The cgmCUBE Project: Optimizing Parallel Data Cube Generation For ROLAP*

Frank Dehne
Griffith University
Brisbane, Australia
frank@dehne.net
http://www.dehne.net

Todd Eavis
Concordia University
Montreal, Canada
eavis@cs.concordia.ca
http://www.cs.concordia.ca

Andrew Rau-Chaplin†
Dalhousie University
Halifax, Canada
arc@cs.dal.ca
http://www.cs.dal.ca/∼arc

Abstract

On-line Analytical Processing (OLAP) has become one of the most powerful and prominent technologies for knowledge discovery in VLDB (Very Large Database) environments. Central to the OLAP paradigm is the *data cube*, a multi-dimensional hierarchy of aggregate values that provides a rich analytical model for decision support. Various sequential algorithms for the efficient generation of the data cube have appeared in the literature. However, given the size of contemporary data warehousing repositories, multi-processor solutions are crucial for the massive computational demands of current and future OLAP systems.

In this paper we discuss the cgmCUBE Project, a multi-year effort to design and implement a multi-processor platform for data cube generation that targets the relational database model (ROLAP). More specifically, we discuss new algorithmic and system optimizations relating to (1) a thorough optimization of the underlying sequential cube construction method and (2) a detailed and carefully engineered cost model for improved parallel load balancing and faster sequential cube construction. These optimizations were key in allowing us to build a prototype that is able to produce data cube output at a rate of over one TeraByte per hour.

Keywords: ROLAP, Data Cube, Parallel Processing

*Research supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

†Corresponding Author: Andrew Rau-Chaplin, Faculty of Computer Science, Dalhousie University, 6050 University Ave. Halifax NS Canada B3J 1W5. Phone: 902-494-2732. Fax: 902-492-1517. Email: arc@cs.dal.ca

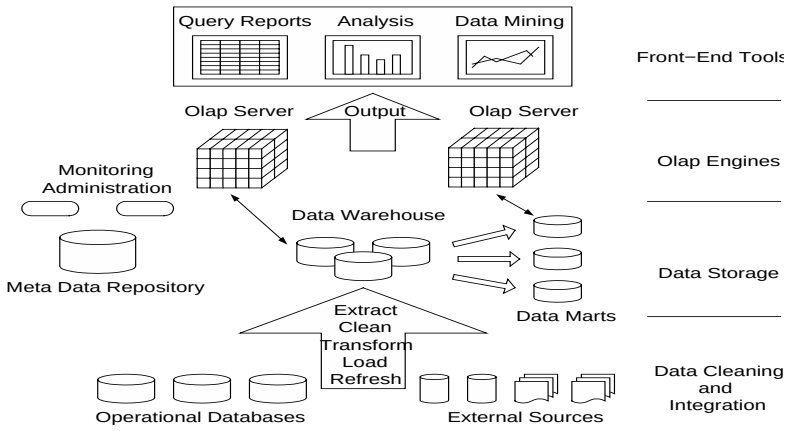


Figure 1: The three-tiered OLAP model.

1 Introduction

While database and data management systems have always played a vital role in the growth and success of corporate organizations, changes to the economy over the past decade have even further increased their significance. To keep pace, IT departments began to exploit rich new tools and paradigms for processing the wealth of data and information generated on their behalf. Along with relational databases, the venerable cornerstone of corporate data management, knowledge workers and business strategists now look to advanced analytical tools in the hope of obtaining a competitive edge. This new class of applications comprises what are known as Decision Support Systems (DSS). They are designed to empower the user with the ability to make effective decisions regarding both the current and future state of an organization. To do so, the DSS must not only encapsulate static information, but it must also allow for the extraction of patterns and trends that would not be immediately obvious. Users must be able to visualize the relationships between such things as customers, vendors, products, inventory, geography, and sales. Moreover, they must understand these relationships in a chronological context, since it is the time element that ultimately gives meaning to the observations that are formed.

One of the most powerful and prominent technologies for knowledge discovery in DSS environments is On-line Analytical Processing (OLAP). OLAP is the foundation for a wide range of essential business applications, including sales and marketing analysis, planning, budgeting, and performance measurement [18, 28]. The processing logic associated with this form of analysis is encapsulated in what is known as the OLAP server. By exploiting multi-dimensional views of the underlying *data warehouse*, the OLAP server allows users to "drill down" or "roll up" on hierarchies, "slice and dice" particular attributes, or perform various statistical operations such as ranking and forecasting. Figure 1 illustrates the basic model where the OLAP server represents the interface between the data warehouse proper and the reporting and display applications available to end users.

To support this functionality, OLAP relies heavily upon a data model known as the *data cube* [16, 19]. Conceptually, the data cube allows users to view organizational data from different perspectives and at a variety of summarization levels. It consists of the *base cuboid*, the finest granularity view containing the full complement of d dimensions (or attributes),

surrounded by a collection of $2^d - 1$ sub-cubes/cuboids that represent the aggregation of the base cuboid along one or more dimensions. Strictly speaking, no special operators or SQL extensions are required to take a raw data set, composed of detailed transaction-level records, and turn it into a data structure, or group of structures, capable of supporting subject-oriented analysis. Rather, the SQL *group-by* and *union* operators can be used in conjunction with 2^d sorts of the raw data set to produce all cuboids. However, such an approach would be both tedious to program and immensely inefficient, given the obvious inter-relationships between the various views. Consequently, in 1995, the data cube *operator* (an SQL syntactical extension) was proposed by Gray et al. [16] as a means of simplifying the process of data cube construction. Subsequent to the publication of the seminal data cube paper, a number of independent research projects began to focus on designing efficient algorithms for the computation of the data cube [1, 3, 19, 22, 23, 30, 31, 32, 33, 35, 36, 38, 40].

The size of data cubes can be massive. In the Winter Corporation's latest report [37], the largest three DSS databases exceed 20 Terabytes in size. More importantly, it is expected that by the end of 2004, the storage requirements of more than 40% of production data warehouses will exceed one Terabyte [8]. It is therefore unlikely that single processor platforms can handle the massive size of future decision support systems. To support very large data cubes, parallel processing can provide two key ingredients: increased computational power through multiple processors and increased I/O bandwidth through multiple parallel disks. However, prior to our work, surprisingly little academic research has focused on the design and implementation of *parallel* algorithms and architectures for the fundamental data cube construction problem (see Section 2).

In [6, 4] we presented parallel data cube construction methods for relational OLAP (ROLAP) on shared disk and shared nothing architectures, respectively. In contrast to other work (see Section 2), our methods scale well and show very good speedup for larger numbers of processors. In addition, our methods integrate well with standard relational database systems and do not require data conversion. On the basis of our preliminary experiments presented in [6, 4], the Natural Sciences and Engineering Research Council of Canada (NSERC) funded a multi -year project, and a dedicated 32 node Linux cluster, with the goal of building a parallel system for ROLAP data cube construction. The latest version of this system, called cgmCUBE, creates ROLAP data cube output at a rate of more than one TeraByte per hour (16 GigaByte per minute).

In this paper we discuss the development of cgmCUBE and how we achieved the above mentioned performance for our system through a very careful optimization of our earlier system. The work presented in this paper represents approximately two person years of system design and performance tuning. It improved the performance of the earlier system by at least one order of magnitude. We discuss in detail the two parts of our work that lead to most of the performance improvement:

1. Improvement of the underlying sequential cube construction method.

2. Improvement of the cost model for better load balancing and faster sequential cube construction.

These results are the core of this paper and represent significant improvements on the state-of-the-art in parallel systems for ROLAP data cube computations. Beyond cgmCUBE, these results are of general interest because issues of load balancing and improving the sequential base method are of almost "universal" importance for parallel cube generation.

The remainder of this paper is organized as follows. After a discussion of other work in the area (Section 2), we give a general overview of our parallel approach for ROLAP data cube construction (Section 3) and an overview of the cgmCUBE software architecture (Section 4). In Sections 5 and 6 we present the core results of this paper: our work on optimizing the underlying sequential cube construction method and our work on optimizing the cost model. Section 7 concludes with a summary of our results and an outline of our planned future work.

2 Comparison With Other Work

As noted above, only a small number of parallel algorithms for data cube generation have been presented in the literature. Muto and Kitsuregawa [26] describe an algorithm that uses a minimum cost spanning tree to define a view generation schedule. Load imbalances are *dynamically* resolved by migrating view partitions from busy processors to idle processors. However, no concrete implementation of the design is provided and the simulated results are based upon assumptions that cannot be guaranteed in practice. In [17], Lu, Huang, and Li describe a parallel data cube algorithm for the Fujitsu AP3000. The method uses hashing to generate each of the 2^d cuboids in the data cube. While an implementation is provided, experimental results indicate little performance advantage beyond four or five processors. A limited ability to exploit smaller intermediate group-bys, coupled with the overhead of independently hashing each cuboid, constrains the potential of this approach. In [27], Ng, Wagner and Yin describe four separate algorithms designed specifically for fully distributed PC-based clusters and large data cubes. Best results were obtained with a technique that "clustered" groups of cuboids on specific nodes. Scheduling decisions were determined dynamically based upon the attribute similarity between the next view to be computed and the cuboids already generated on each local disk. Experimental speedup results were good for one to eight processors. However, relative performance improvements declined rapidly when more than eight processors were used. The primary limitation on speedup appears to be a scheduling mechanism that made incremental, local scheduling decisions rather than global ones. Similar results for data sets with skew were recently presented in [21] where the authors describe a *dynamic* load balancing scheme based on a master-slave style approach, which is closely related to the approach in [26].

Perhaps the most significant previous work on parallelizing the data cube that has been reported in the literature has been undertaken by Goil and Choudhary [14, 13, 15] for the IBM SP2. (Recently, a similar approach based on cluster middleware was presented in [39].) The approach of Goil and Choudhary has two distinguishing features. First, views are individually partitioned across nodes rather than being clustered in groups. In theory, partitioning data in this manner can simplify the load balancing process since an equivalent portion of the input set can be distributed to each node. In practice, however, this kind of partitioning is complicated by the fact that non-uniform record distribution within the data set may make "perfect" partitioning impossible. Moreover, even if optimal partitioning could be guaranteed, a significant amount of merging is required in order to aggregate duplicate summary values from different processors. This requires considerable amounts of data to be communicated between processors. More importantly, the Goil and Choudhary algorithm uses a multi-dimensional OLAP (MOLAP) data cube representation. MOLAP systems store and manipulate the *logical* cubes as multi-dimensional arrays. The obvious advantage of doing so is that (1) only the aggregate values need to be stored and

(2) the arrays provide an implicit index for subsequent data cube queries. That simplifies the problem but as the number of dimensions increases, the size of the logical data cube grows to enormous proportions and various techniques must be used for data compression and indexing. Furthermore, a MOLAP based system is tied to special data structures and database management systems, and any data imported from a standard relational database system needs to go through a potentially very costly conversion phase.

Because of the two main problems with the previously presented solutions, performance/scalability and data accessibility, there is a strong motivation for research into high performance relational OLAP (ROLAP) data cube construction. ROLAP keeps the data in standard relational database format. Hence, no data conversion is required. Since all created views are standard relational tables, they can also be queried with standard relational methods. Our parallel data cube construction methods in [6] and [4] as well as the latest cgmCUBE prototype software presented in this paper are based on the following objectives: (1) Integration with standard relational systems, i.e. ROLAP. (2) Use of proven sequential algorithms for local ROLAP data cube computation. (3) Use of static global load balancing based on a carefully designed cost model. (4) Special attention to minimizing communication overheads and memory-disk transfers.

The case for ROLAP has been discussed above. Clearly, it is desirable that the parallel method re-uses as much as possible from the sequential methods available in order to perform local, sequential data cube computations. There is a large number of sequential ROLAP data cube computation methods available in the literature, including [1, 3, 19, 31, 30, 32] and more recently [22, 23, 33, 35, 36, 38]. As our sequential method for local ROLAP data cube computation we chose Pipesort [32]. Even though this is a fairly "old" method, we chose Pipesort over more recent methods for the following two reasons. (1) We chose to compute cubes with 100% precision rather than iceberg cubes, and in order to support full integration with standard relational systems, we require the views of the cube to be in standard table format instead of a compressed format as is the case in the more recent papers. (2) Among the remaining more traditional methods, Pipesort has the advantage over methods like BUC [3] or [30] in that it performs well for a larger range of data distributions. This is an important feature because the usefulness of the parallel ROLAP system would otherwise be rather restricted.

