0			20			32		

RP	0	1	2	3	4	5	6	7	8
0	3	8	9	2	4	8	6	9	12
1	10	18	21	8	18	29	7	12	19
2	12	24	29	11	24	38	11	21	35
3	3	5	6	5	8	13	2	10	12
4	7	11	13	8	14	23	9	18	23
5	9	18	23	14	21	38	14	24	30
6	4	9	11	7	8	17	3	6	10
7	6	15	19	9	13	23	12	16	23
8	11	24	31	10	17	29	13	26	39

Figure 5: Effects of an update to cell $(1, 5)$ (marked with a star)

is 4^d. For a certain data cube its dimensionality d is fixed, i.e., we obtain a worst query cost that is constant and independent of the size of the query range.

Determining the update cost requires a more detailed analysis. We first illustrate the update process on our two-dimensional example array (Figure 5). Let $A[1, 5]$ be updated with the value 5 (replacing the former value 3) and let B be the overlay box anchored at $(0, 3)$, i.e., the box that covers $(1, 5)$. In *RP* only cells covered by B that reside to the lower right of $(1, 5)$ including the cell itself, have to be updated (shaded area in *RP*). In *OL*, for overlay boxes in the same row as B to its right, the corresponding column values are affected ($(3, 5)$ and $(6, 5)$), similar for the overlay boxes in the same column as B below it. Finally the anchor cells $(3, 6)$ and $(6, 6)$ have to be updated as well. In Figure 5 the affected overlay cells are shaded.

To keep the description of the general case simple, we assume that k, the side-length of an overlay box, evenly divides n, the side-length of the data cube. Let $Z = (z_1, \ldots, z_d)$ be the updated cell and B_Z be the overlay box anchored at $(b_1, \ldots, b_d)$ that covers Z. The cost of an update to Z is the sum of the cost of updating the overlay cells and the cost of updating *RP*. Regarding the relative prefix sum array *RP*, only cells inside overlay box B_Z can be affected by the update. To be more precise, only those cells in B_Z whose indexes are each at least as great as the corresponding index of Z are to be updated, resulting in a worst case cost of k^d.

Next we describe which overlay cells need to be updated. In general, these are all overlay cells that include Z in their aggregation. Let for each dimension i: $S_i = \{b_i+k; b_i+2k; b_i+3k; \ldots; n-k\}$. Intuitively S_i contains the coordinates of the anchors of the affected overlay boxes (the boxes to the "lower right" of B_Z in two-dimensional terminology) in the i-th dimension. Let for each dimension i: $T_i = \{z_i; z_i+1; z_i+2; \ldots; b_i+(k-1)\}$. This set intuitively contains the coordinates of the affected overlay cells inside a certain overlay box (the cells to the "lower right" of Z in two-dimensional terminology) in the i-th dimension. Then the overlay cells $O = (o_1, \ldots, o_d)$ that need to be updated are those that satisfy

$$\forall i : o_i \in \begin{cases} S_i & \text{, if } z_i = b_i \\ S_i \cup T_i & \text{, otherwise} \end{cases}$$

and do not belong to overlay box B_Z. In the example of Figure 5 the sets become $S_x = \{3; 6\}$, $T_x = \{1; 2\}$,

$S_y = \{6\}$ and $T_y = \{5\}$. Since $z_x = 1 > 0 = b_x$ and $z_y = 5 > 3 = b_y$ the affected overlay cells are all cells (o_x, o_y) where $o_x \in \{1; 2; 3; 6\}$ and $o_y \in \{5; 6\}$, minus $(1,5)$ and $(2,5)$ which fall into the same overlay box as $(1,5)$.

Obviously the greatest number of affected overlay cells results from choosing $z_i \neq b_i$ and z_i as small as possible[2]. S_i has at most $n/k - 1$ elements, T_i can have up to $k-1$ elements[3]. Since there are at most $(n/k - 1 + k - 1) = (n/k + k - 2)$ possible values for each dimension, there are at most $(n/k + k - 2)^d$ overlay cells that satisfy the above formula. Among those, $(k-1)^d$ cells fall into overlay box B_Z[4]. Altogether $(n/k + k - 2)^d - (k-1)^d$ overlay cells must be updated in the worst case. Note, that this bound is tight, since it is met for an update on $(1, \ldots, 1)$. For $n/k \geq 2$ the cost of updating the overlay cells determines the worst overall update cost. Therefore the worst update cost is obtained when cell $(1, \ldots, 1)$ is updated, resulting in a cost of $(n/k + k - 2)^d - (k-1)^d + (k-1)^d = (n/k + k - 2)^d$. This value is minimized for $k = \sqrt{n}$. The worst update cost therefore is $O(n^{d/2})$ (compare to $O(n^d)$ for the PS technique).

3.2 The Dynamic Data Cube (DDC)

The Dynamic Data Cube [5] provides sub-linear performance ($O(\log^d n)$) for both range sum queries and updates on the data cube. The method supports dynamic growth of the data cube in *any* direction and gracefully manages clustered data and data cubes. The DDC method utilizes a tree structure which recursively partitions array A into a variant of overlay boxes. Each overlay box will contain information regarding relative sums of regions of A. By descending the tree and adding these sums, we will efficiently construct sums of regions which begin at $A[0, \ldots, 0]$ and end at any arbitrary cell in A. To calculate complete region sums from the tree, we again make use of the inverse property of addition as illustrated in Figure 2. We will first describe the overlay box variant, then describe their use in constructing the Dynamic Data Cube.

