Parallel Multi-Dimensional ROLAP Indexing *

Frank Dehne
School of Computer Science
Carleton University
Ottawa, Canada
frank@dehne.net

Todd Eavis
Faculty of Computer Science
Dalhousie University
Halifax, Canada
eavis@cs.dal.ca

Andrew Rau-Chaplin
Faculty of Computer Science
Dalhousie University
Halifax, Canada
arc@cs.dal.ca

Abstract

This paper addresses the query performance issue for Relational OLAP (ROLAP) datacubes. We present a distributed multi-dimensional ROLAP indexing scheme which is practical to implement, requires only a small communication volume, and is fully adapted to distributed disks. Our solution is efficient for spatial searches in high dimensions and scalable in terms of data sizes, dimensions, and number of processors. Our method is also incrementally maintainable. Using "surrogate" group-bys, it allows for the efficient processing of arbitrary OLAP queries on partial cubes, where not all of the group-bys have been materialized.

Our experiments show that the ROLAP advantage of better scalability, in comparison to MOLAP, can be maintained while providing, at the same time, a fast and flexible index for OLAP queries.

Keywords: Management of large scale distributed data, OLAP, Datacube, Parallel ROLAP Indexing, Cluster and Grid Applications.

1 Introduction

Online Analytical Processing (OLAP) has become a fundamental component of contemporary decision support systems. In 1995, Gray et al. [8] introduced the *datacube*, a relational operator/model used to compute summary views of data that can, in turn, significantly enhance the response time of core OLAP operations such as *roll-up*, *drill down*, and *slice and dice*. Typically constructed on top of relational data warehouses, these summary views (called *group-bys*) are formed by aggregating values across attribute combinations. For a d-dimensional input set R, there are 2^d possible group-bys. Figure 1 illustrates a datacube as well as a *lattice* which is often used to represent the inherent relationships between group-bys [10].

There are two standard datacube representations: MOLAP (multi-dimensional array) and ROLAP (set of relational tables). The array-based model, MOLAP (Multi-dimensional OLAP), has the advantage that native arrays provide an immediate form of indexing for cube queries. Experience has shown, however, that MOLAP has scalability problems [16]. For example, high-dimension datacubes represent extremely sparse spaces that are not easily adapted to the MOLAP paradigm. Hybrid indexing schemes are often used, significantly diminishing the power of the model. Moreover, since MOLAP needs to be integrated with standard relational databases, *middleware* of some form must be employed to handle the conversion between relational and array-based data representations.

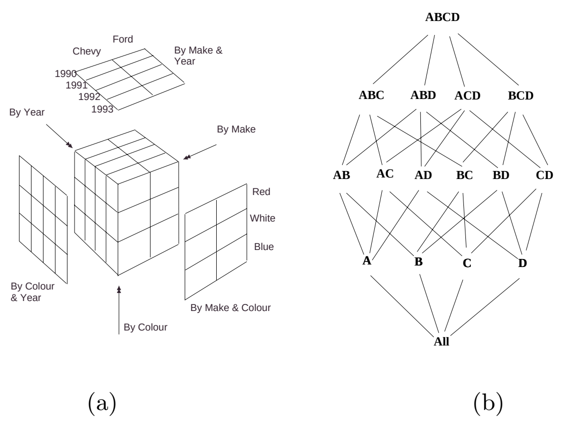

Figure 1: (a) A three dimensional datacube for automobile sales data. (b) The lattice corresponding to a four dimensional data cube with dimensions A, B, C and D.

The relational model, ROLAP (Relational OLAP), does not suffer from such restrictions. Its summary records are stored directly in standard relational tables without any need for data conversion. Its table based data representation does not pose scalability problems. Yet, many current commercial systems use the MOLAP approach [16]. The main reason, as outlined in [16], is the indexing problem for the fast execution of OLAP queries. The problem

*Research partially supported by the Natural Sciences and Engineering Research Council of Canada.

for ROLAP is that it does not provide an immediate and fast index for OLAP queries. Many vendors have chosen to sacrifice scalability for performance.