An important feature of our approach is the use of static global load balancing based on a carefully designed cost model. While in many cases dynamic scheduling is superior, this does not appear to be the case for parallel ROLAP data cube construction. Those previous methods that used dynamic load balancing [26, 21, 17, 27] observed a sharp drop in efficiency beyond eight processors and did not scale well. Our approach, on the other hand, uses static global load balancing and scales well, providing good speedup for larger numbers of processors [6, 4]. The key to the success of this approach is a carefully designed cost model which provides a good a priori measure of expected computational loads. A large part of our work on optimizing cgmCUBE was to improve on the cost model used. As outlined in the remainder, our new optimized cost model played an important role in the improved performance of cgmCUBE.

3 High Performance ROLAP Data Cube Construction Method Overview

We first give an overview of our parallel ROLAP data cube construction approach. Balancing the load assigned to different processors and minimizing the communication overhead are the core problems that need to be solved in order to achieve high performance and scalability. Our method uses an explicit cost model that provides a global description of the complete data cube generation process. As a result we are able to employ partitioning logic that equitably pre-assigns sub-tasks to processors across the network. Given a d-dimensional dataset S, our method for computing the 2^d views on a p-processor parallel machine proceeds as follows.

Step 1. *Construct the cost model:* We estimate the size of each of the 2^d views in the lattice and determine for each view the cost of directly computing its children. We use its estimated size to calculate (a) the cost of scanning the view (b) the cost of sorting it and (c) the cost of writing it to disk.

Step 2. *Compute the sequential schedule:* Using the bipartite matching technique presented in [32], we reduce the lattice to a spanning tree that identifies a cost effective set of prefix-ordered *scan pipelines*. In the remainder of this paper, a subtree of the data cube lattice used as a view generation schedule is referred to as a *schedule tree*.

Step 3. *Compute the parallel schedule:* We partition the schedule tree into $s \times p$ subtrees (s = oversampling ratio) and then define p subproblems, each containing s subtrees.

Step 4. *Compute the data cube in parallel:* We assign the p subproblems to the p processors. On each processor, we use the sequential PipeSort algorithm to build the local set of views.

It is important to note that the entire schedule tree is used when making load balancing decisions. In other words, we do not require dynamic workload migration since we bring together all relevant costing information before the computation and output of views actually begins. By extension, no inter-processor communication is required after the subproblem descriptions have been distributed. Given the size of the data cube, the effect of eliminating cuboid merge costs is very significant. Furthermore, the PipeSort model (both sequential and parallel) is designed to work directly with relational systems. The original data warehouse can reside in a standard DBMS and, if desired, the 2^d cuboids can be written back to the same database.

In the remainder of this paper, we will discuss each of the above steps in detail. Our parallel approach is based upon the sequential PipeSort algorithm [32]. PipeSort provides two primary components. First and foremost, it explicitly defines the algorithm for the creation of the *schedule tree*. Second, it provides a *description* of the process for building the view pipelines identified by the schedule tree. In this section, we discuss the implementation of both the sequential and parallel versions of the schedule tree generation algorithms. Pipeline computation will be discussed in detail in the next section.

The sequential schedule tree

PipeSort constructs a cost-augmented lattice that defines the computational options or trade-offs associated with data cube generation. It divides the "levels" of the lattice into a series of bipartite graphs and, working bottom-up, matches each pair of contiguous levels in order to identify the most efficient distribution of sort and scan orders that can be used to join level i to level $i-1$. The end result is a minimum cost spanning tree or schedule tree. This schedule tree can then be decomposed into a sequences of *scan pipelines*, each of which

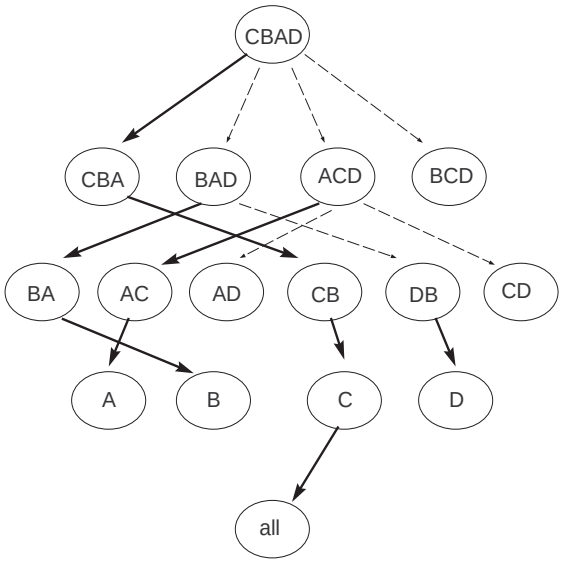


Figure 2: The PipeSort schedule tree. The prefix-ordered *scan pipelines* are bolded. The dashed lines represent sorting operations.

houses a subset of cuboids sharing a common attribute prefix. A single multi-dimensional sort and scan, at the level of the parent view in the pipeline, can subsequently be used to generate all cuboids with the same prefix order. Figure 2 provides a simple illustration for a four dimensional cube.

The parallel schedule tree

At the heart of our parallel method is a min-max k-partitioning algorithm [2] that decomposes the sequential schedule tree into subtrees. We note, however, that the standard min-max algorithm must be modified in order to work in the data cube context. First, a min-max k-partitioning does not provide a minimum cost (balanced) partitioning of the schedule tree which is an NP-complete problem. Second, the min-max k-partitioning algorithm [2] is designed for static graphs. In other words, it is expected that the edge weights remain fixed during the execution of the algorithm. In our case, that is not true. When a subtree S is extracted from the spanning tree for the purpose of network distribution, the root node of S cannot be computed from its designated parent since that parent will now be computed concurrently on another processor. Instead, the root view of S must be computed from the original input set. To capture this additional complexity, we maintain a *cut hash table* that records the current status/cost of the root edge of every partition. A simple illustration of the algorithm's adaptive costing can be seen in Figure 3.

Another important feature of our partitioning method is *oversampling*, that is the partitioning algorithm for p processors divides the schedule tree into $s \times p$ partitions, where s refers to the *oversampling factor*. While min-max k-partitioning produces an optimal partitioning of a weighted tree (i.e., the weight of the largest partition is minimized), this partitioning may still be far from balanced. By initially producing a multiple of p subtrees, we are able to combine these $s \times p$ subtrees into p subsets whose cumulative costs are much more evenly balanced. To produce these subtree combinations, we use an iterative technique that combines the largest and smallest trees with respect to construction cost.

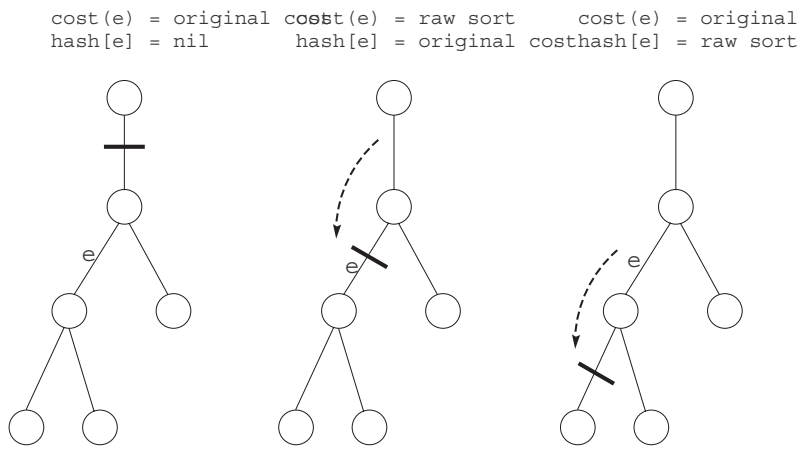


Figure 3: The use of a "cut" hash table to support dynamic min-max k-partitioning.

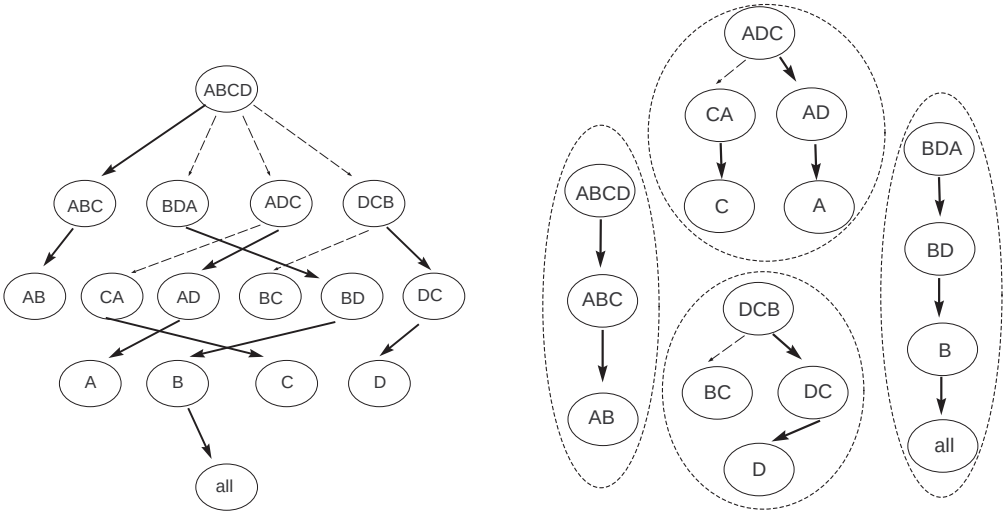
A description of these subsets is then sent to each of the p processors where a sequential pipeline computation algorithm constructs the views destined for the local node. Figure 4 shows an example of how a schedule tree is partitioned on a four processor machine. It is important to note that, as for sequential PipeSort, the views computed on each local node also comprise a sequence of prefix-ordered pipelines, which is an important property of our method.

4 cgmCUBE Software Architecture Overview

In total, approximately 8,000 lines of C++ code support the full data cube system. The development of the code base involved a number of independent programmers, several of whom were in geographically distinct locations. A number of third-party software libraries were utilized. Node-to-node communication is supported by LAM's Message Passing Interface (MPI) [25]. Thread functionality is provided by the Pthreads (POSIX Threads) libraries [29]. We also incorporated the LEDA Library of Efficient Data Structures and Algorithms graph libraries into our data cube code base [24]. We selected LEDA because of its rich collection of fundamental data structures (including linked lists, hash tables, arrays, and graphs), the extensive implementation of supporting algorithms (including bipartite matching), and the C++ code base. Though there is a slight learning curve associated with LEDA, the package has proven to be both efficient and reliable.

Having incorporated the LEDA libraries into our system, we were able to implement the lattice structure as a LEDA graph, thus allowing us to draw upon a large number of built-in graph support methods. In this case, we have sub-classed the graph template to permit the construction of algorithm-specific structures for node and edge objects. As such, a robust implementation base has been established; additional algorithms can be "plugged in" to the framework simply by (1) sub-classing the lattice template, (2) over-riding or adding methods and (3) defining the new node and edge objects that should be used as template parameters.

The final system, though large, has been constructed to be as modular and extensible as possible. A basic block diagram of the architecture is shown in Figure 5. Extensions to the core algorithms may treat the current system as a data cube *back-end*, using its



(a) (b)

Figure 4: (a) The schedule tree. (b) A four-processor partitioning.

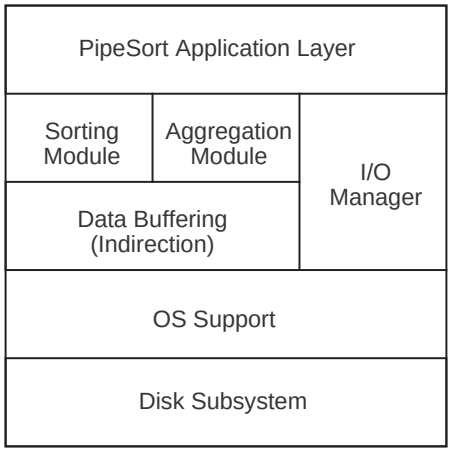


Figure 5: Software architecture of the cgmCUBE system.

core methods as an API. Because of the emphasis on fundamental software engineering principles, the current data cube engine is more than a mere "proof of concept". It is a robust parallel OLAP engine with the ability to support OLAP related research projects that build on it.