3.2.1 Overlay Variant

For the DDC we define an overlay as before, i.e., as a set of disjoint hyper-rectangles (hereafter called "boxes") of equal size that completely partition the set of cells of array A into non-overlapping regions. However, these overlay boxes differ from those in RPS in the values they store and in the number of overlay boxes used to partition the data space. Referring to Figure 6, S is the subtotal cell, while x_1, x_2, x_3 are row sum cells in the first dimension and y_1, y_2, y_3 are row sum cells in the second dimension. Each box stores exactly $k^d - (k-1)^d$ values; the other cells covered by the overlay box are not needed in the overlay, and would not be stored. Values stored in an overlay box provide sums of regions *within* the overlay box. Figure 6 demonstrates the calculation of those values. The row sum values shown in the figure are equal to the sum of the associated shaded cells in array A. Note that row sum values are cumulative; i.e., y_2 includes the value of y_1, etc. Formally, given an overlay box anchored at $A[i_1, i_2, \ldots, i_d]$, the row sum value contained in cell $(i_1, i_2, \ldots, j, \ldots, i_d)$ is equal to $\text{SUM}(A[i_1, i_2, \ldots, i_d] : A[i_1, i_2, \ldots, j, \ldots, i_d])$.

3.2.2 Constructing the Dynamic Data Cube

Overlay boxes are used in conjunction with a tree that recursively partitions array A. We now describe its construction (Figure 7). The root node of the tree encompasses the complete range of array A. It forms children by dividing its range in each dimension in half. It stores a separate overlay box for each child. Each of its children are in turn subdivided into children, for which overlay boxes are stored. This recursive partitioning continues until the leaf level. Thus, each level of the tree has its own value for the overlay box size k; k is $n/2$ at the root

[2]The choice of z_i determines the value of b_i. The smaller z_i, the smaller b_i, i.e., the more elements in S_i.

[3]This is because $z_i \neq b_i$, i.e., $z_i \geq b_i + 1$.

[4]These are all cells where $\forall i : o_i \in T_i$.

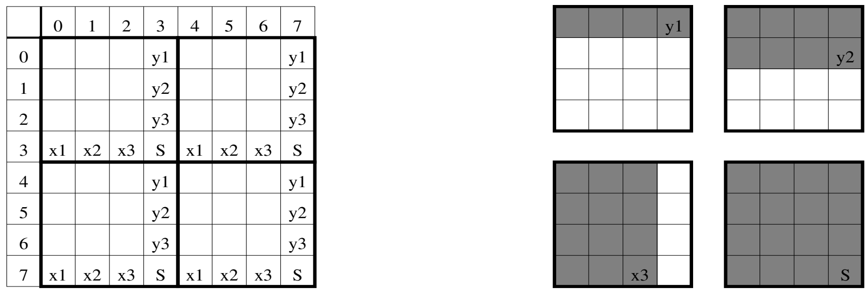

Figure 6: Partitioning of array A into overlay boxes and calculation of overlay values

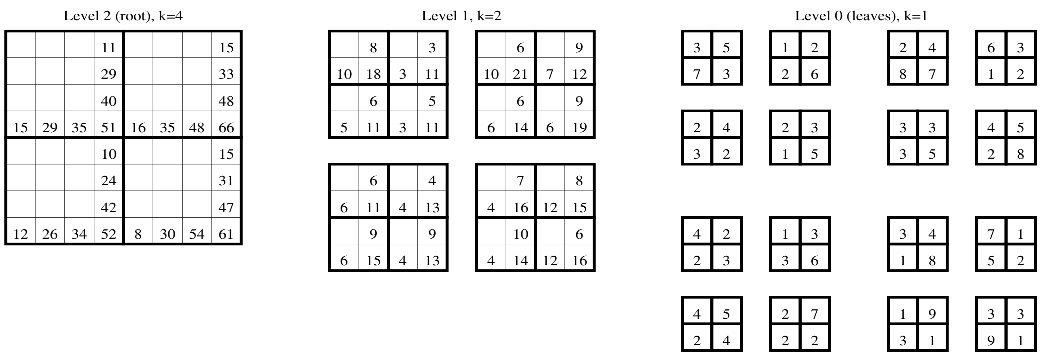

Figure 7: Dynamic Data Cube (computed for the data in array $A[0,0]:A[7,7]$)

of the tree, and is successively divided in half for each subsequent tree level. We define the leaf level as the level wherein $k=1$. When $k=1$, each overlay box contains a single cell; since a single-cell overlay box contains only the subtotal cell, the leaf level contains the values stored in the original array A.

Overlay box values are stored in special structures to guarantee the sublinear query and update times. For two-dimensional overlays (i.e., $d=2$) we do not store the values of an overlay box in arrays. Instead a hierarchical structure is used (B ^{c}-tree, see [5]) that has an access and update cost of $O(\log n)$. For higher dimensional data cubes ($d>2$) we make the observation that the surfaces containing the overlay values of a d-dimensional overlay box are $(d-1)$-dimensional. Thus, the overlay box values of a d dimensional data cube can be stored as $(d-1)$-dimensional data cubes using Dynamic Data Cubes, recursively. The recursion stops for $d=2$.