This paper addresses the query performance issue for ROLAP and proposes a novel, distributed multi-dimensional ROLAP indexing scheme. We show that the ROLAP advantage of high scalability can be maintained, while at the same time providing a fast index for OLAP queries. We propose a distributed indexing scheme which is a combination of packed R-trees with distributed disk striping and Hilbert curve based data ordering. Our method requires only very low communication volume between processors and works in "low bandwidth connectivity" multi-processor environments such as Beowulf type processor clusters or workstation farms. Our method does not require a shared disk and scales well with respect to the number of processors used.

To further improve the scalability of ROLAP with respect to the size and dimension of the data set (which was already better than MOLAP's scalability), we extend our indexing scheme to the partial cube case. The large number of group-bys, 2^d, is a significant problem in practice for any datacube method. We consider the case where we do not wish to build (materialize) all group-bys, but only a subset. For example, a user might want to only materialize those group-bys that are most frequently used, thereby saving disk space and time for the cube construction. The problem then is to find a way to answer effectively those less frequent OLAP queries which require group-bys that have not been materialized. We present an indexing scheme, based on "surrogate group-bys", which answers such queries efficiently. In fact, our experiments show that our distributed query engine is almost as efficient on "virtual" group-bys as it is on ones that actually exist.

In summary, our method provides a framework for distributed high performance indexing of ROLAP cubes with the following properties:

- practical to implement,
- low communication volume,
- fully adapted to external memory (i.e. disks),
- no shared disk required,
- incrementally maintainable,
- efficient for spatial searches in high dimensions,
- scalable in terms of data sizes, dimensions, and number of processors.

We have implemented our distributed multi-dimensional ROLAP indexing scheme in C++, STL and MPI, and tested it on a 17 node Beowulf cluster (a frontend and 16 compute nodes). While easily extendible to *shared everything* multi-processors, our algorithms perform well on these low-cost commodity-based systems. Our experiments show that for RCUBE index construction and updating, close to optimal speedup is achieved. An RCUBE index for a fully materialized data cube of ≈640 million rows (17 Gigabytes) on a 16 processor cluster can be generated in just under 1 minute. Our method for distributed query resolution also exhibits good speedup achieving, for example, a speedup of 13.28 on 16 processors. For distributed query resolution in partial datacubes, our experiments show that searches against absent (i.e. non-materialized) group-bys can typically be resolved at only a small additional cost. Our results demonstrate that it is possible to build a ROLAP datacube that is scalable and tightly integrated with the standard relational database approach and, at the same time, provide an efficient index for OLAP queries.

The remainder of this paper is organized as follows. In Section 2, we review some of the key research results from the sequential and parallel settings and describe our framework for distributed index generation, including mechanisms for building and updating the indexes. Section 3 presents the distributed query engine that is used to access the indexed group-bys. A performance analysis of our current prototype is presented in Section 4. Section 5 concludes the paper.

2 Distributed Index Construction For ROLAP

Various methods have been proposed for building ROLAP datacubes [1, 2, 4, 5, 3, 7, 8, 10, 17, 20] but there are only very few results available for the indexing of such cubes. For sequential query processing, Gupta et al. [9] propose an indexing model composed of a collection of b-trees. While adequate for low-dimensional datacubes, b-trees are inappropriate for higher dimensions in that (a) their performance deteriorates rapidly with increased dimensionality and (b) multiple, redundant attribute orderings are required to support arbitrary user queries. In [19] Roussopoulos et al. propose the *cubetree*, an indexing model based upon the concept of a *packed* R-tree [18]. For parallel query processing, a typical approach used by current commercial systems like ORACLE 9i RAC [15] is to improve throughput by distributing a stream of incoming queries over multiple processors and having each processor answer a subset of queries. However, such an approach provides no speedup for each individual query. For OLAP queries, which can be time consuming, the parallelization of each query is important for the scalability of the entire OLAP system. With respect to the parallelization of R-tree queries, a number of researchers have presented solutions for general purpose environments. In [12], Koudas, Faloutsos and Kamel present a *Master R-tree* model that employs a centralized index and a collection of distributed data files. Schnitzer and Leutenegger's Master-Client R-tree [21] improves upon the earlier model by partitioning the central index into a