In order to support a robust experimental environment, we have written our own data generation subsystem. In fact, the data generator has evolved into a significant application in its own right. Because of the number of optional parameters that need to be passed to the generator application (e.g., number of records in the data set, the number of feature attributes, and the number of unique values in each dimension), coupled with the desire to maintain a permanent record of specific data set "schemas", we have constructed the data generator with the compiler tools FLEX and BISON [10], the GNU equivalent of LEXX (lexical analysis) and YACC (statement parsing). In so doing, we allow users to define a data set schema (in the form of an ASCII text file) that identifies data set parameters as *key/value* pairs. This file is parsed by the generator front-end to produce a final specification set that, in turn, is passed to the back-end component that actually produces the data set. In other words, we have defined a small data set specification language (DSSL) that provides tremendous flexibility in terms of defining data characteristics and proved to be very helpful in the performance analysis of our system.

5 Optimizing Pipesort

As discussed in Section 1, one of the important aspects of the performance optimization for cgmCUBE is the improvement of the underlying sequential cube construction method. The final performance of cgmCUBE is ultimately limited by the efficiency of the underlying sequential algorithms. Within the PipeSort context, this suggests that the algorithm for constructing each of the 2^d cuboids deserves special consideration. While the PipeSort authors described the algorithm for spanning tree generation in some detail, they provided very little information on the computationally expensive *pipeline* phase. Though it is true that the views of a given pipeline can be constructed in a single pass of the sorted input set, it is not at all obvious that this can be done as efficiently as one might expect.

In this section we examine the parameters, goals, and logic of the pipeline algorithm to determine whether variations in algorithm design or implementation can tangibly affect runtime performance. Our analysis has led to the identification of four *performance enhancements*, i.e. components of the pipeline algorithm whose improvement can significantly impact the overall runtime.

1. **Data Movement**. The PipeSort algorithm is data intensive. Because $O(2^d)$ views may be created, many of which will have entirely different attribute orderings (necessitating record permutations), the physical movement/copying of records and fields can become extremely expensive on large, high dimension problems.

2. **Aggregation Operations**. At the heart of the pipeline process is the aggregation of data into records of varying granularity. The existence of unnecessary or redundant aggregation operations results in a needless slowdown in pipeline execution.

3. **Input/Output Patterns**. The PipeSort algorithm moves enormous amounts of data to/from disk. In such environments, disk head movement and buffer flushing are a serious performance issue.

4. **Sorting**. The first step in pipeline processing is a sort of the input view. Since $\binom{d}{\lceil d/2 \rceil}$ is a lower bound on the number of sorts required to construct the full cube [30], a reduction in sort costs will be an important component in improving overall performance.

In the remainder of this section, we discuss our approach to the optimization of each of the performance limiters. Later in the paper, we demonstrate how these optimizations result in an order of magnitude improvement in overall performance.

5.1 Optimizing Data Movement

Since 2^d views of the data set need to be generated for a full cube computation, and each pipeline requires a different ordering of attribute values, the amount of data movement is a significant concern for an efficient PipeSort implementation. We identify two algorithm components that require particular attention. First, the sort used to construct the top of each pipeline, either Quicksort or Radix Sort, performs a large number of record movements during the course of execution. Quicksort moves records after each of its $\Omega(n \log n)$ comparisons, while Radix Sort must write the input list to a sorted output list following each of its k iterations. Second, considerable data movement is associated with pipeline processing because the pipeline aggregation process relies upon a prefix ordering that is by definition different from that of the input set (this is why a re-sorting is required). The order of the attributes in the records of the input set must be permuted to that of the parent view in the pipeline before aggregation can be performed.

Our goal in minimizing data movement is to eliminate record copying once the input set has been read into memory. To accomplish this objective, we have designed the algorithm to allow it to exploit *data indirection*. By indirection, we mean that instead of manipulating data directly, we instead manipulate *references* to that data. Within the pipeline algorithm, there are actually two distinct forms of indirection — and consequently, two types of references. In the first case, we utilize *vertical* or record-based indirection. Here, we maintain a table of pointers (memory addresses) to the records in the sorted input set. For any operation that would require a record as input, that record is obtained by following a pointer to the record's location in the original input buffer.

In the second case, we employ *horizontal* or attribute-based indirection. Here, we attempt to largely eliminate the costs associated with permuting attribute values within individual records. To do so, we maintain an *attribute indirection table* of size k that maps the *intended* position of each of the k attributes in the pipeline parent view to its *actual* position in the original record. Each pipeline has its own unique map. The cost of horizontal indirection is an additional offset calculation for each attribute access. We note, however, that during pipeline computation, the heavily accessed mapping table is likely to be held in the high speed L1/L2 processor caches, significantly reducing offset calculation costs. Data permutation, on the other hand, generates true core-memory to core-memory swapping. Experimental evaluation has shown that the cost of a full permutation of the input set is at least an order of magnitude more expensive than that of horizontal indirection.

Figure 6 graphically illustrates the use of both forms of indirection. A two-dimensional attribute reference is identified by first following the appropriate pointer into the input buffer and then mapping the dimension reference to the proper position in the record array. We note that although the input set does not *appear* to be sorted, it is nonetheless ordered

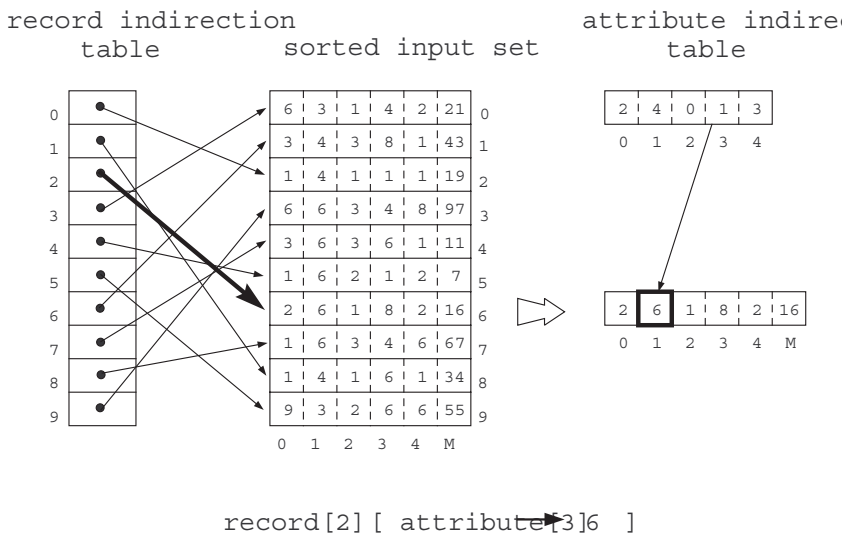


Figure 6: Resolving an attribute reference by way of vertical and horizontal indirection.

as per the requirements of the application (in this case by the attribute order defined in the attribute indirection table).

Note that both the sort routines and the pipeline methods utilize both vertical and horizontal indirection. Radix Sort and Quicksort are both implemented so as to work with record references. Changes to record positions, whether on the basis of record comparisons (Quicksort) or counting-oriented re-orderings (Radix Sort), are reflected by changes in reference positions. Once the input set has been (indirectly) sorted, the record indirection array is passed to the pipeline processing module as input. The pipeline code makes a linear pass through the reference table, using the attribute mapping table to identify values relevant to each granularity level in the pipe.

Formal analysis of the indirection mechanism in [9] indicates that for an input set of n records with k attributes to be sorted, vertical and horizontal indirection eliminates up to $n(k^2 + 2k + 2)$ attribute copies from *each* pipeline computation. In practice this cost reduction can be very significant. Consider, for example, an eight dimensional data set with 10 million records. Using a Radix Sort and indirection, the upper limit for data transfer costs was shown in [9] to be 8×10^7. In the absence of indirection, the potential cost balloons to 8.2×10^8, a difference of over 70,000,000 operations.

Figure 6 shows, for full cube construction, the benefit of indirection in terms of *relative improvement*. Relative improvement is defined as how much faster the optimized PipeSort is than the original PipeSort due to a particular enhancement (in this case indirection). For example, in computing the six dimensional data cube in Figure 6, the original PipeSort took 53.55 seconds, while the fully optimized PipeSort took 4.77 seconds for an overall relative improvement of 1123%. This overall relative improvement is due to the contributions of all four of the performance enhancements. The relative improvement due exclusively to the introduction of indirection is 51%, while the other three enhancements accounted for the remaining 1073% improvement.

We observe that in six and seven dimensions using indirection nearly halves the time required. In these cases efficient in-memory computation is critical. However, in generating

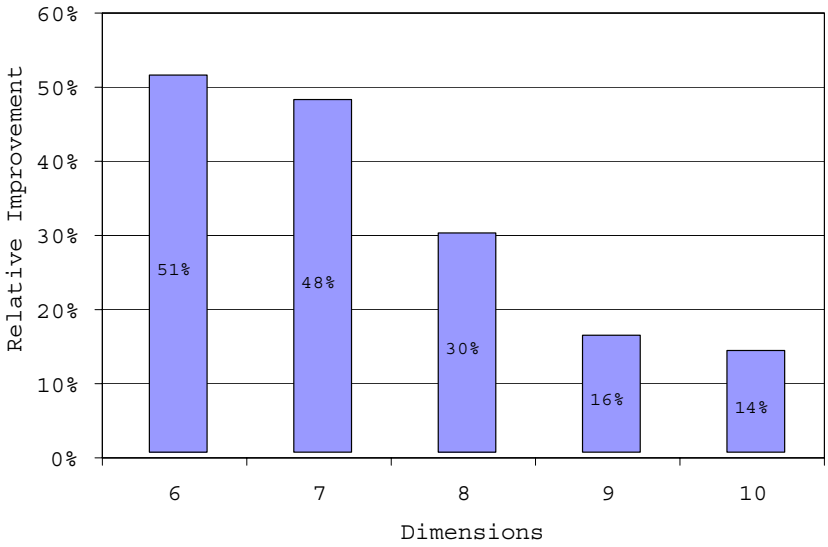


Figure 7: Relative improvement in performance due to indirection as a function of dimensions. (Fixed parameters: Data size n = 100,000 rows. Views selected = all.)

higher dimensional cubes the relative improvement is not so pronounced largely because the overall time taken is now dominated by increasing disk I/O costs.

5.2 Optimizing Aggregation Operations

Once the sorting phase has been completed, the input set is passed to the pipeline processing module. As per the description of the original PipeSort paper [32], only a single pass of the input set is required to generate each of the m views in a given pipeline. As previously noted, however, the details of a pipeline implementation were never provided. In fact, even though this component of the algorithm would appear to run in $O(n)$ steps, the overhead within each of the n iterations can be very large, with the result that a straight-forward implementation of the pipeline algorithm is be unacceptably expensive in practice.

To see why this might be the case, we need to examine what happens during each iteration. The algorithm compares two records during each step — the current record R_i and previous record R_{i-1}. It must examine the record R_i to find those changes in attribute values that might affect any of the m cuboids in the current pipe. For a given attribute d, a change in its value might or might not result in the creation of a new output record on a view V_i, where $1 \leq i \leq m$, depending on whether or not d is actually a component of V_i. Checking each of the d attributes to determine if it affects any of the up to d-attribute views in the m-length pipe produces a $O(d^2m)$ run time for the record checking phase. Doing this for each of the n input records quickly begins to erode the perceived benefit of a linear time algorithm.

The number of aggregation operations actually performed is another concern. Because we are iterating through n records, each of which will contribute its measure value to the subtotals of each of the m views, there is the potential for $n * m$ aggregation operations. For large n and large m — and multiple pipelines — the amount of CPU time dedicated to aggregation can be very significant.

Algorithm 1 presents a "lazy" pipeline algorithm that addresses these concerns. Essentially, the algorithm loops over each record, aggregating input records into output views

when required. In Line 2, it identifies the first attribute — working from the most significant to the least significant of the attributes relevant to the current pipeline — on which there is a change in column value. If there has been no change, the measure attribute is simply aggregated into the buffer of the parent view and it moves on to the next record. However, if there has been a change, it uses the *change position* j as an implicit indicator of *all* affected cuboids. For example, given that attribute checking proceeds from coarse to fine granularity in the prefix order, we know that a view with at least l feature attributes *must* be affected if $l >= j$. Conversely, for a "coarse" view with less then j attributes, we can be guaranteed that a change has not occurred on any of its attributes. We may therefore skip any further processing on this view. For each of the n records, then, the total number of conditional checks is simply the number required to find j, plus the number required to identify the affected cuboids. This $O(d + m)$ Pipeline Aggregation method is significantly more efficient than the naive $O(d^2m)$ design described above.

Algorithm 1 Lazy Pipeline Aggregation

Input: A sorted input set of n record and d dimensions, and an ordered set of k dimensions that form a pipeline prefix order.