3.2.3 Query and Update Operations

The range sum for any query anchored at $(0,\ldots,0)$ is obtained by only accessing overlay values. We describe this process for a query $\mathrm{SUM}(A[0,\ldots,0]:A[z_1,\ldots,z_d])$, i.e., a query that sums the values of all array cells up to $Z=(z_1,\ldots,z_d)$. The query process begins at the root of the tree. The algorithm checks the relationship of cell Z and the overlay boxes in the node. When an overlay box covers Z, a recursive call to the function is performed, using the child associated with the overlay box as the node parameter (i.e., the algorithm descends the tree for that case). When Z comes before the overlay box in any dimension (i.e., has a smaller index than each cell covered by the overlay box in that dimension), the query region does not intersect the overlay box, and therefore this box does not contribute a value for the sum. When Z comes after the overlay box in every dimension (i.e., has a greater index than each cell covered by the overlay box in every dimension), the query includes the entire overlay box, and the box contributes the subtotal to the sum. Otherwise the cell is neither before nor after the box, i.e., the query area intersects the overlay box, and the box contributes the corresponding overlay value (a row sum value in two-dimensional terminology) to the sum.

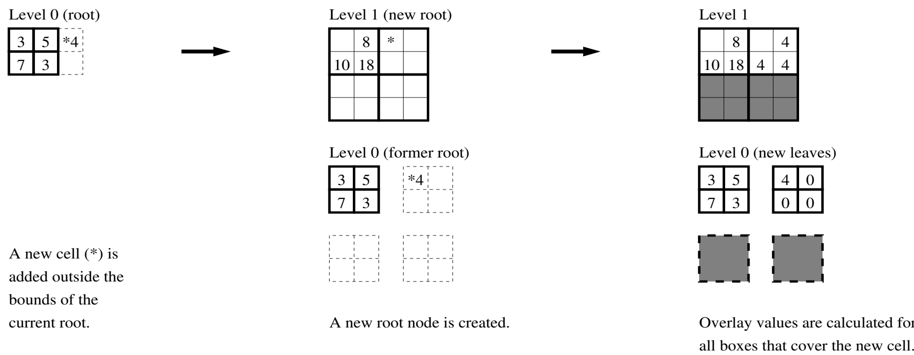

Figure 8: Example for the growth of the Dynamic Data Cube (shaded areas do not store values)

Since overlay boxes at the same tree level are non-intersecting, at most one child will be descended at a tree level. The contribution of overlay boxes that intersect the query area but do not cover Z is obtained by accessing a single overlay value. There are at most $2^d - 1$ overlay boxes in a node that can have this property. In sum at most $(2^d - 1)\log_2 n = \mathrm{O}(\log n)$ overlay values are accessed. Due to the recursive way of storing the overlays, accessing a single overlay value costs $\mathrm{O}(\log^{d-1} n)$, resulting in an overall query cost of $\mathrm{O}(\log^d n)$ (for details see [5]).

To perform an update on cell Z the DDC tree has to be descended in a way similar to the query process. Even though the cells of an overlay box store cumulative values, the balanced query/update cost of the B^{c}-trees (for $d = 2$) together with the recursive way of storing higher dimensional boxes result in a worst case update cost of $\mathrm{O}(\log^d n)$ (for details see [5]).

3.2.4 Dynamic growth of the cube

Neither the Prefix Sum (PS), nor the Relative Prefix Sum (RPS), nor the Hierarchical Cubes (HC) [2] methods address the growth of the data cube. Instead, they assume that the size of each dimension is known a priori. For some applications, however, it is more convenient and space efficient to grow the size of the data cube dynamically to suit the (size of the) data. For instance, an attribute might have a large domain, but the data cube only contains non-empty cells that can be addressed by a much smaller range of values for that attribute. Take for example astronomical databases where new telescopes allow discovering stars in greater distance; or commercial applications where new products and customers are added or deleted from time to time.

The PS, RPS and HC methods would store each single cell in non-populated areas, wasting a huge amount of space. The Dynamic Data Cube, on the other hand, could start by building the smallest data cube that contains all non-empty cells. As soon as a cell is inserted that lies outside the current data cube, the data cube "grows" into the required direction. New roots are created successively, each time doubling the size of the data cube in each dimension, until the new root encompasses the new cell. This update process is incremental, i.e., the old tree structure appears unchanged as the descendent of the new root (see Figure 8 for a simple example). Only one overlay box at each tree level is affected by an update; therefore, we will create only one child node and overlay box per tree level during this process.

This incremental construction of the Dynamic Data Cube is naturally suited to clustered data and data that contains large, non-populated regions. Where data does not exist, overlay boxes will not be instantiated; thus, the Dynamic Data Cube avoids the storage of empty regions. Since overlay boxes are self-contained, there is no cascading update problem associated with adding a new cell. The Dynamic Data Cube allows graceful growth of the data cube in any direction, making it more suitable for applications which involve change or growth. Note that the PS and RPS techniques could be augmented by methods to handle a data cube growth by appending rows. Handling growth in *any* direction, however, will be very costly.

4 Conclusion

In the near future an increasing number of applications will require or be enabled by providing fast and frequent updates on data cubes and avoiding long down-times of the data analysis tools. Together, the RPS and DDC methods offer a range of options for implementing data cubes in such dynamic environments. The Dynamic Data Cube provides balanced sub-linear performance for queries and updates. It is suitable for dynamic environments where queries and updates are both frequent; where data cubes are very large; where data is clustered and sparse; and where the data can grow in any direction relative to the original data (i.e., updates are not append-only). The Relative Prefix Sum technique does not offer this flexibility, but has its merits for applications that do not deal with frequent updates but require fast answers within a guaranteed time limit.

