A Cluster Architecture for Parallel Data Warehousing

Frank Dehne
Carleton University
Ottawa, Canada
frank@dehne.net
http://www.dehne.net

Todd Eavis
Dalhousie University
Halifax, Canada
eavis@cs.dal.ca

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

Abstract

We describe the parallel, cluster-based implementation of an algorithm for the computation of a database operator known as the datacube. *Though a number of efficient sequential algorithms have recently been proposed for this problem, very little research effort has been expended upon cost-effective parallelization techniques. Our approach builds directly upon the existing sequential proposals and is designed to be both load balanced and communication efficient. We also provide experimental results that demonstrate the viability of our technique under a variety of test conditions. Ultimately, we show that parallel performance relative to the underlying sequential algorithm (speedup) is near optimal.*

1. Introduction

Over the past five years, we have seen tremendous growth in the data warehousing market. Despite the sophistication and maturity of conventional database technologies, however, the ever-increasing size of corporate databases, coupled with the emergence of the new global Internet "database", suggests that new computing models may soon be required to fully support many crucial data management tasks. In particular, the exploitation of parallel algorithms and architectures holds considerable promise, given their inherent capacity for both concurrent computation and data access.

In our current research, we focus on the datacube [1, 3, 8, 10, 13, 16, 18], a database operator that can be used to pre-compute multiple views of selected data by aggregating values across all possible attribute combinations (a *group-by* in database terminology). The resulting data structures can then be used to dramatically accelerate visualization and query tasks associated with large information sets.

In the sequential setting, a significant amount of datacube-related work has already been carried out. The primary focus of that research has been upon algorithms that reduce computuation by sharing sort costs [1, 16], that minimize external memory sorting by partitioning the data into memory-size segments [3, 13], and that represent the views themselves as multi-dimensional arrays [8, 18]. By contrast, relatively little research effort has been focused upon parallel computation. In [7], the authors propose an algorithm that computes each possible view across all processors. While that technique does provide better performance than the purely sequential alternatives, it can also produce excessive inter-node communication, particularly in high dimension spaces.

In this paper, we describe the implementation of an algorithm that is both load balanced and communication efficient. As described in [5], our approach is to distribute view subsets to individual nodes, where efficient sequential algorithms can be used to independently calculate their assigned workload. In terms of the architecture, we have chosen to target the emerging Beowulf cluster model for the initial implemetation. Though the theoretical work was designed with a shared memory architecture in mind, the cluster platform is extremely attractive in terms of both price and accessiblity. As the technology matures, the opportunities for such research should only increase.

We also present experimental results that demonstrate the viability of our approach. Our evaluation explores the impact of parameters such as the number of processors, the size of the input set, and the total number of views. In short, we demonstrate that our algorithm produces running times that are near optimal with respect to those of the underlying sequential approach.

The paper is organized as follows. Section 2 provides an overview of the data warehousing model, as well as the datacube operator. Section 3 describes the issues relevant to a parallel implementation. We discuss the details of that implementation in Section 4, with experimental results highlighted in Section 5. Section 6 identifies a number of new problem areas relating to the current datacube algorithms, as well as the more general field of data warehousing for

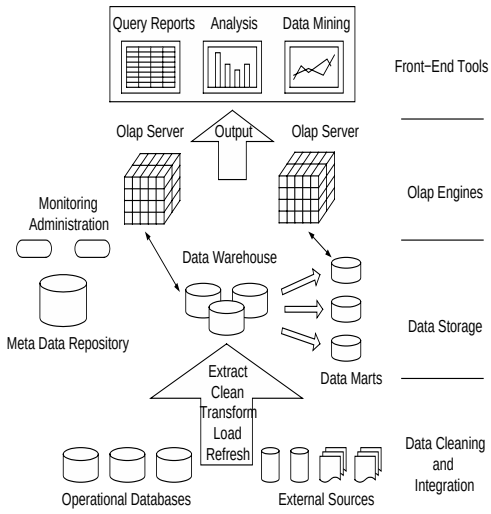

Figure 1. The basic OLAP model. Our current emphasis is on the third level — the OLAP Engine.

cluster architectures. Section 7 concludes the paper and offers a few final observations.