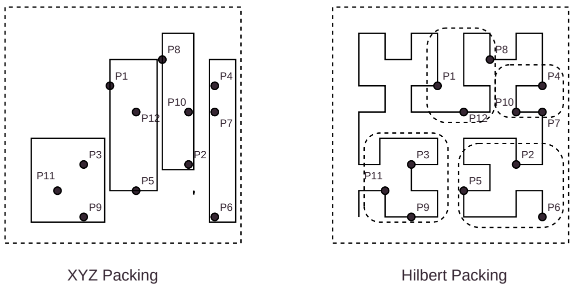

Figure 2: Hilbert curve packing versus XYZ.

smaller master index and a set of associated client indexes. While offering significant performance advantages in generic indexing environments, neither approach is well-suited for OLAP systems. In addition to the sequential bottleneck on the main server node, both utilize partitioning schemes that can lead to the localization of searches. In addition, neither approach provides a mechanism for incremental updates. In the remainder of this section, we present the distributed RCUBE indexing method, which has no sequential bottleneck, provides load balancing across the p processors during the resolution of each query (i.e. good parallelization), and allows for incremental updates.

2.1 Generating the Distributed RCUBE Index

The distributed RCUBE consists of a distributed datacube and a distributed RCUBE index which is used to answer multi-dimensional range queries on individual group-bys. The challenge is in how data ordering and partitioning can be used to help satisfy the following goals: 1) partition the data such that the number of records retrieved per node is as balanced as possible, thereby maximizing the simultaneous involvement of *all* processors for each query resolution, and 2) minimize the number of disk seeks required in order to retrieve the records returned by a query.

In the distributed RCUBE, as with the Master-Client technique, local partial R-tree indexes are constructed on each processor and used to resolve a portion of the query. However, for our distributed RCUBE, there is no global R-tree on the front-end. Instead, queries are passed directly to each processor in the cluster, via a single short message, and intermediate results remain distributed and available for further processing. For OLAP query results that are to be further processed, this also avoids the possible bottleneck of previous solutions, where the results were always gathered on the front-end. Another difference to previous methods is that the distributed RCUBE index results in the generation of local packed R-tree *forests* rather than a single R-tree.

A further important difference between our distributed RCUBE index and the previous work in [9, 19, 21] is that our distributed RCUBE index method applies a novel combination of Hilbert-curve sort ordering and round robin disk striping for data partitioning. Previous approaches used *XYZ* (sometimes also called *lowX* or *nearest-X*) data ordering, which is simply the standard multi-dimensional sort ordering. The disadvantage of that approach is that response time deteriorates rapidly when non-primary indices are required, since relevant points are dispersed broadly across the entire data set. Our approach applies a combination of Hilbert-curve sort ordering and round robin disk striping. Hilbert-curve orderings have been shown to be an effective tool for ordering data such that items that are close to each other in the original space are likely to be placed close to each other in the sorted order [11, 6]. Experimental evidence indicates a significant performance advantage over the XYZ ordering on sequential range queries [11]. Figure 2 illustrates a typical case. While XYZ is likely to be efficient for range queries with a large X component and a small Y component, queries with large Y components are likely to require an excessive number of disk accesses. In higher dimensions, the problem is exacerbated. Hilbert-based ordering, on the other hand, favors no single dimension and is therefore very well suited to arbitrary range queries. In the parallel environment, considered here for distributed RCUBE index constrution, we have, however, the additional requirement that we seek to balance the retrieval times for arbitrary range queries across all p processors. Therefore, an effective data partitioning mechanism is essential. Our approach is to stripe the Hilbert-curve ordered data in a round robin fashion such that successive records are sent to successive processors. We then build local packed R-trees from the striped data. The motivation for this striping pattern is that it dramatically increases the likelihood that the space bounded by the hyper-rectangle of an arbitrary user query will be evenly distributed across the p processors. Figure 3 illustrates this argument. The diagram shows the effect of striping the original space across two processors. The user query (shown as a dashed rectangle) results in the retrieval of eight points, with each processor contributing four points from a pair of contiguous blocks. It