The obvious disadvantage of our methods compared to PS and HC is the usage of additional space. While PS and HC require exactly the same amount of storage as the original data cube ($O(n^d)$), RPS and DDC need to store the overlay values. In the case of RPS the storage overhead is provably within small bounds. Since each overlay box of size k^d stores $k^d-(k-1)^d$ values, the ratio of RPS's total storage requirements to the requirements of the original data cube is $2-((k-1)/k)^d$, i.e., less than 2. For $d=4$ and $k=100$ RPS uses only 4% more storage. In the case of DDC it is obvious that the lowest tree levels consume the most storage. By deleting a certain number of those levels, one can trade off query speed for storage space, or conversely bring the DDC within delta of the storage required by the Prefix Sum method (for details see [5]). Also, in the case of DDC the worst case storage overhead will only occur for dense data cubes. For empty chunks whole subtrees will not be created, saving a considerable amount of space for practical applications.

We are currently developing new techniques that apply the basic ideas of the presented approaches to high-dimensional and sparse data sets.

References

[1] R. Agrawal, A. Gupta, and S. Sarawagi. Modeling multidimensional databases. In *Proc. 13th ICDE*, 1997.

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

[3] E. F. Codd. Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate. Technical report, E. F. Codd and Associates, 1993.

[4] The OLAP Council. *MD-API the OLAP Application Program Interface Version 5.0 Specification*, September 1996.

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

[6] S. Geffner, D. Agrawal, A. El Abbadi, and T. Smith. Relative prefix sums: An efficient approach for querying dynamic OLAP data cubes. In *Proc. 15th ICDE*, 1999.

[7] 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.

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

On Sampling and Relational Operators

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Rajeev Motwani
Stanford University
rajeev@cs.stanford.edu

Abstract

A major bottleneck in implementing sampling as a primitive relational operation is the inefficiency of sampling the output of a query. We highlight the primary difficulties, summarize the results of some recent work in this area, and indicate directions for future work.

1 Introduction

Data warehouses based on relational databases are becoming popular. The investment in data warehouses is targeted towards developing decision support applications that leverage the massive amount of data stored in data warehouses for a variety of business applications. On Line Analytical Processing (OLAP) and data mining are tools for analyzing large databases that are gaining popularity. Many of these tools serve as middleware or application servers that use a SQL database system as the backend data warehouse. They communicate data retrieval requests to the backend database through a relational (SQL) query. On a large database, the cost of executing such ad-hoc queries against the relational backend can be expensive. Fortunately, many data mining applications and statistical analysis techniques can use a sample of the data requested in the SQL query without compromising the results of the analysis. Likewise, OLAP servers that answer queries involving aggregation (e.g., "find total sales for all products in the NorthWest region between 1/1/98 and 1/15/98") can significantly benefit from the ability to present to the user an approximate answer computed from a sample of the result of the query posed to the relational database. It is well-known that for results of aggregation, sampling can be used accurately and efficiently. However, it is important to recognize that whether for data mining, OLAP, or other applications, *sampling must be supported on the result of an arbitrary SQL query*, not just on stored relations. For example, the preceding example of the OLAP query uses a star join between three tables (date, product, and sales).

This paper is concerned with supporting random sampling as a primitive operation in relational databases. In principle, this is easy — introduce into SQL an operation $\text{SAMPLE}(R, f)$ which produces a uniform random sample S that is an f-fraction of a relation R. While producing a random sample from a relation R is not entirely trivial, it is a well-studied problem and efficient strategies are available [15]. However, these techniques are not effective if sampling needs to be applied to a relation R produced by a query Q rather than to a base relation. It seems grossly inefficient to evaluate Q, computing the entire relation R, only to throw away most of it when applying $\text{SAMPLE}(R, f)$. It would be much more desirable and efficient to *partially* evaluate Q so as to generate only the sample of R.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

For this purpose, it suffices to consider the case where we are given a query tree T with $\text{SAMPLE}(R, f)$ only at the root. In this setting, it seems plausible that tremendous gains in efficiency can be achieved by "pushing" the sample operation down the tree towards the leaves, since then we would be feeding only a small (random) fraction of the relations (stored as well as intermediate relations) into the query tree and thereby minimizing the cost of query evaluation. To this end, we need to be able to "commute" the sample operation with standard relational operations.

There has not been much past work on supporting sampling as an operation for the end-user of a database system. While random sampling has been proposed and used in many different ways in databases [14, 15], the main focus has been on the use of random sampling for the purposes of estimating query result size, aggregate values, and parameters for query optimization [16, 13, 10, 11, 7, 6, 3].

2 Difficulty of Sampling and Possible Solutions

In this section we elucidate some of the main difficulties in efficiently implementing sampling in a relational system and suggest some avenues for circumventing these difficulties.

2.1 Join

We focus primarily on developing a technique for commuting sampling with a *single* join operation, since we could apply this technique repeatedly to push down the sample operator from the root to the leaves of a join tree. We establish that it is not possible to produce a sample of the result of even a single join from random samples of the two relations participating in the join, and show how we can leverage database statistics to circumvent this impossibility result.