Output: A set of m output views, each aggregated at a different level of granularity.

```
1: for all records in the input set do
2:    Find position of first change in current record
3:    if change position j == k then
4:       Aggregate measure value of parent view
5:    else
6:       perform lazy aggregation
7:    end if
8: end for
```

Returning to the issue of the number of aggregation operations, recall that $O(mn)$ summations may be required. In Lines 3-7 of Algorithm 1, we use a technique called *lazy aggregation* to dramatically reduce the required number of aggregation operations. In short, aggregation only occurs on any of the $m - 1$ views beneath the parent view when a change in attribute value has occurred on the *next coarsest* view. In other words, aggregation on these views is never done directly from the records of the input set. Instead, we use the knowledge that a change has occurred on view V_{i-1} in the pipeline (i.e., an *ancestor* view) to *trigger* an update on the running subtotal of V_i. The term *lazy aggreation* is indicative of the fact that we hold off on aggregating into V_i until absolutely necessary. In fact, for a given input data set and its associated pipeline, it is possible to show that lazy aggregation results in the optimal/minimal number of aggregation operations.

Again, a practical example may be used to illustrate the potential performance gain. Consider a data set of one million records, and a collection of k attributes, each with a cardinality of ten. For a view A, the naive form of aggregation would result in $n = 1,000,000$ distinct aggregation operations. With lazy aggregation, however, summing into A would only occur when triggered by record creation on its immediate parent, say AB. Since the size of AB is bounded by its cardinality product $\prod_{i=1}^{d} C_i$, we can see that the total number of aggregation operations required to compute A in this example is just 100. The Ph.D. thesis [9] contains a detailed implementation oriented description of Algorithm 1.

Figure 8 shows for full cube construction the benefit of using lazy pipeline aggregation over the straight-forward aggregation algorithm sketched in the original PipeSort paper

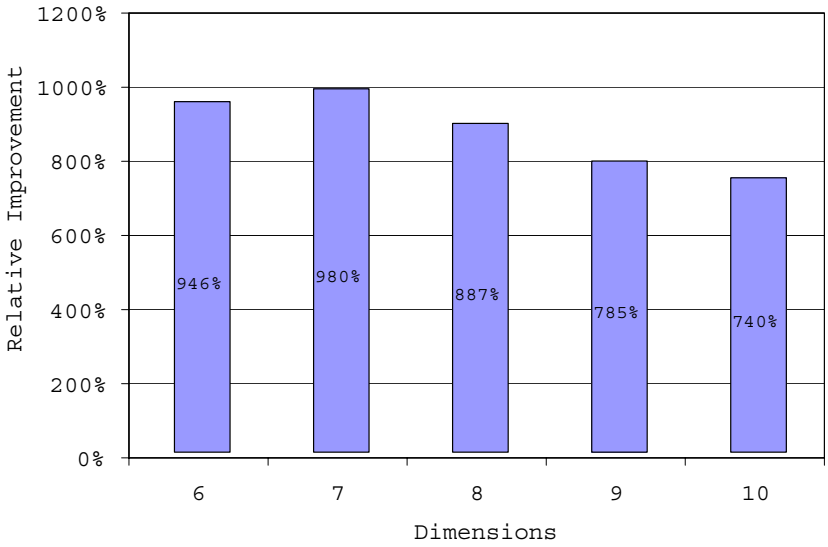


Figure 8: Relative improvement in performance due to the lazy pipeline aggregation algorithm as a function of dimensions. (Fixed parameters: Data size n = 100,000 rows. Views selected = all.)

[32]. We observe that the relative improvement due exclusively to lazy aggregation ranges between 740% and 980%. This makes lazy pipeline aggregation the most significant of our sequential PipeSort enhancements. In six and seven dimensions using lazy aggregation reduces by close to a order of magnitude the time required for cuboid generation. In generating higher dimensional cubes the relative improvement is somewhat less pronounced, largely due to increasing disk I/O costs, but is still very significant at over 700%.

5.3 Optimizing Input/Output Management

The preceding issues have been associated with computational performance. In this section, the focus shifts to I/O issues relevant to the view construction phase. We note that PipeSort is a very I/O intensive application. Specifically, in high dimensions the output size can grow to hundreds of times larger than that of the input. In this context, careful treatment of I/O issues can lead to significant overall performance gains. Conversely, without adequate attention to the execution and ordering of disk accesses, particularly write operations, the benefit of algorithmic optimizations may be seriously undermined.

We have identified two areas in particular that have the potential to affect PipeSort performance. The first is the problem of *disk thrashing*, a phenomenon that occurs when the operating system tries to do a series of small writes to a set of distinct files stored at different locations on the hard disk unit. The pipeline component of the PipeSort algorithm exhibits exactly this type of structure as each of the n iterations in the pipeline loop has the potential to generate m output records, each bound for a different file.

Modern operating systems attempt to limit the effect of this problem by *buffering* write operations. If the OS delays the write operations too long, we are presented with a second major problem — the operating system's caches eventually become so saturated that a massive amount of "house cleaning" is required to return the OS to a productive state. This wholesale flushing is extremely resource intensive and is a serious drain on performance.

We address these issues with the introduction of an I/O manager specifically designed for

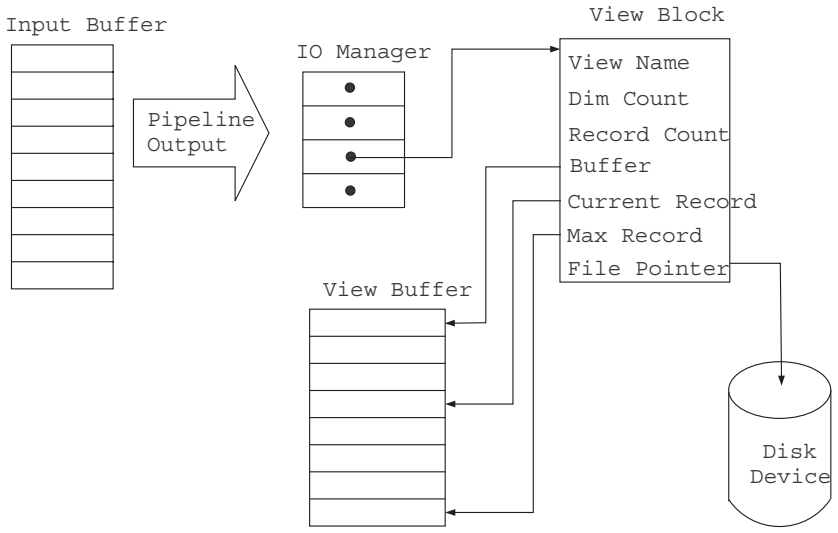


Figure 9: An illustration of the data cube I/O manager, showing the resources managed by one of its view "blocks."

PipeSort. Embedded directly within the PipeSort framework, the job of the I/O manager is to take over control of input/output operations from the operating system. Figure 9 illustrates the basic I/O subsystem. For a pipeline of length m, the I/O manager controls m "view blocks", each associated with the resources required to manipulate one cuboid. When a *logical* write is issued, the output record actually goes to the I/O manager which then places it into the next available slot in the appropriate view buffer. If the current buffer is full, then the I/O manager will instruct the operating system to write the entire buffer to disk.

For an output view of size j and an application buffer size of l, the use of an explicit I/O manager reduces the number of write operations — and potential thrashing instances — from j to j/l. However, because the operating system is still free to buffer the PipeSort's write operations, it is entirely possible that it will continue to do so until it is overwhelmed with back-logged I/O requests — our second major concern. To deal with this problem, we introduce a *throttling* mechanism that prevents the operating system from ever reaching this point. Specifically, when all processing for a given pipeline is completed, we issue a blocking *sync()* call. The sync (or synchronization) call flushes all application I/O streams so that the contents of any operating system buffers are physically written to disk.

We note, however, that the use of the sync-based throttle mechanism introduces an additional problem. Modern disk controllers employ a feature known as Direct Memory Access (DMA). In short, DMA controllers can assume responsibility for moving segments of data directly from memory to disk, thereby allowing the CPU to return to other, computationally demanding tasks. It should be clear that the use of a blocking sync call largely eliminates the benefit of DMA execution since the CPU stands idle as the I/O call executes.

We address this issue by recasting the PipeSort implementation as a multi-threaded application. In the PipeSort, the use of separate I/O and computation threads allows us to enjoy the benefits of DMA without the penalties of blocking sync calls. Specifically, we dynamically generate a new I/O thread at the completion of each pipeline. The sole purpose

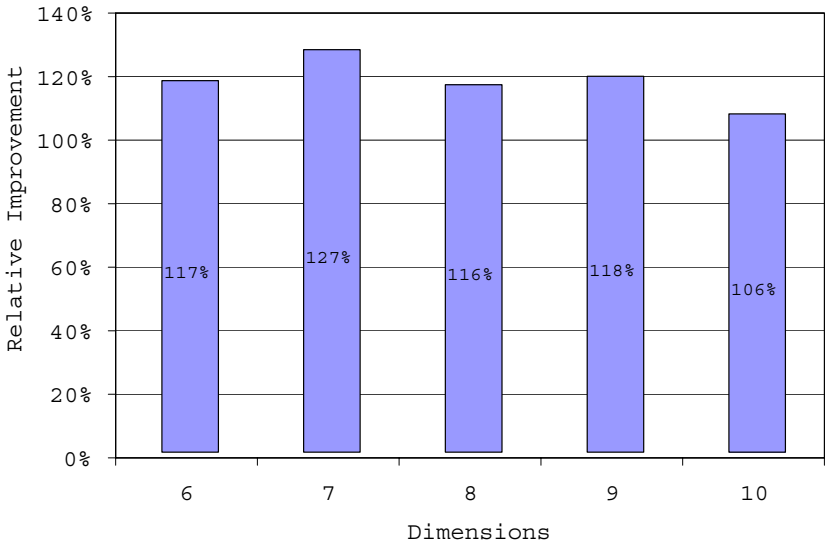


Figure 10: Relative improvement in performance due to use of an I/O manager as a function of dimensions. (Fixed parameters: Data size n = 100,000 rows. Views selected = all.)

of this I/O thread is to invoke the blocking sync call. While the new thread waits for the call to terminate, control returns to the original thread which then initiates execution of the *next* pipeline. In this way, we can concurrently process portions of two pipelines — one doing I/O and one doing computation — on a single CPU/DMA architecture. The end result is an I/O system that is well tailored to the application and the resources it runs upon.

Figure 10 shows, for full cube construction, the benefit of using an explicit I/O manager over the basic file I/O provided by the operating system. We observe that the relative performance improvement due exclusively to use of the I/O manager ranges between 106% and 127%. In all cases careful management of I/O halves the time required to generate a data cube. This is in large part due to the way in which the I/O managers overlaps computation and I/O. Even at six dimensions the I/O system is saturated so that the relative performance improvement derived from using the I/O manager remains basically constant even as the amount of I/O grows with the number of dimensions.

5.4 Optimizing Sorting Operations

The first step in processing each pipeline is to sort the m records of the pipeline's root view. This entails a multi-dimensional sort on $k \in \{1 \ldots d\}$ attributes. An obvious approach to performing this multi-key sort operation is to use a well optimized comparison sort like Quicksort [5]. Although Quicksort has a $\theta(m^2)$ worst case time bound, its average case performance is $O(m \log m)$ and it has low constants in practice. Note that in this application some dimensions may contain many duplicate values, so it is important that pivot selection method used by Quicksort be well optimized for this case. As long as this is done Quicksort behaves well, particularly when k, the number of attributes, is large. When k is small a k-iteration Radix Sort [5], which requires $O(kn)$, may outperform Quicksort.