2 Background

Data warehouses can be described as *decision support systems* in that they allow users to assess the evolution of an organization in terms of a number of key data attributes or dimensions. Typically, these attributes are extracted from various operational sources (relational or otherwise), then cleaned and normalized before being loaded into a relational store. By exploiting multi-dimensional views of the underlying data warehouse, users can "drill down" or "roll up" on hierarchies, "slice and dice" particular attributes, or perform various statistical operations such as ranking and forecasting. This approach is referred to as Online Analytical Processing or OLAP. Figure 1 illustrates the basic model.

2.1 The Datacube

Because the views in a data warehouse are multi-dimensional in nature, it is often convenient to consider the data as being housed in an n-dimensional cube (see Figure 2).This datacube consists of a core or base *cuboid*, surrounded by a collection of sub-cubes/cuboids that represent the aggregation of the base cuboid along one or more dimensions [8]. (We refer to the dimension to be aggregated as the *measure* attribute, while the remaining dimensions are known as the *feature* attributes). In total, a d-dimensional data warehouse is associated with 2^d such cuboids. Though these cuboids can be constructed "on the fly", in practice they are normally pre-computed so as to improve the efficiency of user queries.

Since the datacube suggests a multi-dimensional interpretation of the data space, a number of OLAP vendors have chosen to physically model the cube as a multi-dimensional array. These MOLAP (multi-dimensional OLAP) products offer rapid response time on OLAP queries since it is possible to index directly into the datacube structure to retrieve subsets of aggregated data. Unfortunately, MOLAP solutions have not proven to scale effectively to large, high-dimensionality data sets. The problem is that as the number of dimensions grows, the data in the datacube becomes increasingly *sparse*. In other words, many of the attribute combinations represented by the datacube structure do not contain any aggregated data. As such, a fully materialized MOLAP array can contain an enormous number of empty cells, resulting in unacceptable storage requirements [13]. Though compression techniques can be used to alleviate this problem, doing so destroys the natural indexing that makes MOLAP so appealing. Consequently, MOLAP is often a more attractive option when used in conjunction with the relational data warehouse. IBM's DB2 OLAP server, for example, has been integrated with Hyperion's Essbase to provide MOLAP efficiency on a subset of the smaller cuboids, while the DB2 engine can be used to implement larger, more sparse cuboids within the relational model. Nevertheless, with data warehouses increasing not only in number, but in size and complexity as well, the robustness of the MOLAP approach seems doubtful.

In contrast, relational OLAP (ROLAP) seeks to exploit the maturity and power of the relational paradigm. Instead of a multi-dimensional array, the ROLAP datacube is implemented as a collection of up to 2^d relational tables, each representing a particular cuboid. Because the cuboids are now conventional database tables, they can be processed and queried with traditional RDBMS techniques (e.g., indexes and joins). Moreover, they are much more efficient on large data warehouses since only those datacube cells that actually contain data are housed within the tables. It is for these reasons that we have chosen to focus our efforts on the ROLAP model.

2.2 Building the Datacube

As mentioned, a fully materialized datacube consists of 2^d individual views. Though we can use conventional SQL to construct the tables, it is immensely inefficient to do so. In particular, we would require 2^d sorts of the original data set, a task that is especially expensive given that virtually

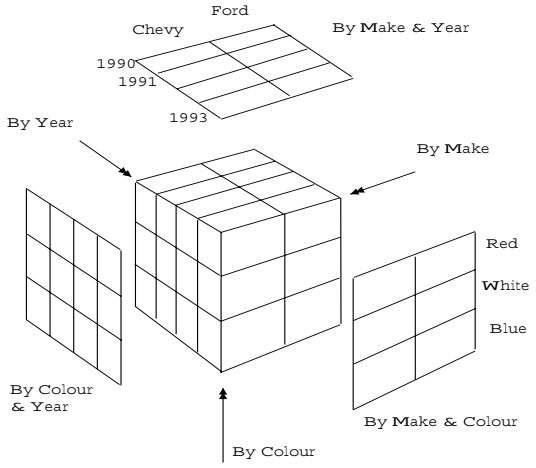

Figure 2. A typical datacube, in this case describing the views relevant to an automobile manufacturer.