Example 1: Suppose that we have the relations

$$R_1(A, B) = \{(a_1, b_0), (a_2, b_1), (a_2, b_2), (a_2, b_3), \ldots, (a_2, b_k)\},$$

$$R_2(A, C) = \{(a_2, c_0), (a_1, c_1), (a_1, c_2), (a_1, c_3), \ldots, (a_1, c_k)\}.$$

That is, R_1 is defined over the attributes A and B; amongst its $n_1 = k + 1$ tuples, one tuple has the A-value a_1 and k tuples have the A-value a_2, but all have distinct B-values. Similarly, R_2 is defined over the attributes A and C; amongst its $n_2 = k + 1$ tuples, k tuples have the A-value a_1 and one tuple has the A-value a_2, but all have distinct C-values. Observe that their join over A, $J = R_1 \bowtie R_2$, is of size $n = 2k$ and has k tuples with A-value a_1 and k tuples with A-value a_2.

Assume that we wish to choose a random sample. Consider a random sample $S \subseteq J$. We expect that roughly half of the tuples in S have A-value a_1, and roughly half of the tuples in S have A-value a_2.

Suppose we pick random samples $S_1 \subseteq R_1$ and $S_2 \subseteq R_2$. It is quite unlikely that S_1 will contain the tuple (a_1, b_0), or that S_2 will contain the tuple (a_2, c_0). Thus, given the samples S_1 and S_2, it is impossible to generate a random sample of $J = R_1 \bowtie R_2$ for any reasonable sampling fraction. Note that this conclusion holds even if we allow (say) S_2 to be all of R_2 but require that S_1 be a *proper* subset of R_1. In fact, in all these cases we would expect $S_1 \bowtie S_2$ to be empty.

This suggests that SAMPLE does not commute with join since $\text{SAMPLE}(R_1, f_1) \bowtie \text{SAMPLE}(R_2, f_2)$ may not even contain any non-trivial size subset of J, and so further computation or sampling from it cannot be used to extract a sample of J. We formalize this intuition as follows. Suppose we are given samples $S_1 = \text{SAMPLE}(R_1, f_1)$ and $S_2 = \text{SAMPLE}(R_2, f_2)$. Given only S_1, S_2, and any desired set of statistics for R_1 and R_2, but without direct access to the tuples in R_1 and R_2, we are required to produce a sample $S = \text{SAMPLE}(R_1 \bowtie R_2, f)$ for some value of $f > 0$. The theorem below states the extremely negative result implicit in Example 1 — even when we are given *arbitrarily large samples* from R_1 and R_2, as well as *arbitrarily detailed statistics*, it is still not possible to generate any *non-empty* random sample of $R_1 \bowtie R_2$

Theorem 1: Suppose that at least one of f_1, f_2 is strictly less than 1. Then, it is not possible to generate the sample $S = \text{SAMPLE}(R_1 \bowtie R_2, f)$ from S_1 and S_2 for any $f > 0$ where $S_1 = \text{SAMPLE}(R_1, f_1)$ and $S_2 = \text{SAMPLE}(R_2, f_2)$.

Despite this strong negative result, it is still possible to exploit the power of sampling. Our key observation is that given some partial statistics (e.g., histograms) on one of the operand relations, we can use the statistics to *bias* the sampling from the second relation in such a way that it becomes possible to produce a sample of the join. We devise a variety of sampling schemes based on these observations, improving the state-of-the-art for join sampling. In the context of a join tree, our work shows that it is possible to push down the sampling operation to *one* of the two operand relations. At the same time, our negative results above show that it is inherently difficult to achieve greater efficiency by pushing sampling down to *both* operands of a join in a query tree.

For the rest of the discussion, we will need to use some notation. Assume that the two relations R_1 and R_2 are of size n_1 and n_2, respectively, and that we are interested in an equi-join with respect to an attribute A. We denote the domain of the attribute A by D. For each value $v \in D$, let $m_1(v)$ and $m_2(v)$ be the number of distinct tuples in R_1 and R_2, respectively, that contain value v in attribute A.

Observe that the impossibility of commuting SAMPLE with join does not preclude the possibility of somehow obtaining $\text{SAMPLE}(R_1 \bowtie R_2, f)$ from *non-uniform samples* of R_1 and R_2. To better understand this point, consider the tuple $t = (a_1, b_0) \in R_1$ and its influence on $R_1 \bowtie R_2$. While $m_1(a_1) = 1$, the set $J_t(R_2)$ has size $m_2(a_1) = k$. Thus, even though a random sample of R_1 is unlikely to pick up the tuple with A-value a_1, half of the tuples in the join J have A-value a_1. This suggests that we sample a tuple $t \in R_1$ of join attribute value v with probability proportional to $m_2(v)$, in the hope that the resulting sample is more likely to reflect the structure of J. This is the basic insight behind most of our strategies in [4]. As an example, consider the Frequency-Partition-Sample strategy [4], based on the insight that the naive sampling strategy performs badly only when the average frequency of attribute values is high enough to make the join size significantly larger than the size of the operand relations. The idea is to partition the operands into two subrelations, one with the high-frequency values and the other with the low-frequency values. For the low-frequency values, we can use the naive sampling strategy, but for the high-frequency values we need to develop more refined approaches as this is precisely the set of values for which computing the full join is expensive.