Our experimental evaluation suggests that for a data set of d dimensions and n records, a k-attribute sorting is best performed with a Radix Sort rather than a QuickSort if $3k < \log n$ [9]. Our pipeline sorting mechanism therefore dynamically determines the appropriate sort

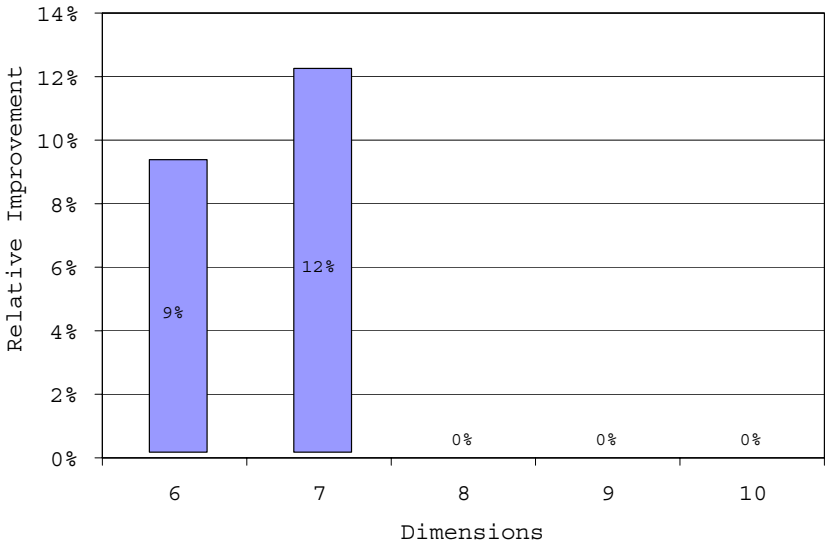


Figure 11: Relative improvement in performance for *full* cube computation due to adaptive sort selection as a function of dimensions. (Fixed parameters: Data size n = 100,000 rows. Views selected = all.)

for a view of a given size and dimension count by evaluating this expression for each input set. We note that while Radix Sort offers relatively little benefit for small problems, it becomes an increasingly valuable algorithmic component as the data sets become very large.

Figure 8 shows, for full cube construction, the benefit of adaptively selecting a sorting method over simply using Quicksort. We observe that in six and seven dimensions the relative performance improvement, due exclusively to adaptively selecting a sorting method, is around 10%, while in higher dimensions there is no observable improvement. The lack of improvement in higher dimensions is due to the fact that in such cases few of the root views of pipelines meet the criteria for calling Radix sort, and those few that that do tend to be too small to benefit. The benefit of this enhancement is more clear in the context of computing partial cubes. Figure 12 shows, for *partial cube* construction, the benefit of adaptively selecting a sorting method sorting over simply using Quicksort. Here portions, all views up to a fixed number of dimensions, of a ten dimensional cube were generated as is commonly done in, for example, visualization applications. We observe that when all views consisting of three dimensions or less are generated, the relative performance improvement due to adaptively selecting a sorting method is close to 50%, this drops to around 15% for views of five dimensions or less, and to zero beyond this point. The dynamic selection of a sorting method can be an important optimization, especially for partial cubes, but its value depends very much on the parameters of the problem.

5.5 Sequential PipeSort: Performance and Scalability

We now examine the performance benefits obtained by the integration of our four main PipeSort optimizations. Figure 13 shows for full cube construction that the relative performance improvement delivered by integrating these optimizations is 1123%, 1167%, 1033%, 919%, and 860% for 6, 7, 8, 9, and 10 dimensional cubes, respectively. These are impressive performance gains that dramatically decrease the time required to generate a cube of fixed size or, alternatively, allow significantly larger cubes to be generated in the same time

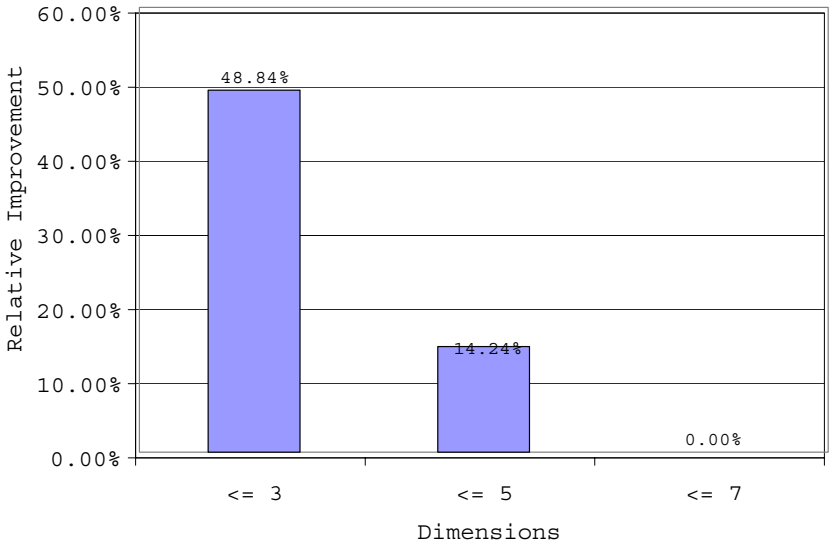


Figure 12: Relative improvement in performance for *partial* cube computation due to adaptive sort selection as a function of subset of views selected. (Fixed parameters: Data size n = 1,000,000 rows. Views selected = All views of 3d and less, 5d and less, and 7d and less, from a 10 dimensional cube.)

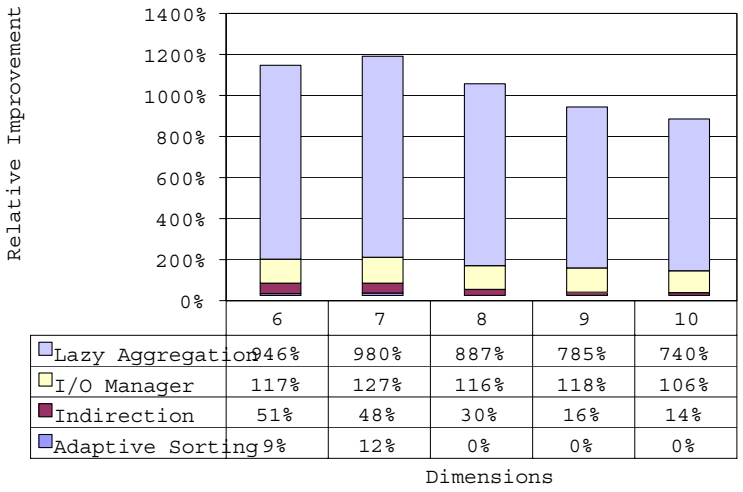
bounds. The key optimization is clearly lazy aggregation which accounts for a significant fraction of the performance improvement. However, especially in the case of computing partial cubes, (See Figure 14) the other optimization plays a very significant role accounting for relative performance gains of up to 375%.

In this section we have been primarily focussed on the relative performance improvement due to our optimizations, however is is also interesting to consider the absolute improvement in terms of time. Figure 15(a) contrasts the performance of the two sequential algorithms on 100,000 records, while Figure 15(b) looks at a data set of $1,000,000$ records. The curves are very similar in both cases and clearly demonstrate the runtime reductions achieved with the optimized algorithm. At ten dimensions, for example, the run-time for the original PipeSort algorithm is approximately 1000% greater than that of the Optimized algorithm (808 seconds versus 98.4 seconds on 10^5 records, and 7548 seconds versus 724 seconds on 10^6 records). Another way of saying this is that a single processor machine running the new algorithm (assuming adequate resources) would likely beat a 10-processor parallel machine running the original code.

6 Optimizing the Cost Model

In previous sections, we have assumed the existence of a costing model that populates the lattice with estimates of cuboid construction costs. Developing an accurate predictive costing model is an important and challenging task. Moreover, it is the accuracy of the cost metrics, as much as any other component of the system, that dictates the quality/balance of the workload distribution. Specifically, if the algorithm does not accurately represent the true cost of constructing pipelines, then there is no guarantee that the decomposed spanning tree will distribute equivalent workloads to each of the processors.

We note that the parallel costing model is in fact substantially different than the sequen-



	6	7	8	9	10
Lazy Aggregation	946%	980%	887%	785%	740%
I/O Manager	117%	127%	116%	118%	106%
Indirection	51%	48%	30%	16%	14%
Adaptive Sorting	9%	12%	0%	0%	0%

Figure 13: Relative improvement in performance due to the four performance enhancements as a function of dimensions. (Fixed parameters: Data size $n = 100{,}000$ rows. Views selected = all.)

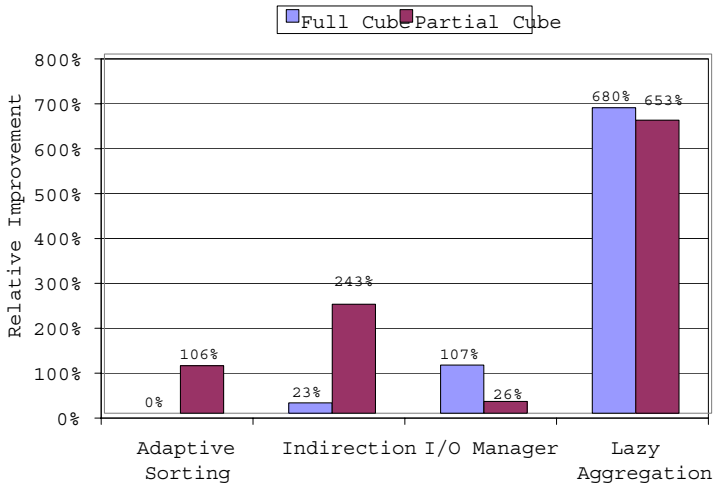


Figure 14: Relative improvement in performance due to the performance enhancements as a function of dimensions for full and partial cubes. (Fixed parameters: Data size $n = 1{,}000{,}000$ rows. Views selected = all and views of 3 dimensions and less.)

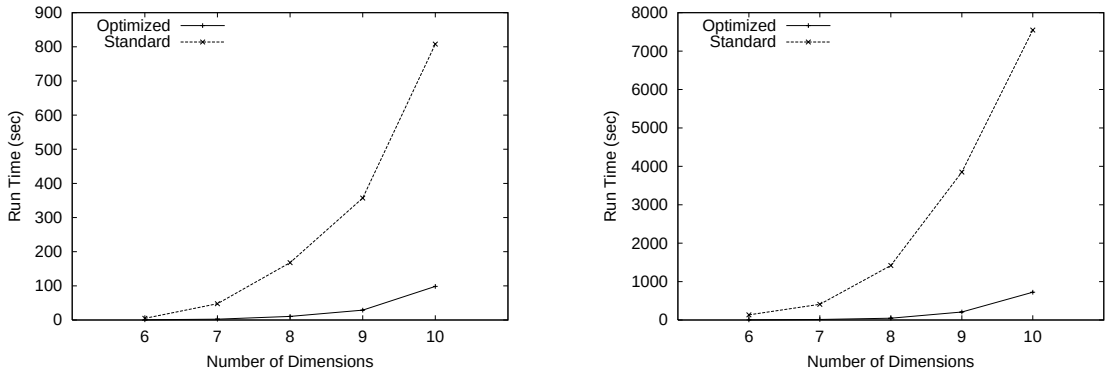


Figure 15: (a) Performance comparison for sequential PipeSort on 10^5 records. (b) The same comparison for 10^6 records.

tial model. In particular, the sequential framework only concerned itself with the relative costs of sorting and scanning. By incorporating these costs into a bipartite matching algorithm, it was possible to construct efficient pipelines. The designers of the PipeSort did not concern themselves with the I/O costs of the data cube, even though the enormous size of the output makes such costs substantial. They did not do so because in a sequential environment the I/O costs are fixed. In other words, no matter how a view is created (e.g., ABCD versus DBCA), it always contains the same number of records and, consequently, will always have the same I/O cost. While this is also true in parallel, note that computation is no longer associated with a single processor. Instead, just as each processor will do its own pipeline computation (sorts and scans), it will do its own I/O as well. For this reason, good load balancing can only be achieved if we can incorporate I/O costs into the costing model. In practice, the resulting parallel cost model is significantly more complex than its sequential counterpart.

For illustrative purposes, we break the costing infrastructure into two basic phases. The first deals with the estimation of cuboid sizes while the second takes these size estimates and determines pipeline construction costs. In this section we take a detailed look at both components of the framework.

6.1 Cuboid Size Estimation

Since the cuboids that populate the lattice have yet to be created, we must estimate their final sizes. This is in fact a very difficult thing to do accurately in a data cube context since we know nothing about the distribution of data in the fact table. Nevertheless, a number of techniques for size estimation have been described in the literature [20, 34]. The simplest of these involve (1) the *cardinality product* and (2) *sample scaling*. However, while the product of the dimension cardinalities provides a crude upper bound for view estimates, it tends to dramatically over-estimate the sizes for high dimensional views. By contrast, sample scaling takes a small sample of the input set, computes a cube and then scales up the estimates. In practice, however, it tends to significantly underestimate the number of duplicate records that will be found in large data sets.