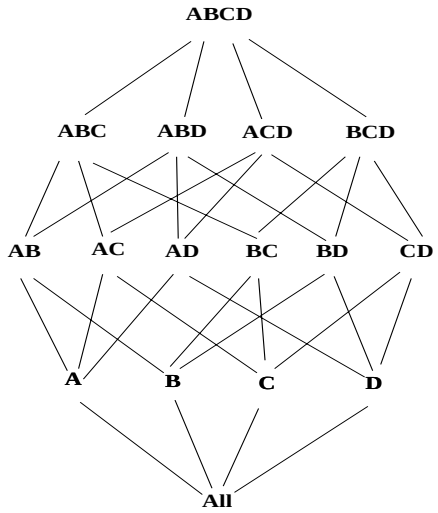

Figure 3. The datacube lattice consists of all possible attribute combinations. The "all" node represents the aggregation of all records.

all real-world input sets would be far larger than the size of main memory and would each necessitate an external memory sorting of considerable expense. As a result, a number of algorithms have been proposed that attempt to reduce the time required to build the cube by exploiting the inherent relationships that exist between various tables. For example, a three-dimensional cuboid can be viewed as the parent of three two-dimensional cuboids, each of which contains a distinct combination of two dimensions of the parent. Clearly, it should not be necessary to independently compute all four views since the parent and one or more of the children may be able to share some portion of the aggregation workload.

Typically, we represent the collection of cuboids as a lattice [10] of height $d + 1$. Starting with the base cuboid — containing the full complement of dimensions — the lattice branches out by connecting every parent node with the set of child nodes/views that can be derived from its dimension list. In general, a parent comtaining k dimensions can be connected to k views at the next level in the lattice (see Figure 3).

The algorithms themselves can generally be divided into *top-down* and *bottom-up* approaches. In the former case, we first compute the parent cuboids and then utilize these aggregated views to efficiently compute the children. Various techniques have been employed for this purpose, including those based upon sorting, hashing, and the manipulation of in-memory arrays [1, 16, 18]. In all cases, the goal is to generate coarse granularity tables from views that have previously been aggregated at a finer level of granularity.

In contrast, bottom-up computation seeks to first partition the data set on single attributes [3, 13]. Within each partition, we recursively aggregate at finer levels of granularity until we reach the point where no more aggregation is possible/necessary. It appears that bottom-up cube computation may be well suited to large, sparse datacubes. By starting with the smallest cuboids (i.e., the ones with the greatest degree of aggregation) and working thereafter with partition-sized chunks of the data set, it is possible to avoid a certain number of the large external memory sorts that are required in a top-down implementation. Despite this fact, it is quite difficult to rank the approaches for practical applications since very little comparative testing has been performed.

3 Our Approach to Parallelizing the Datacube

To date, little work has focused upon the application of parallel algorithms and data structures to speed up datacube construction. In our recent research, we have sought to develop load balanced and communication efficient parallel algorithms that, in turn, exploit the efficiency of the existing sequential approaches. More specifically, our techniques for building the datacube partition the original problem into a set of sub-cube computations which are then distributed

to individual processors. This is in contrast to the technique described in [7] that calculates each sub-cube across all processors. Not only do these mechanisms fail to directly utilize current sequential algorithms, but they can create excessive inter-node communication. Our algorithms require very little communication overhead and are applicable to high dimension spaces.

3.1 Implementation

We have proposed parallel algorithms for both the top-down and bottom-up paradigms. Our initial implementation is based upon the sequential Pipesort developed at IBM's Almaden Research Center [16]. This is a top-down technique that attempts to find a set of distinct sort paths within the lattice such that the cost of computing child views from their parents is minimized. Ultimately, a subset of the child views are computed using a linear scan of the parent view, while the remaining views require a re-sorting of a parent cuboid.

In parallel, our task is to find a partitioning of the lattice that balances the cost of sub-cube computation across the p processors. For this purpose we employ a k-min-max partitioning algorithm. Though the min-max procedure does not guarantee an optimal split across the network (this is an np-complete problem), it does guarantee a lower bound on the size of the largest subset. To further improve the accuracy of the algorithm, we incorporate a tunable over-sampling factor into the min-max phase. The over-sampling produces $k * p$ sub-lattices — where k is the over-sampling factor — and allows us to group the views into more evenly balanced subsets. Once these subsets have been established, they can be distributed to the local nodes where the existing sequential algorithms are executed.