The preceding discussion suggests that we sample tuples from R_1 based on frequency statistics for R_2. This leads us to a natural classification of the problem and applicability of sampling techniques based on availability of indexes and statistics on operand relations. We remark that the sampling strategy due to Olken et al [14, 15] applies only to the case where indexes and statistics are available on both operands of the join. In contrast, our hybrid sampling technique does not need a full index on R_2 but merely some partial statistics (on the high frequency values).

The preceding discussion was primarily concerned with the issue of sampling from a single join operation. We now turn to the question of implementing sampling as a primitive relational operation in the context of a join tree. Suppose we have a query Q generating a join tree T with a sampling operation applied to its result R. The earlier negative results state that it is essentially impossible to push down the sampling operation to both operands of a join operation, shedding some light on the difficulty of dealing with linear and arbitrary join trees. Consider sampling from $Q = (R_1 \bowtie R_2 \bowtie R_3)$ - we cannot just select a uniform random sample of $R_1 \bowtie R_2$ but have to pick a non-uniform sample whose distribution depends on the statistics of R_3. But then, what about pushing the sampling further down the tree to R_1? Now, we will have to sample from R_1 using statistics for both R_2 and R_3. In principle, this can be done (e.g., using multidimensional histograms on $\{R_1, R_2\}$), since the operand relations are all base relations and their statistics can be precomputed. However, such requirements provide a high bar on being able to exploit sampling "on-the-fly", i.e., on ad-hoc queries.

Finally, consider the important special case where the query consists of *foreign key* joins, e.g., $R_1 \bowtie R_2$ where R_2 is the dimension (referenced) table. In such cases, each tuple from R_1 joins with at most one tuple in R_2. This allows us to uniformly sample R_1 to create a sample for $R_1 \bowtie R_2$. Unfortunately, in most cases, such queries would contain selection conditions on R_2. As the next subsection shows, presence of selections limits our ability to use sampling.

2.2 Select and Group By

Since most queries involve selection conditions, it is important to study the interaction of sampling with selection. The main insight here is that if the selectivity of a query is low, then it dramatically and adversely impacts the accuracy of sampling-based estimation. Consider a simple query *select count(*) from lineitem*. Suppose we wish to use sampling to approximate the query result. Consider a sample where each tuple is included with probability p (the sampling fraction). If the number n of tuples in the table is large, then the expected number of tuples in the sample (i.e., np) is large and we know from Chernoff bounds [12] that the actual number of tuples in the sample is very close to the expected value with very high probability. As a result, we can accurately estimate the size of the table from the sample.

Suppose, however, we were faced with the following query: *select count(*) from lineitem where city = "Palo Alto"*. Let n_s be the number of tuples that satisfy the selection condition in the query. If n_s is not large enough, then the expected value of the number of tuples from the sample that satisfy the selection condition (i.e., $n_s p$) is low and the variance is large compared to this expected value. In that case, we will incur a large error in query estimation based on the number of sampled tuples passing the selection condition. This intuition is formalized in the following theorem due to Chaudhuri, Datar, Motwani, and Narasayya [5] where we show that the relative error for a query is proportional to the inverse square root of the selectivity of the conditions in the query. Therefore, we need a significantly larger sample for queries with low selectivity. Earlier work has not satisfactorily addressed this issue. Acharya et al [1] maintain samples of varying size and use different samples for different queries. This requires estimating the selectivity of a query accurately. If we underestimate the selectivity, we will end up using a bigger sample than required, thereby decreasing efficiency. On the other hand, if we overestimate the selectivity, we will end up using a smaller sample and incurring a loss in accuracy. Likewise, the technique adopted by Hellerstein et al [9] will fail to obtain good confidence bounds unless we have looked at a large part of the relation. Thus, low selectivity is a serious problem that limits the use of sampling as a robust technique.

Although we have focussed exclusively on Selection, a careful reader will note that the effect of *Group By* is no different from that of selection since the effect of Group By is to create multiple selection queries that differ by virtue of their qualifying values on the grouping attribute. Therefore, similar issues as has been studied for Selection arises.

2.3 Select Distinct

We now turn to the problem of estimating the number of distinct values in a column of a table. Our objective is to provide estimation for such aggregates without scanning the entire table. A natural idea is to use a random sample of the data. Unfortunately, we [3] proved that large errors are unavoidable for estimates derived only from uniform random samples unless the sample size is very close to the size of the table itself. Indeed, we have generalized this result and proved a stronger negative result for the the most general possible class of estimators that, instead of being restricted to merely a random sampling, are allowed to use *any* (possibly randomized) strategy to select a sequence of r values to examine in the the input table. Such estimators could even pick a sequence of r tuples *adaptively*, i.e., pick the next tuple to examine based on the values of the previously-examined tuples. This explains the observation by Haas et al [8] that no known estimator behaves well on all data distributions primarily because their performance is sensitive to the skew of the data. Indeed, our lower bound on the error stems in part from the difficulty in distinguishing between low-skew and high-skew data. In fact, the proof hinges on the inability to distinguish between two specific scenarios: one where there are few distinct values of very high frequency, and another where there a few high-frequency values together with a large number of very low-frequency values (thus having a large number of distinct values). The first scenario has low skew with few distinct values and the second one has high skew with a large number of distinct values. The following theorem, due to Charikar, Chaudhuri, Motwani, and Narasayya [2], states that any possible estimator must incur large error on at least one of these two scenarios.