In [34], the authors propose a view estimation method that builds upon the Probabilistic Counting algorithm first presented in [12]. The Counting-based method works as follows. During a single linear pass over the data set, it concatenates the d-dimension fields into bit-vectors of length L and then hashes the vectors into the range $0 \ldots 2^L - 1$. The algorithm then uses a probabilistic technique to count the number of distinct records (or hash values) that are likely to exist in each of the 2^d cuboids. To improve estimation accuracy, a universal hashing function [5] is used to compute k hash functions that, in turn, allow the algorithm to average the estimates across k counting vectors. The end result is a method that can produce very accurate cuboid size estimates (with a bounded error), even for data sets with significant skew.

We have implemented the algorithm in [34] and verified its accuracy. For example, the algorithm produces estimation error in the range of 5–6 % with 256 hash functions. However, its running time on large problems is disappointing. Specifically, despite an asymptotic bound of $O(n * 2^d)$, the constants hidden inside the inner computing loops are quite large. For the small problems described in previous papers, this is not an issue. In high dimension space, the running time of the estimator extends into weeks or even months.

Considerable effort was expended in trying to optimize the implementation of the algorithm. Despite a factor of 30 improvement in running time, the algorithm remained far too slow. We also experimented with the GNU-MP (multi-precision) libraries in an attempt to capitalize on more efficient operations for arbitrary length bit strings. Unfortunately, the resulting estimation phase was still many times slower than the construction of the views themselves. At this point, it seems unlikely that the counting-based estimator is viable in high dimension space.

Currently, we are using simpler statistical techniques to estimate view sizes. The fundamental idea is based on a probabilistic analysis that examines the cardinalities of the attributes of the data set. Specifically, the goal is to accurately estimate the number of duplicate records likely to be generated for a specific set of attributes. While we have developed our own method during the course of implementing the larger system, the spirit (and results) are very similar to those described in [11]. It is important to note, however, that these techniques do not actually examine the contents of the data set, only the cardinality data. As a result, they are limited in their capacity to capture data-specific skew patterns, an important problem in practice. Cost effective estimation therefore remains an important area of open research.

6.2 Pipeline Cost Estimation

In the previous section, we described how the sizes of views in the lattice might be estimated. In this section, we describe a cost model that, given estimates of view sizes, predicts the time required to generate those views. The model takes into account the contribution of the three fundamental processes: (1) input/output, (2) pipeline scanning, and (3) sorting.

When modelling the costs of the algorithm, we must strike a balance between *mapping granularity* and *model transparency*. In other words, our costing framework must be sufficiently detailed to capture the salient features of the algorithms, but not so detailed that it becomes unwieldy and difficult to modify if there are changes to the algorithm implementation, supporting libraries, or hardware. In the remainder of this section, we provide a concise description of the current data cube costing infrastructure.

6.2.1 Input/Output

The I/O metric is the one component of the cost model that is heavily influenced by the architecture of the machine being used. Specifically, I/O costs are intimately related to the hardware and supporting software upon which the data cube implementation actually runs. As such, it is necessary to experimentally define the relative impact of reading/writing data to disk. That being said, the impact cannot be defined in "absolute" terms; it must instead be a *relative* definition that can be measured against the scanning and sorting costs.

To represent I/O impact, we define what we call the *Write I/O Factor*. This expression represents the degree to which a "write" to the local physical disk unit is more expensive than a comparable write to a RAM (Random Access Memory) buffer. In a generic application environment, such a measure would be virtually impossible to capture given the impact of random disk head movements and OS intervention. Recall, however, that our I/O manager assumes control of application input and output and reduces disk access to a finely coordinated sequence of streaming reads and writes. We can therefore accurately capture the I/O factor by experimentally comparing the cost of a RAM-to-RAM "write" versus a RAM-to-disk "write" for data sets of sufficient size (small writes can be dominated by the initial disk head movement). The Write I/O Factor for our local Linux cluster, for example, is 120. In other words, for streaming writes the cost of sending data to disk is a little more than two orders of magnitude greater than that of accessing the same records in a memory buffer.

Since each individual record is composed of k feature attributes and a single measure value, the write cost for a k-dimensional data cube view with an *estimated* size of n is equivalent to $n(k+1) \times Write\,IO\,Factor$. The reader will note that this form of I/O metric can be deemed "relative" in that it is defined only in terms of n and k, parameters that will also be used by the sorting and scanning cost functions.

6.2.2 Scanning

Though conceptually we think of scanning as representing the process of linearly passing through a parent view in order to compute a child, this is in fact not the case. Recall that the pipeline algorithm uses a single scan of the n records of the input set to compute all m child views in the pipeline. While a simple scan metric that incorporates the values of n and m might seem to be an obvious solution, we cannot model scan costs in this fashion because during the spanning tree construction and partitioning phases, we do not actually know how many views will end up in a given pipeline. In other words, we do not know the value of m.

We deal with this problem by decomposing the scanning process into two components. First, there is the portion of the scan cost that is associated with passing through the input view and identifying the i^{th} attribute in the set of k dimensions that represents an attribute change on contiguous records. We will in fact incorporate this cost into the sort metric (discussed below). Second, there is the component of the scan that is associated with identifying a new record for view V (based upon the position of the attribute change), and moving that aggregated record to the I/O manager's output buffer. For a view V of size n, this cost is exactly equivalent to $n(k+1)$. Furthermore, this cost does not depend upon the number of views that end up in the final pipeline. Instead, each view must *always* be associated with this cost, regardless of its position in the pipeline or how it was created.

6.2.3 Sorting

Each pipeline is associated with a sort on some input view. We have described the process of dynamically choosing the appropriate sorting algorithm (Radix Sort or Quicksort). However, estimating the sort cost involves the computation of two other metrics. First, we must include the read cost of the input view, defined as $n(k+1) \times Read\ I/O\ factor$ (note that the $Read\ I/O\ factor$ might be slightly different than the $Write\ I/O\ factor$.) Second, we must capture the cost of finding the position of the first attribute that differs in contiguous records, i.e. the second component of the pipeline scan. While we cannot know where that position will be for an individual record (e.g., $1 \leq pos \leq k$), we can define the average or *amortized* position as $k/2$. In other words, we expect that, on average, we will have to look at half of the k attributes to find a change. The full cost of this portion of the scan is therefore $n * (k/2)$. We will refer to this cost as the *dim check* metric. Putting the three sort components together, we have $sort = read\ cost + (Radix \mid Quicksort) + dim\ check$.

We note that for the Standard pipeline implementation, costing of the Radix and Quicksort algorithms would be quite cumbersome due to the effect of data movement. Recall, however, that the use of vertical and horizontal indirection virtually eliminates such costs, allowing us to work with the much more manageable and more familiar $3kn$ and $n \log n$ metrics. Thus, the fully materialized sort metric is defined as:

$$(n(k+1) \times Read\ I/O\ Factor) + ((3kn) \mid (n \log n)) + (n \times (k/2))$$

6.3 Putting it all together

While the previous sections have defined the cost functions that make up the larger model, the careful reader may have noticed that there is still one slight problem with the direct application of these metrics. Specifically, the I/O manager uses separate computation and I/O threads to exploit the functionality of modern DMA controllers. The result is that I/O and computation are to a large degree overlapped. We note that this does not mean that we can ignore I/O costs. Doing so would indicate to the bipartite matching algorithm that I/O costs were effectively zero — an assumption that is most definitely untrue — leading to a partitioning of work that would likely be poorly balanced.

At first glance, it would seem difficult to incorporate this knowledge into the model. Specifically, the I/O costs for a given partition are not known until the spanning tree has been cut; however, the cutting process itself relies upon accurate knowledge of the I/O costs. We deal with this problem by further augmenting the min-max partitioning algorithm. To accurately represent the construction costs in the new multi-threaded design, we define the *pipe weight* $\sum_{i=1}^{l} e_{pipe}$ as the sum of the sort/scan edges in a *candidate partition* and the *I/O weight* $\sum_{i=1}^{l} e_{IO}$ as the sum of the I/O costs. We then define the *actual partition weight* as $max\{\sum_{i=1}^{l} e_{pipe}, \sum_{i=1}^{l} e_{IO}\}$. In other words, for the current partition we dynamically decide which of the two components, each with its own execution threads, will dominate or bound the cost of the current computation. Since the execution threads almost completely overlap, we know that the smaller of the two costs will be mostly absorbed by the larger cost and will have little effect upon final run-time.

Figure 16 provides an illustration of the complete model for a sample partition defined on a four-dimensional space, including the cost metrics for each of the relevant computational components. The following costing features can be observed:

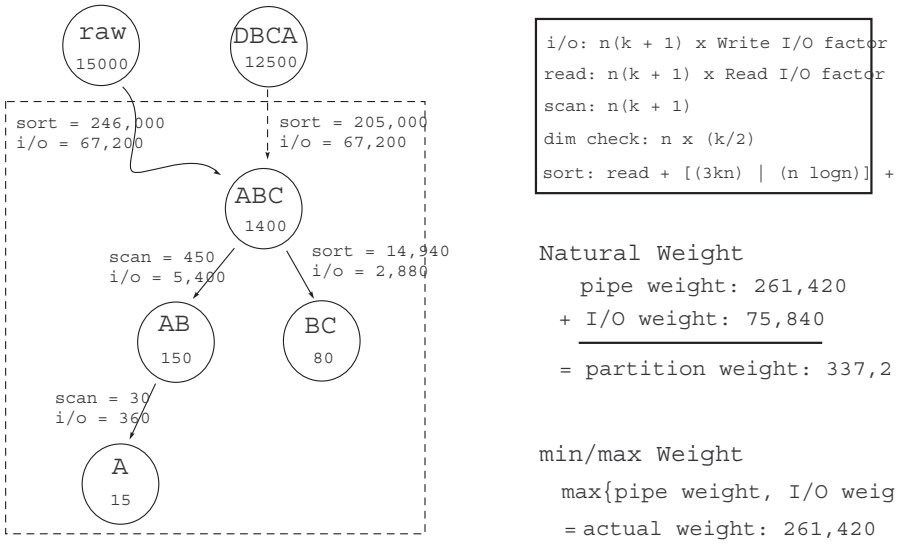


Figure 16: Dynamic cost calculations for a sample partition from a four-dimensional space. The numeric values inside each view represent the estimated sizes.

- First, note that the existence of the "parent" cut requires that the "raw" data set be used as input for the initial sort.
- Edges from prefix-ordered pipelines are associated with scan and I/O costs, while non-prefix views are associated with sort and I/O costs.
- The *actual partition weight* in this example is bounded by the pipeline costs.

Finally, we note that the reader should not assume that the relative weights in this particular example are indicative of general trends. Each data cube problem is unique, and the size and nature of the data set, coupled with the number of feature attributes, will produce different partitioning patterns.

6.4 Parallel Scalability and Speedup

As described earlier, an accurate cost model is key to achieving good parallel throughput, speedup, and scalability. If the cost model accurately estimates the size of cubiods and correctly predicts the relative costs of sorting, scanning, and I/O, then the decomposition into parallel subtasks based on it may lead to good parallel speedup and scalability. If not, e.g., if the estimates are off by more than a few percent, then the parallel speedup will be poor, little scalability will be evident, and throughput will be low. The performance analysis in this section examines these issues using two experimental platforms, namely a distributed memory Linux cluster and a shared disk Sunfire cluster.

1. Linux Cluster. The Linux system is a 64-processor (1.8 GHz), 32-node Beowulf configuration, with a 100-Mb/sec Ethernet based network. Each node has its own pair of 40 GB 7200 RPM IDE disks. (We note, however, that to simplify the interpretation of results, only one CPU and one disk were utilized on each node.) Every node was running Linux Redhat 7.2 with gcc 2.95.3 and MPI/LAM 6.5.6 as part of a ROCKS cluster distribution.

2. SunFire 6800 Cluster. The SunFire 6800 is a 24-processor shared memory SUN multiprocessor that runs the Solaris 8 operating system. (We use this machine in a distributed memory fashion. There is no change in code from that used on the Linux machine.) Our SunFire 6800 uses SUN 900 MHz UltraSPARC III processors with one GB of RAM per CPU. Finally, the SunFire comes equipped with a SUN T3 shared disk array.

In the following experiments all sequential times were measured as wall clock times in seconds. All parallel times were measured as the wall clock time between the start of the first process and the termination of the last process. We will refer to the latter as parallel wall clock time. All times include the time taken to read the input from files and write the output into files. Furthermore, all wall clock times were measured with no other user except us on the clusters.