3.2 The Basic Algorithm

The algorithm itself consists of the following steps:

1. Construct a lattice housing all 2^d views.
2. Estimate the size of each of the views in the lattice.
3. To determine the cost of using a given view to directly compute its children, use its estimated size to calculate (a) the cost of scanning the view and (b) the cost of sorting it.
4. Using the bipartite matching technique presented in [16], reduce the lattice to a spanning tree that identifies the appropriate set of prefix-ordered sort paths.
5. Partition the tree into p sub-trees.
6. Distribute the sub-tree lists to each of the p compute nodes.
7. On each node, use the sequential Pipesort algorithm to build the set of local views.

4 A look at the details

In the following section, we provide a detailed description of the implementation. We first describe the code base and supporting libraries, followed by an overview of each of the steps listed in Section 3.2.

4.1 The Code Base

Though some initial coding was done in C, we chose to move to a C++ platform in order to more efficiently support the growth of the project. With the expansion of the code base and the involvement of a number of independent developers, several of whom were in geographically distinct locations, it made more sense to employ an object-oriented language that allowed for data protection and class inheritance. One notable exception to the OOP model, however, was that the more familiar C interface to MPI functions was used.

4.1.1 The LEDA Libraries

In practice, a meaningful implementation of parallel datacube algorithms tends to be quite labour intensive. Consequently, we chose to employ third-party software libraries so that we could focus our own efforts more completely upon the new research. After a review of existing packages, we selected the LEDA libraries because of the rich collection of fundamental data structures (including linked lists, hash tables, arrays, and graphs), the extensive implementation of supporting algorithms, and the C++ code base [12]. Though there is a slight learning curve associated with LEDA, the package has proven to be both efficient and reliable.

4.1.2 The OOP framework

Having incorporated the LEDA libraries into our C++ code base, 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 sub-classing the lattice template and (a) overriding or adding methods and (b) defining the new node and edge objects that should be used as template parameters.

In the current implementation, the base lattice has been sub-classed so as to augment the graph for the sort-based

optimization. For each view/node, we estimate its construction cost in two formats: as a linear scan of its parent and as a complete resorting of its parent. Since these cost assessments depend upon accurate estimates of the sizes of the views themselves, the inclusion of a view estimator is required.

4.2 A probabilistic view estimator

A number of inexpensive algorithms have been proposed for view estimation [17]. The simplest merely entails using the product of the cardinalities of each dimension to place an upper bound on the size of each cuboid. A slightly more sophisticated technique computes a partial datacube on a ramdomly selected sample of the input set. The result is then "scaled up" to the appropriate size. Though both approaches can give reasonable results on small, uniformly distributed datasets, they are not as reliable on real world data warehouses.

Consequently, the use of probabilistic estimators that rely upon a single pass of the dataset have been suggested. As described in [17], our implementation builds upon the counting algorithm of Flajolet and Martin [6]. Essentially, we concatenate the d dimension fields into bit-vectors of length L and then hash the vectors into the range $0 \ldots 2^L - 1$. The algorithm then uses a probablistic technique to count the number of distinct records (or hash values) that are likely to exist in the input set. To improve estimation accuracy, we employ a universal hashing function [4] to compute k hash functions, that in turn allows us to average the estimates across k counting vectors.

The probabilistic estimator was quite accurate, producing estimation error in the range of 5–6 % with 256 hash functions. However, its running time on large problems was disappointing. In short, the problem is that, despite an asymptotic bound of $O(n * 2^d)$, the constants hidden inside the inner computing loops are enormous (i.e, greater than one million!). For the small problems decribed in previous papers, this is not an issue. In high dimension space, it is intractable; the running time of the estimator extends into weeks or even months.

Considerable effort was expended in trying to optimize the algorithm. All of the more expensive LEDA structures (strings, arrays, lists, etc.) were replaced with efficient C-style data types. 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 Flajolet and Martin estimator is viable in high dimension space.

4.3 A simple view estimator

We needed a fast estimator that could be employed even in high dimension environments. We chose to use the technique that bounds view size as the product of dimension cardinalties. We also improve upon the basic estimate by exploiting the fact that a child view can be no bigger than the smallest of its potential parents. For size = S, we therefore have $S_{child} = min(Product_{child}, min(S_k for k \epsilon ParentSet))$. However, additional optimizations that incorporate intermediate results are not possible since a parallel implementation prevents us from sequentially passing estimates up and down the spanning tree. Section 5 discusses the results obtained using this version of the view estimator.