Theorem 2: Consider any (possibly adaptive and randomized) estimator $\widehat{D}$ for the number of distinct values D that examines at most r rows in a table with n rows. Then, for any $\gamma > e^{-r}$, there exists a choice of the input data such that with probability at least γ,

$$\text{error}(\widehat{D}) \geq \sqrt{\frac{n-r}{2r} \ln \frac{1}{\gamma}}.$$

Our error bounds seem to be close to the errors observed in real experiments, even though our results give worst-case bounds and the data in the experiments may have had special structure that could have been used to obtain better estimates. However, in [2], we were also able to provide a robust estimator that has an error bound that matches the lower bound above.

2.4 Aggregation

So far we have restricted our focus to queries that count the number of tuples, e.g., *select count(*) From sales_table*. However, in many cases, we are interested in other aggregating functions in addition to *count*, e.g., *select sum(sales) from sales_table*. For such aggregates, any skew present in the data may have a severe impact on use of sampling. The following example is instructive.

Example 2: Suppose we have $10,000$ tuples of which 99% have value 1 in the aggregate column, while the remaining 1% of the tuples have value 1000. Consider using a uniform random sample of size 100 to estimate the average value of the column over all tuples. It is quite likely that the sample would not include any tuple of value 1000, leading to an estimate of 1 for the average value. On the other hand, if perchance two or more tuples of value 1000 were to be included in the sample, then our estimate of the average value would be more than 20.98. In either case, the estimate would be far from the true value of the average which is 10.99. It is only in the case where we get exactly one tuple of value 1000 in the sample that we would obtain a reasonable estimate of the average value. But the latter event has probability 0.37, therefore with high probability we would get a large error in the estimate.

As in the above example, a skewed database is characterized by the existence of certain tuples that are deviant from the rest with respect to their aggregate value. We refer to these tuples as *outliers*. We observe that we cannot afford to miss these outliers in the sample. It should be noted that the presence of these tuples in the wrong proportion in the sample would also lead to large errors in the estimate of a query's result. In Datar, Motwani, and Narasayya [5], we show that the relative error is proportional to the ratio C_{MAX}/C_{AVG}, where C_{MAX} and C_{AVG} are the maximum and average values in the data set. This ratio increases with the skew in the data. Not surprisingly, the approach to handle such skew is to identify tuples that are outliers [5]. The earlier work of Acharya et al [1] does not address this issue. In the work of Hellerstein et al [9], they assume that the aggregate attributes are not skewed and the experiments reported in their paper are over grades data where there is not much variation. The confidence intervals provided with their estimate could be severely affected by the presence of skew.

3 Practicality of Sampling

Random sampling is an attractive data-reduction operation that reduces the cost of executing complex ad-hoc queries on large databases. But the preceding discussion shows that there are non-trivial barriers to implementing sampling as a relational operator. While the techniques we reviewed are able to circumvent some of these barriers, the goal of pushing sampling down to the leaf nodes of a complex query tree requires a substantial increase in the sample size to keep the error within bounds. In effect, sampling the results of an ad-hoc query is an inherently difficult and unsolved problem. Our work has demonstrated that auxiliary information such as histograms can be leveraged to facilitate efficient sampling. It remains an interesting question to determine what other auxiliary information may be employed for this purpose and more broadly for computing approximate answers to relational queries.

References

[1] S. Acharya, P. Gibbons, V. Poosala, and S. Ramaswamy. Join Synopses for Approximate Query Answering. In *Proc. ACM SIGMOD Conference,* 1999.

[2] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards Estimation Error Guarantees for Distinct Values. Submitted for publication, 1999.

[3] S. Chaudhuri, R. Motwani, and V. Narasayya. Using Random Sampling for Histogram Construction. In *Proc. ACM SIGMOD Conference,* pages 436–447, 1998.

[4] S. Chaudhuri, R. Motwani, and V. Narasayya. On Random Sampling Over Joins. In *Proc. ACM SIGMOD Conference,* pages 263–274, 1999.

[5] S. Chaudhuri, M. Datar, R. Motwani, and V. Narasayya. Overcoming Limitations of Sampling for Aggregation Queries. Submitted for publication, 1999.

[6] S. Ganguly, P.B. Gibbons, Y. Matias, and A. Silberschatz. Bifocal Sampling for Skew-Resistant Join Size Estimation. In *Proc. ACM SIGMOD Conference,* pages 271–281, 1996.

[7] P.J. Haas, J.F. Naughton, and A.N. Swami. On the Relative Cost of Sampling for Join Selectivity Estimation. In *Proc. 13th ACM PODS,* pages 14–24, 1994.

[8] P.J. Haas, J.F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *Proc. 21st VLDB,* pages 311–322, 1995.

[9] J.M. Hellerstein, P.J. Haas, and H.J. Wang. Online Aggregation. In *Proc. ACM SIGMOD Conference,* pages 171–182, 1997.

[10] W. Hou, G. Ozsoyoglu, and E. Dogdu. Error-Constrained COUNT Query Evaluation in Relational Databases. In *Proc. ACM SIGMOD Conference,* pages 278–287, 1991.

[11] R.J. Lipton, J.F. Naughton, D.A. Schneider, and S. Seshadri. Efficient Sampling Strategies for Relational Database Operations. *Theoretical Computer Science* 116(1993): 195–226.

[12] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[13] J.F. Naughton and S. Seshadri. On Estimating the Size of Projections. In *Proc. Third International Conference on Database Theory,* pages 499–513, 1990.