We will look at a sequence of data cube tests, each designed to highlight one important characteristic or parameter of the model. In effect, we utilize a set of base parameters and then vary exactly one of these parameters in each of the tests. These base parameters are (with defaults listed in parenthesis):

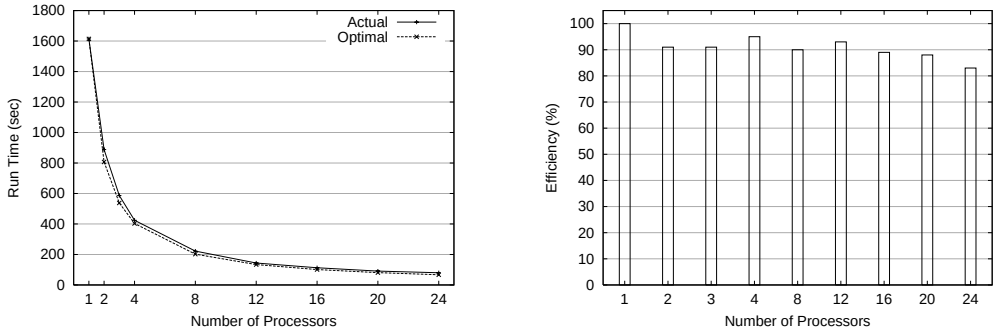
1. Processor Count (16)
2. Fact Table Size (2 million rows)
3. Dimension Count (10)
4. Over-sampling Factor (2)
5. Skew (uniform distribution)

When default values are not, or cannot, be used for a particular test, this fact will be clearly noted. Finally, we add that each point in the graphs represents the mean value of three separate test runs. Note that in general the variation between runs was less then 3%.

Finally, to assess the peak throughput of the cgmCUBE system we focussed on generating large high dimensional cubes. For example, we generated data cubes using a 14-dimensional, 10 million record data set. The resulting cubes were up to 1.2 TeraBytes in size, contained 16,384 cuboids, and were produced in under 72 minutes on a 24-processor Linux cluster. In general we were able to produce data cube output at a rate of over one TeraByte per hour.

6.5 Speedup

Figure 17 shows, for full cube construction, the parallel wall clock time in seconds as a function of the number of processors on a Linux cluster and corresponding parallel efficiency. The input set consists of 2 million 10-dimensional records. Note that for the "shared nothing cluster" a copy of this input set is initially available on each local disk. Figure 17(a) depicts the performance curve. Also shown is the optimal curve, calculated as $T_{opt} = T_{sequential}/p$. Note that the actual performance tracks the optimal curve closely. For 24 processors, our method achieves a Speedup of 20.18. In Figure 17(b), we provide the corresponding efficiency ratings. With smaller processor counts (i.e., $p \leq 12$), efficiency values lie somewhere between 90% - 95%, while for processor counts beyond this point, the ratings are between 83% and 90%. Given the complexity of the data cube generation problem, these results



(a) (b)

Figure 17: Parallel wall clock time in seconds as a function of the number of processors on a Linux cluster and (b) corresponding parallel efficiency. (Fixed parameters: Data size $n =$ 2,000,000 rows. Dimensions d = 10.)

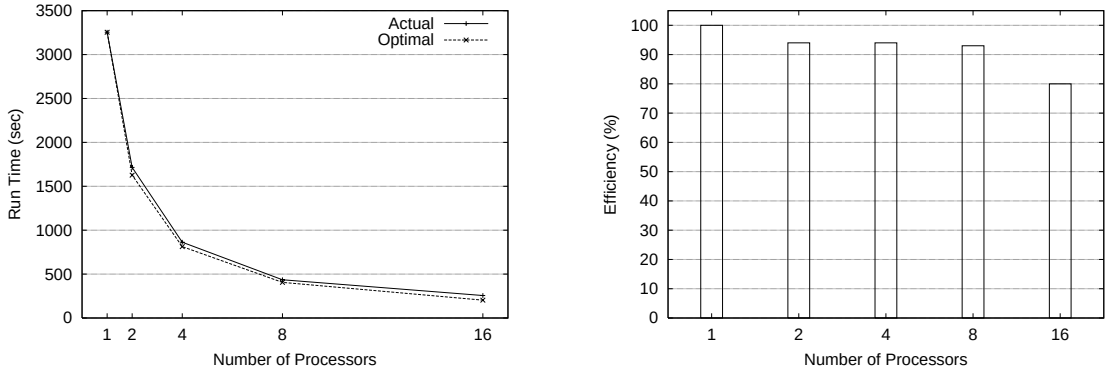
represent an extremely efficient use of parallel resources. Moreover, the relatively mild decrease in efficiency at 16+ processors suggests that even higher processor counts are likely to provide acceptable efficiency.

We now examine parallel cube generation on the SunFire 6800. Note that these tests utilize 16 of the SunFire 6800's 24 processors as these were all that were available to us. With the disk array, it is important to note that only a single instance of the base table is required. Figure 18(a) shows the Speedup curve, while Figure 18(b) provides the efficiency ratings. From two to eight processors, the efficiency again exceeds 90%. At 16 processors, efficiency slips to about 80%. Since this is slightly more noticeable than the decrease on the Linux cluster, it begs the question, "Is this an algorithmic issue?" Our analysis suggests that the answer is likely no. Rather, it is the nature of the SunFire disk array itself that is beginning to affect performance. Specifically, the disk array houses a fixed number of independent disk units. This creates a bottleneck at high processor counts because the parallel system's ability to reduce I/O costs is more limited than its ability to reduce computation costs (i.e., fewer disk heads than processors).

These results, would suggest that shared disk data cube implementations would be best suited to parallel settings with a small to moderate processor count. We note that, in practice, this observation would not likely be seen as a significant restriction. Specifically, because data warehouses would almost certainly be constructed as dedicated systems, unlike the multi-purpose MPPs often seen in scientific computing environments, small to moderate parallel systems would often be the platform of choice.

6.6 Data Set Size

In this section, we analyze the effect on performance as we increase the size of the input set, holding other parameters constant. Figure 19 depicts the running time on the Linux cluster for data sets ranging in size from 500,000 records to 10,000,000 records. Before examining the curve, we note the following. As we increase problem size, the performance of the I/O component of the algorithm will effectively increase at a linear rate since it essentially performs streaming writes to disk. On the computational side, however, run-



(a) (b)

Figure 18: (a) Parallel wall clock time in seconds as a function of the number of processors on a Sunfire cluster and (b) corresponding parallel efficiency. (Fixed parameters: Data size n = 2,000,000 rows. Dimensions d = 10.)

time is bounded by the time to perform sorting. Recall that we may either use QuickSort of Radix Sort for this purpose. In higher dimensional space (e.g., the 10-dimensional set used for the test), Quicksort will generally be chosen for smaller data sets, but will give way to Radix Sort once the sets become very large. Thus, we would expect to see a slightly super-linear growth pattern for the smaller sets (due to the $O(n \log n)$ Quicksort), but then a more or less linear pattern after a certain point (due to the asymptotically linear bound of Radix Sort).

Returning to Figure 19, we see a curve that closely resembles the expected shape. On small sets, there is a modest super-linear increase in run-time. Between 5 million and 10 million records, however, we see a curve that is almost perfectly linear in shape. At this point, almost all of the sorting is performed by Radix Sort. Consequently, the curve begins to flatten out. This, in fact, is a very encouraging sign for data cube implementations that will deal with even larger fact tables.

6.7 Dimension Count

In this test, we examine the effect on increasing the number of dimensions for data sets with a fixed row count (the default of 2,000,000). Figure 20 illustrates the effect of increasing the number of dimensions in the problem space from six to 14. Recall that an increase of one in the dimension count corresponds to a doubling in the number of views to be produced. Interestingly, the general shape of the curve is quite similar to that of the data set size test (Figure 19). The explanation for this, however, is slightly different. In this case, the combination of data set size and dimension count is likely to favour the QuickSort for most of the large sorts. It is important, incidentally, to distinguish between large and small sorts because even though most of the views in the lattice would have smaller dimension counts and would thus be sorted with Radix Sort, it is the large views at the top of the pipelines that dominate overall sorting costs. In any case, based upon this observation one might expect this curve to continue to increase at a super-linear rate. It does not do this because I/O costs become increasingly important as we pass 10 dimensions. This is the case

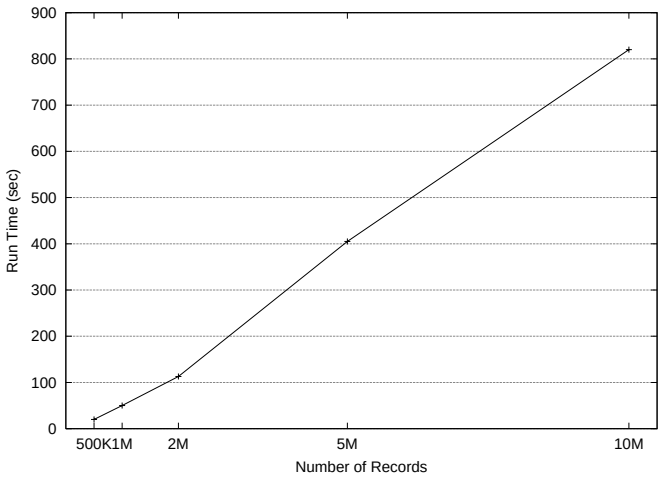


Figure 19: Parallel wall clock time in seconds as a function of the number of records in the input.

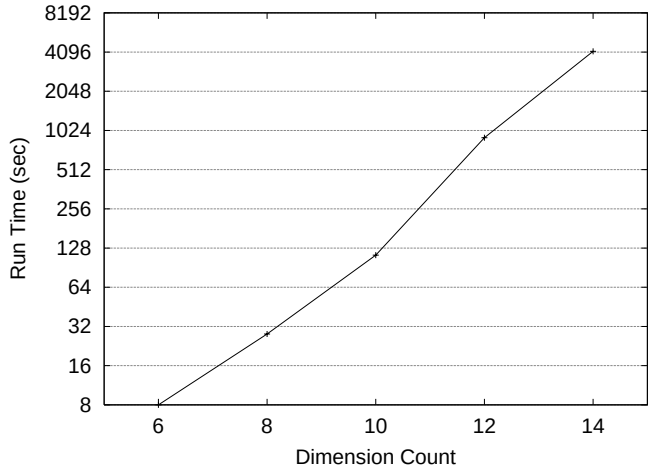
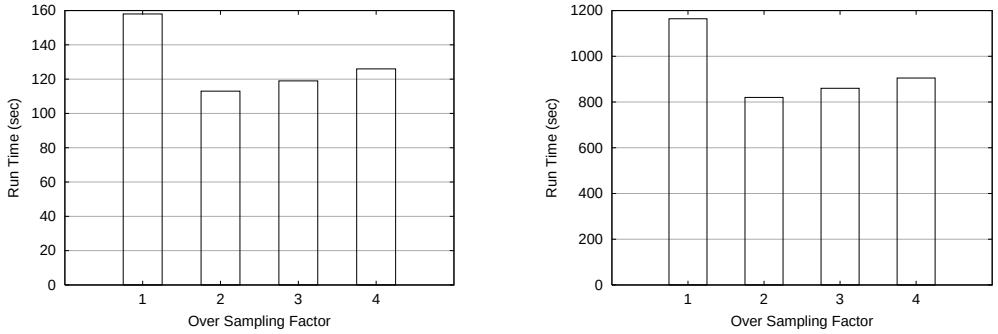


Figure 20: Parallel wall clock time in seconds as a function of the number of dimensions.

because in high dimensional space a large percentage of the views are almost as large as the original fact table. Given the significant penalty attached to writing all of this data to slow secondary storage, the I/O phase of the algorithm begins to dominate the in-memory computation. Since the I/O in this system is strictly linear (i.e., it is streaming, not random access), the growth curve again begins to flatten out. We may therefore conclude that for significantly large problems, either in terms of data set size or dimension count, increases in input parameters result in a roughly linear increase in run-time performance.

6.8 Over-Sampling Factor

Recall from Section 3 that the use of a sample factor represents an attempt to reduce run-time by improving the quality of workload partitioning. Figure 21 provides experimental evidence that our approach is justified. In both test cases, we vary the sampling factor *sf* from one (i.e., no over-sampling) to four. Figure 21(a) presents results for the default data



(a) (b)

Figure 21: Parallel wall clock time in seconds as a function of over-sampling factor on (a) two million records, and (b) 10 million records, respectively.

set of two million records, while Figure 21(b) looks at 10 million records. The results are virtually identical. In both cases, the use of an over-sampling factor, with $sf = 2$, reduces the *base* run-time by approximately 35%. This reduction in run-time is entirely the result of improved partitioning. For larger values of sf, however, performance actually begins to decline. There are two reasons for this. First, as previously noted, each additional partition on each compute node is associated with a slightly more expensive sort of the raw data set. Second, as we continue to partition the original tree, we create subtrees with increasingly shorter pipelines. The effect is to reduce the degree to which sorts can be shared by multiple views. As such, it appears quite likely that an over-sampling factor of two will be optimal for virtually all practical parallel architectures.