4.4 Computing the Edge Costs

As previously noted, the values produced by the estimator only represent the sizes of each output view, not the final edge costs that are actually placed into the lattice. Instead, the algorithm uses the view estimate to calculate the potential cost of scanning and sorting any given cuboid. An appropriate metric must be experimentally developed for every architecture upon which the datacube algorithm is run. For example, on our own cluster, an in-memory multi-dimensional sort is represented as $(d + 2)/3) * (n \log n)$, where d is the current level in the lattice. At present, we are working on a module that will be used to automate this process so that appropriate parameters can be provided without manually testing every architecture.

4.5 Constructing the spanning tree

Once the lattice has been augmented with the appropriate costs, we apply a weighted bipartite matching algorithm that finds an appropriate set of sort paths within the lattice (as per [16]). Working bottom-up, matching is performed on 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 matching algorithm itself is provided by LEDA and requires only minor modification for inclusion in our design (e.g., it must be converted from a maximum weighted matching to a minimum weighted matching by subtracting each edge cost from the maximum cost at that level in the tree).

4.6 Min-Max Partitioning

As soon as the bipartite matching algorithm has been executed, we partition the lattice into a set of k sub-trees using the min-max algorithm proposed by Becker, Schach and Perl [2]. The original algorithm is modified slightly since

it was actually designed to work on a graph whose costs were housed in the nodes, rather than the edges. As well, a *false root* with zero cost must be added since the algorithm iterates until the root partition is no longer the smallest sub-tree. The min-max algorithm performs quite nicely in practice and, in conjunction with the over-sampling factor mentioned earlier, produces a well balanced collection of sub-trees (see [5] for a more complete analysis).

Once min-max terminates, the k sub-trees are collected into p sets by iteratively combining the largest and smallest trees (with respect to construction cost). Next, each sub-tree is partitioned into a set of distinct prefix-ordered sort paths, then packaged and distributed to the individual network nodes. The local processor decompresses the package into its composite sort paths and performs a pipesort on each pipeline in its assigned workload. No further communication with the root node is required from this point onward.

4.7 Local Pipesorts

In short, a pipesort consists of two phases. In the first round, the root node in the list is sorted in a given multi-dimensional order. In phase two, we perform a linear pass through the sorted set, aggregating the most detailed records into new records that correspond to the granularity level of each cuboid in the sort path. As the new records are produced they are written directly to disk. For example, if we sort the data in the order *ABCD*, we will subsequently create the *ABCD*, *ABC*, *AB*, and *A* views as we traverse the sorted set.

Though we originally exploited LEDA's array sorting mechanism to sort the root node in memory, we have since re-written the sort using the C library routines so as to maximize performance. At present, all input sorting is performed in main memory. In the future, we expect to incorporate robust external memory sorting algorithms into the project.

5 Experimental Evaluation

In this section, we discuss the performance of our parallel pipesort implementation. We first provide an overview of the computing platform, as well as a description of the datasets that were generated for this phase of testing. We then analyze the algorithm in terms of a number of key parameters.

5.1 The Platform

The current prototype has been implemented on an inexpensive cluster architecture. Commonly referred to as a Beowulf-class cluster, the network consists of 9 Pentium processors — a root and eight compute nodes — connected by a full wire-speed Fast Ethernet switch. Each node runs its own copy of the Linux operating system and all resources are locally managed. In terms of inter-node communication, we adopt the message passing model and employ LAM's version of MPI (TCP/IP). Due to networking problems with earlier versions of the Linux kernel — that impacted upon the efficiency of MPI — the installation relies upon the 2.2.14 kernel to ensure optimal performance.

5.2 Data Generation

In order to effectively test the algorithms, it is necessary to utilize a wide variety of test sets that reflect the patterns and idiosyncracies one is likely to encounter in practical settings. As such, we have designed our own simple data generator that produces integer-based records (non-integer records would typically be mapped to integers in an industrial application). In addition to row and dimension counts, the generator accepts parameters that define the number of unique values in each dimension, as well as the proportion of total records that should contain a particular value (i.e., the skew). For example, it is possible to specify a dimension that contains 10 unique values, with 85% of all records holding just one of those possibilities. We are in the process of modifying the generation algorithm to utilize the *zipf* power-law function. Here, we express the probability of encountering a particular value i in a given dimension as $P_i \sim 1/i^a$, with the zipf factor a close to unity. As a is incremented from one, the data set becomes more skewed. The next version of our datacube application will include this feature.