[14] F. Olken and D. Rotem. Simple random sampling from relational databases. In *Proc. 12th VLDB,* pages 160–169, 1986.

[15] F. Olken. *Random Sampling from Databases.* PhD Dissertation, Computer Science, University of California at Berkeley, 1993.

[16] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proc. ACM SIGMOD Conference,* pages 256–276, 1984.

CALL FOR PARTICIPATION

The 16th International Conference on Data Engineering

Holiday Day Inn on the Bay, San Diego, CA, USA
February 29 - March 3, 2000
Sponsored by
IEEE Computer Society TC on Data Engineering

THE CONFERENCE

Data Engineering deals with the use of engineering techniques and methodologies in the design, development and assessment of information systems for different computing platforms and application environments. The 16th International Conference on Data Engineering will continue the tradition of being a premier forum for presentation of research results and advanced data-intensive applications and discussion of issues on data and knowledge engineering. The mission of the conference is to share research solutions to problems of today's information society and to identify new issues and directions for future research and development work.

TECHNICAL PROGRAM HIGHLIGHTS

Keynote Speakers

- Jim Gray, Senior Researcher, Microsoft Research, 1999 Turing Award winner
- Dennis Tsichritzis, Chairman, Executive Board of GMD German National Research Center for Information Technology

Research Paper and Poster Sessions

• Time series • Transactions and workflow • Query processing • Mobile and embedded systems • Storage and process optimization • System administration • Data warehousing • Heterogeneous queries • New trends in data mining • Association rules and correlations • Spatial and temporal data • High-dimensional data • Internet, performance and systems management • New applications • OLAP, DW and data mining

Industrial Sessions

• Fast and reliable database engines • OLAP and data warehousing • Java, Internet and databases • XML and databases • Main memory and small footprint databases

Panels

• Is E-business a new wave for database research • Data mining – niche market or killer app? • Object/database standards soup • XML + databases = ?

CONFERENCE VENUE

The conference will be held at Holiday Inn San Diego On The Bay, 1355 N. Harbor Dr., San Diego, CA 92101, USA. Phone: +1-619-2323861. Toll-free: +1-800-8778920. Fax: +1-619-2324924. E-mail: hi-sandiego@bristolhotels.com. Best location in "America's finest city" - that is where you will find Holiday Inn San Diego On The Bay. The 600 room high-rise hotel lies at the foot of downtown on the waterfront, close to everything there is to see and do in beautiful San Diego. Contemporary accommodations and exceptional service when you check into Holiday Inn San Diego On The Bay, located harborside designed for both business travelers and vacationers. Please visit the conference Web site for additional travel information.

TUTORIAL PROGRAM

1. Web information retrieval, Monika Henzinger (Google Inc), Feb 29, 2000

2. Mobile and wireless DB access, Panos K. Chrysanthis, Evaggelia Pitoura (Univ. of Pittsburgh, Univ. of Ioannina), Feb 29, 2000

3. Data mining with decision trees, Johannes Gehrke (Cornell Univ.), Feb 29, 2000

4. Directories: Managing Data for Networked Applications, Divesh Srivastava (AT&T Research), March 2, 2000

5. Indexing High-Dimensional Spaces, Daniel Keim, Stefan Berchtold (Univ. of Halle, stb gmbh), March 2, 2000

FURTHER INFORMATION

The advanced program and information regarding conference registration and accommodation are available at the ICDE'2000 Conference Web Site: **http://research.microsoft.com/icde2000**.

ICDE'2000 will be preceded by the 10th IEEE Workshop on Research Issues in Data Engineering: Mobile Data Management, February 27-28, 2000. For details, please visit the workshop Web site: **http://www.cs.umbc.edu/ride2000**.

ORGANIZING COMMITTEE

General Chair: P.-Å. (Paul) Larson, Microsoft, USA
Program Co-chairs: David Lomet, Microsoft, USA
Gerhard Weikum, Univ of Saarland, Germany
Program Vice-chairs:
Phil Bernstein, Microsoft, USA
Stavros Christodoulakis, Univ of Crete, Greece
Theo Haerder, Univ of Kaiserslautern, Germany
H. V. Jagadish, Univ of Michigan, USA
Hank Korth, Lucent - Bell Labs, USA
Donald Kossmann, Univ of Passau, Germany
Jeff Naughton, Univ of Wisconsin, USA
Beng Chin Ooi, Nat'l Univ of Singapore, Singapore
Arnie Rosenthal, Mitre, USA
Sunita Sarawagi, IIT Bombay, India
Hans Schek, ETH Zurich, Switzerland
Jeff Ullman, Stanford Univ, USA
Steering Committee Chair: Erich Neuhold, GMD-IPSI, Germany
Panel Program Chair: Mike Carey, IBM Almaden, USA
Tutorial Program Chair: Praveen Seshadri, Cornell Univ, USA
Industrial Program Co-Chairs: Anil Nori, Asera, Inc, USA
Pamela Drew, Boeing, USA
Demo/Exhibits Chair: Ling Liu, Georgia Tech, USA
Publicity Chair: Qiang Zhu, Univ of Michigan, USA
Financial Chair: Roger Barga, Microsoft, USA
Publication Chair: Vijay Kumar, U Missouri - Kansas City, USA
Local Arrangements: Yannis Papakonstantinou, UCSD, USA
Chaitan Baru, San Diego Supercomp. Center

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

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