6.9 Record Skew

In this section we present tests that are designed to illustrate the impact of record skew on the algorithm's costing model. Recall that the current size estimator assumes a uniform distribution of data. One would expect then that as the values of each dimension become increasingly skewed (i.e., many more occurrences of some dimension values than others), the size estimates would become increasingly inaccurate. By extension, load balancing decisions would suffer, resulting in larger run-times. Figure 22 presents performance results for data sets that have been created with varying degrees of skew. We note that skew is produced with a *zipf* function [41], a technique commonly employed in the data cube literature [34, 3]. In this case, $zipf = 0$ corresponds to no skew, $zipf = 0.5$ to moderate skew, and $zipf = 1$ to heavy skew.

As the graph demonstrates, there is relatively little difference between the results for uniformly distributed data and those of the skewed sets. In fact, there are two reasons for this. First, though estimation quality does decrease, resulting in suboptimal load balancing, this degradation is partly offset by reductions in I/O costs for skewed data. Specifically, the introduction of significant skew creates more clustered data cube output, thereby reducing view size and, in turn, reducing run-time. Second, the use of the zipf function for data cube input sets tends to create a very regular pattern of skew. In particular, all dimensions are skewed to the same degree, with the result that the associated errors in size estimation

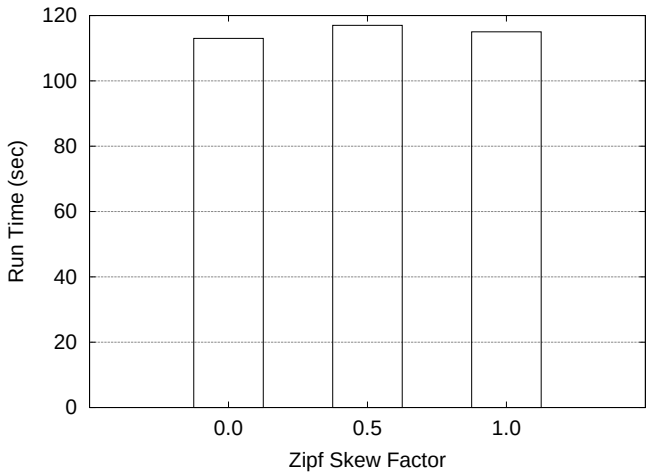


Figure 22: Parallel wall clock time in seconds as a function of the number skew.

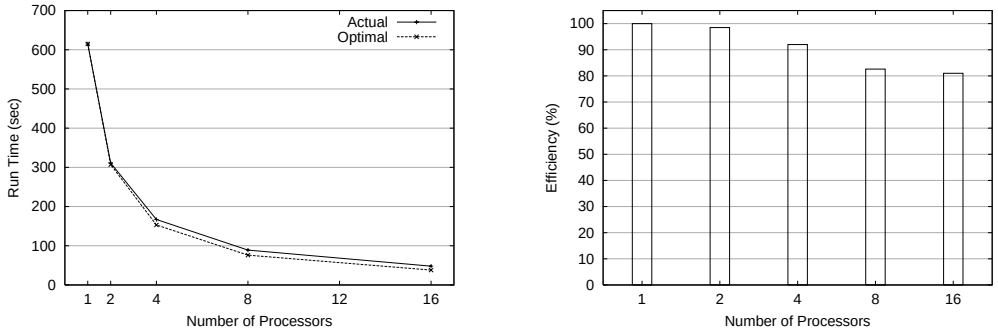
tend to be "amortized" across the network. In other words, costing errors are unlikely to be disproportionately associated with any one node. The result, at least with this form of synthetic skew pattern, is that the algorithm handles estimation errors somewhat better than one might expect.

Having said that, it is clear that not all data sets would exhibit this kind of regularity in their skew patterns. Specifically, real world data sets are much more likely to have arbitrary clusters and pockets of skew that might lead to more irregular workload partitioning. We have therefore obtained a one million record data set containing information on weather patterns. While not a typical OLAP subject, the weather set makes an appropriate test set in that it consists of categorical attributes of reasonably low cardinality, and it has a meaningful "total" field that can serve as the measure attribute. In our case we have extracted the following 10 feature attributes from the 20-attribute records (cardinality in parenthesis): hour of day (24), brightness (2), present weather (101), lower cloud amount (10), lower cloud base height (11), low cloud type (12), middle cloud type (13), high cloud type (11), solar altitude (1800), and relative illuminance (218). The measure attribute is total cloud cover (9).

Figure 23(a) depicts the parallel speedup on the weather data set for one to 16 processors (both actual and optimal), while Figure 23(b) provides the efficiency ratings. Observe that as the processor count increases to 16 processors, there is a slightly more noticeable decline in efficiency than was seen in the original cluster testing in Section 6.5. We believe this to be the result of estimation error that is not evenly distributed to each of the p nodes. Nevertheless, given that the current size estimator is not designed to handle this situation, parallel efficiency still exceeds 80% at 16 processors. We are currently investigating estimators with better performance in the presence of skew.

7 Conclusion

In this paper, we have described a new parallel method for the parallelization of the data cube, a fundamental component of contemporary OLAP systems. We have described an approach that models computation as a task graph, then uses graph partitioning algorithms to distribute portions of the task graph to each node. Locally, a heavily optimized pipeline



(a) (b)

Figure 23: Parallel wall clock time in seconds as a function of (a) the number of processors, and (b) the corresponding parallel efficiency, on the Weather dataset.

processing algorithm computes its portion of the workload.

As noted in Section 1, relatively little work in the area of parallel data cube construction has been published. Given the significance and size of the underlying problem, there would appear to be a genuine need for this type of research. In our case, we have contributed to the literature by providing one of the most comprehensive frameworks for the computation of the data cube. Of particular significance is the fact that we have grounded our algorithmic work with an extensive implementation that demonstrates the viability of relational data cube systems as high performance OLAP platforms. We expect that the lessons learned during this process should also be of significant benefit to those constructing similar systems, whether in academia or industry.

Going forward, we note that the current system will serve as a computational backend for further OLAP related research. We have, for example, already constructed a robust multi-dimensional indexing subsystem and parallel query engine for the data cube [7]. We are also looking at new techniques for data cube compression and visualization, as well as seamless integration with existing database and OLAP products.

References

[1] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. *Proceedings of the 22nd International VLDB Conference*, pages 506–521, 1996.

[2] R. Becker, S. Schach, and Y. Perl. A shifting algorithm for min-max tree partitioning. *Journal of the ACM*, 29:58–67, 1982.

[3] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. *Proceedings of the 1999 ACM SIGMOD Conference*, pages 359–370, 1999.

[4] Y. Chen, F. Dehne, T. Eavis, and A. Rau-Chaplin. Parallel rolap data cube construction on shared-nothing multiprocessors. *Distributed and Parallel Databases*, 15:219–236, 2004.

[5] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1996.

[6] F. Dehne, T. Eavis, and A. Rau-Chaplin. Parallelizing the datacube. *Distributed and Parallel Databases*, 11(2):181–201, 2002.

[7] F. Dehne, Todd Eavis, and A. Rau-Chaplin. Distributed multi-dimensional ROLAP indexing for the data cube. *The 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, 2003.

[8] The Rising Storage Tide, 2003. http://www.datawarehousing.com/papers.

[9] T. Eavis. *Parallel Relational OLAP*. PhD thesis, Dalhousie University, 2003.

[10] Flex and Bison, 2003. `http://dinosaur.compilertools.net/`.

[11] W. Feller. *An Introduction to Probability Theory and its Applications*. John Wiley and Sons, 1957.

[12] P Flajolet and G. Martin. Probabilistic counting algorithms for database applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[13] S. Goil and A. Choudhary. High performance OLAP and data mining on parallel computers. *Journal of Data Mining and Knowledge Discovery*, (4), 1997.

[14] S. Goil and A. Choudhary. High performance multidimensional analysis of large datasets. *Proceedings of the First ACM International Workshop on Data Warehousing and OLAP*, pages 34–39, 1998.

[15] S. Goil and A. Choudhary. A parallel scalable infrastructure for OLAP and data mining. *International Database Engineering and Application Symposium*, pages 178–186, 1999.

[16] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Proceeding of the 12th International Conference On Data Engineering*, pages 152–159, 1996.

[17] X. Huang H. Lu and Z. Li. Computing data cubes using massively parallel processors. *7th Parallel Computing Workshop (PCW '97)*, 1997.

[18] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.

[19] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes. *Proceedings of the 1996 ACM SIGMOD Conference*, pages 205–216, 1996.

[20] P. Hass, J. Naughton, S. Seshadri, and L. Stokes. Sampling based estimation of the number of distinct values of an attribute. *Proceedings of International VLDB Conference*, pages 311–322, 1995.

[21] H.Lu, J.X. Yu, L. Feng, and X. Li. Fully dynamic partitioning: Handling data skew in parallel data cube computation. *Distributed and Parallel Databases*, 13:181–2002, 2003.

[22] L.V.S. Lakshmanan, J. Pei, and J. Han. Quotient cube: How to summarize the semantics of a data cube. *Proceedings of the 28th VLDB Conference*, 2002.

[23] L.V.S. Lakshmanan, J. Pei, and Y. Zhao. Qc-trees: An efficient summary structure for semantic olap. *Proceedings of the 2003 ACM SIGMOD Conference*, pages 64–75, 2003.

[24] Leda, 2003. `http://www.mpi-sb.mpg.de/LEDA/`.

[25] The Message Passing Interface standard, 2003. `http://www-unix.mcs.anl.gov/mpi/`.

[26] S. Muto and M. Kitsuregawa. A dynamic load balancing strategy for parallel datacube computation. *ACM 2nd Annual Workshop on Data Warehousing and OLAP*, pages 67–72, 1999.

[27] R. Ng, A. Wagner, and Y. Yin. Iceberg-cube computation with PC clusters. *Proceedings of 2001 ACM SIGMOD Conference on Management of Data*, pages 25–36, 2001.

[28] The OLAP Report. http://www.olapreport.com.

[29] Programming POSIX threads. http://www.humanfactor.com/pthreads.

[30] K. Ross and D. Srivastava. Fast computation of sparse data cubes. *Proceedings of the 23rd VLDB Conference*, pages 116–125, 1997.

[31] N. Roussopoulos, Y. Kotidis, and M. Roussopolis. Cubetree: Organization of the bulk incremental updates on the data cube. *Proceedings of the 1997 ACM SIGMOD Conference*, pages 89–99, 1997.

[32] S. Sarawagi, R. Agrawal, and A.Gupta. On computing the data cube. Technical Report RJ10026, IBM Almaden Research Center, San Jose, California, 1996.

[33] Z. Shao, J. Han, and D. Xin. Mm-cubing: Computing iceberg cubes by factorizing the lattice space. *to appear in the Proceedings of the 16th International Conference on Scientific and Statitistical Database Management (SSDBM)*, 2004.

[34] A. Shukla, P. Deshpande, J. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. *Proceedings of the 22nd VLDB Conference*, pages 522–531, 1996.

[35] Y. Sismanis, A. Deligiannakis, N. Roussopolos, and Y. Kotidis. Dwarf: Shrinking the petacube. *Proceedings of the 2002 ACM SIGMOD Conference*, pages 464–475, 2002.

[36] W. Wang, J. Feng, H. Lu, and J.X. Yu. Condensed cube: An effective approach to reducing data cube size. *Proceedings of the International Conference on Data Engineering*, 2002.

[37] The Winter Report, 2003. `http://www.wintercorp.com/vldb/2003_TopTen_Survey`.

[38] D. Xin, J. Han, X. Li, and B. W. Wah. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. *in Proceedings Int. Conf. on Very Large Data Bases (VLDB'03)*, 2003.

[39] G. Yang, R. Jin, and G. Agrawal. Implementing data cube construction using a cluster middleware: Algorithms, implementation experience, and performance evaluation. *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*, 2002.

[40] Y. Zhao, P. Deshpande, and J. Naughton. An array-based algorithm for simultaneous multi-dimensional aggregates. *Proceedings of the 1997 ACM SIGMOD Conference*, pages 159–170, 1997.

[41] W. Zipf. *The Psycho-Biology of Language: An Introduction to Dynamic Philology.* Houghton Mifflin, 1935.