5.3 Analysis of Results

We have been evaluating the algorithm on our cluster under a variety of test conditions. Running time, input size, dimension count and over-sampling factor have all been assessed by varying the relevant parameter and holding all others constant. As such, four charts have been included in this paper to illustrate some of the preliminary analysis. Figure 4 shows the running time of our parallel datacube construction method as we increase the number of processors (note: in this test, record count = 100000, dimension count = 6, sampling factor = 2, and distribution = uniform). Here, curves for average running time and *max* running time (i.e., the slowest single node) almost coincide with the optimal curve — $T_{opt} = T_{sequential}/p$. This is very promising. We note that the error in the *max* curve does increase with the number of processors. However, we can associate much of this error with the shortcomings of the estimator used during this round of testing. As the number of processors increases, the local workload becomes progressively smaller and, as such, estimation error becomes more pronounced. Again, we plan to explore the estimator problem

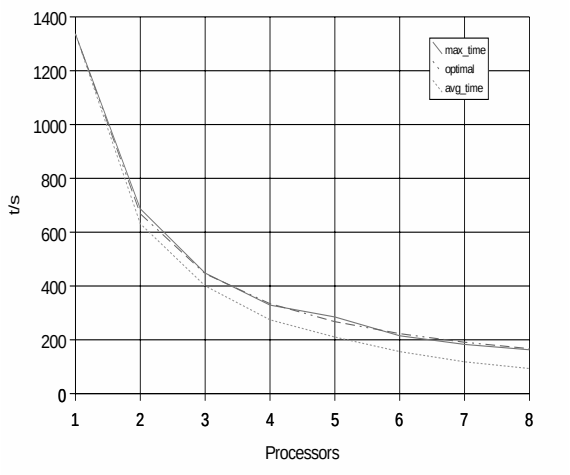

Figure 4. Processor Test

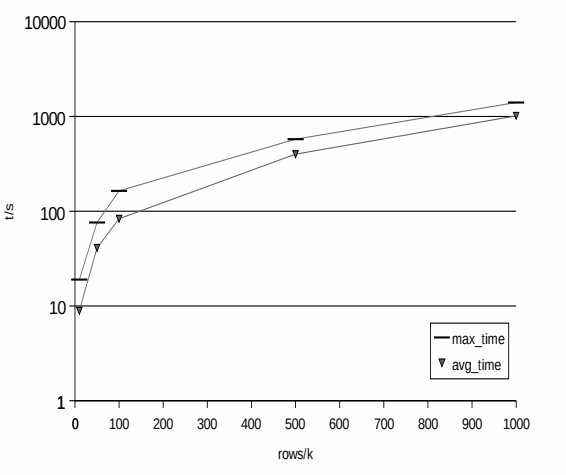

Figure 5. Data Test

in the coming months.

Figure 5 depicts the running time — using a logarithmic scale — as the size of the data set grows from 10,000 records to one million records (note: processor count = 8, dimension count = 6, sampling factor = 2, and distribution = uniform). The running time increased in an almost linear fashion. This is very encouraging for future tests that will evaluate performance on very large input sets.

Figure 6 illustrates the effect of increasing the number of dimensions in the problem space. We see that the algorithm's running time roughly doubles as the dimension count is incremented by one. In fact, this is exactly as one would expect given that adding a dimension doubles the number of views that must be computed.

Finally, in Figure 7, we look at the inpact of adjusting the over-sampling factor. The use of a sampling factor of two or more clearly inproves upon the balancing, and hence the running time, of the algorithm. However, what is perhaps most interesting is the fact that, despite higher average running times, sampling factors of either three or four actually had a slightly lower *max* time. It is therefore possible that the improved partitioning may be worth the cost of an extra sort on each node.

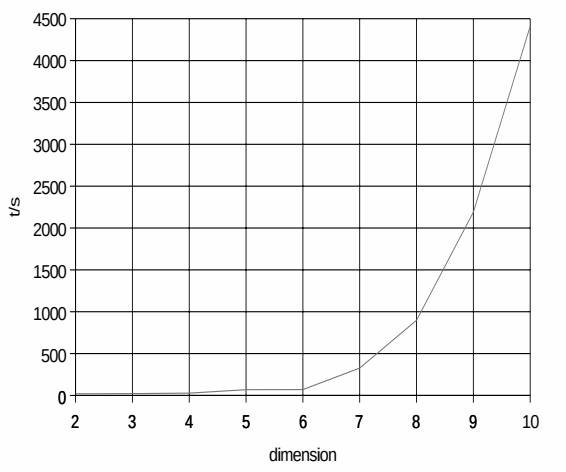

Figure 6. Dimension Test

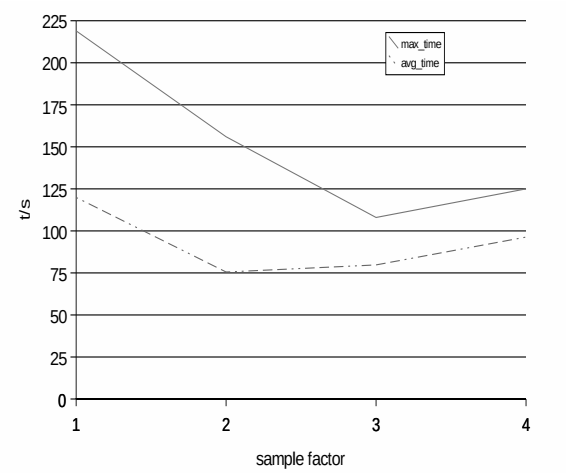

Figure 7. Sampling-factor Test

6 Future Work

A number of datacube related problems remain to be addressed. They include the following:

- In practice, data warehouse designers may wish to generate some subset of all available views. Cost effective algorithms for doing this must do more than simply calculate a partial cube on the selected cuboids; cost metrics must include intermediate views that would allow the selected cuboids to be calculated efficently.

Though a technique based upon a Steiner tree approximation was proposed in [16], this mechanism is likely to be too expensive in high dimensions.

- Given the potential advantages of the bottom-up approaches to datacube construction, it will be interesting to compare our current parallel Pipesort with the the bottom-up alternative. With the modular approach that we have taken on the implemenation project, this comparative analysis should be possible in the near future.

- Once the cuboids are generated, there is a need to address the problem of querying the data in a parallel environment. Should query performance be optimized by balancing many requests in a pipelined fashion, or should we try to ensure that individual queries are distributed evenly across all nodes?

- Appropriate indexing mechanisms are required since simple linear scans of the cuboids, or even single-dimension b-tree accesses, are far too time-comsuming for OLAP processing. One promising area of research with respect to high dimension OLAP concerns the use of packed r-trees [14, 11]. Long used as an indexing method for extended spatial objects [9], r-trees are attractive in the current context because they may provide a means of efficiently indexing the aggregation points in a multi-dimensional cuboid. Moreover, techniques for efficiently updating the packed trees in bulk increments suggest that packed r-trees may be used to improve the refresh rate of existing data warehouses [15].

7 Conclusions

As data warehouses continue to grow in both size and complexity, so too will the opportunities for researchers and algorithm designers who can provide powerful, cost-effective OLAP solutions. In this paper we have discussed the implementation of a parallel algorithm for the construction of a multi-dimensional data model known as the datacube. By exploiting the strengths of existing sequential algorithms, we can pre-compute all cuboids in a load balanced and communication efficient manner. Our experimental results have demonstrated that the technique is viable, even when implemented in a *shared nothing* cluster environment. In addition, we have suggested a number of opportunities for future work, including a parallel query model that utilizes packed r-trees. More significantly perhaps, given the relatively paucity of research currently being performed in the area of parallel OLAP, we believe that the ideas we have proposed represent just a fraction of the work that might lead to improved data warehousing solutions.

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

[5] F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin. Parallelizing the datacube. *International Conference on Database Theory*, 2001.

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

[7] S. Goil and A. Choudhary. High performance olap and data mining on parallel computers. *Journal of Data Mining and Knowledge Discovery*, 1(4), 1997.

[8] 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.

[9] A. Guttman. R-trees: A dynamic index structure for spatial searching. *Proceedings of the 1984 ACM SIGMOD Conference*, pages 47–57, 1984.

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

[11] I. Kamel and C. Faloutsos. On packing r-trees. *Proceedings of the Second International Conference on Information and Knowledge Management*, pages 490–499, 1993.

[12] Leda. http://www.mpi-sb.mpg.de/LEDA/.

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

[14] N. Roussopolis and D. Leifker. Direct spatial search on pictorial databases using packed r-trees. *Proceedings of the 1985 ACM SIGMOD Conference*, pages 17–31, 1985.

[15] 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.

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

[17] 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.

[18] 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.