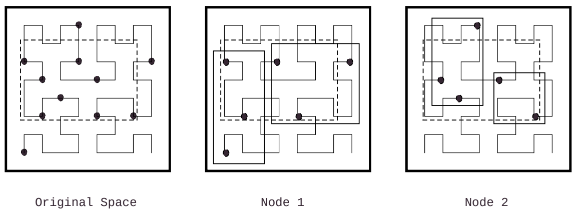

Figure 3: Striping the data across two processors. (Block capacity = 3)

is also worth noting that this example would require four accesses with a sequential R-tree implementation.

Algorithm 1 Outline of *Distributed RCUBE* Construction

```
Input: Raw data set $R$.
Output: A distributed data cube, $C$, distributed RCUBE index, $I$.
1: Using the parallel ROLAP data cube generation algorithms from [3] or [4, 5] generate the distributed data cube, $C$.
2: Using parallel sample sort [13], order each group-by $v$ of $C$ in Hilbert order and stripe the result across the processors in a round-robin fashion such that each of the $p$ processors receives a stripe of size $\lceil \frac{n}{p} \rceil$, where $n$ is the number of records in $v$.
3: Each processor $P_i$, independently and in parallel, performs the following for each local data stripe for a group-by $v$: For a disk block size of $m$ records, and a local record count $k$ for the group-by $v$, associate a bounding box with each of the $\lceil \frac{k}{m} \rceil$ blocks in the stripe. Using these blocks as the base (for the leaves), build the packed R-tree in the usual bottom-up fashion. Write the disk blocks representing the R-tree to disk in level ordering, starting with the block representing the root.
```

Algorithm 1 presents an outline of our distributed RCUBE index generation method. Much of the communication complexity of the algorithm is associated with Step 2 which we will now discuss in more detail. In Step 1, the distributed data cube was generated using the parallel ROLAP data cube generation algorithms from [3] or [4, 5]. Note that, in [4, 5] every group-by generated is entirely stored on one single processor, whereas in [3] every group-by is distributed evenly across the p processors. This implies different sort criteria for these two cases. The computation of the comparison function for the global sort ordering is a non trivial combination of the Hilbert curve comparison function (in our implementation, we use code from [14]) and a comparison function representing round robin disk striping. Furthermore, we do not wish to execute a separate sort for each group-by, which could result in up to 2^d sort operations. Instead, we combine the comparison functions for all group-bys into one single global sort operation. As a result, we can implement Step 2 with only two h-relation (MPI_AllToAllv) operations.

2.2 Updating the Distributed RCUBE

An important advantage of our distributed RCUBE generation method is that it is easy to perform efficient cube updates. In typical data warehousing applications, updates consist of an accumulated additional data set R' that needs to be added to the original data set R. Such updates typically occur on a daily or monthly schedule.

In order to add R' to the data cube, our method constructs the data cube C' for R', sorts each group-by of C' in Hilbert-curve ordering and stripes it across the disk in round-robin fashion. Each processor performs, for each group-by v and received update v' of C' relevant for v, the following two operations: (1) it merges v' into v and agglomerates, and (2) it merges the two packed R-trees for v and v'.

3 Distributed ROLAP Query Engine

Previous R-tree parallelization results have focused exclusively on the retrieval characteristics of R-trees [9, 19]. However, in an OLAP environment, accessing disk blocks is only the first phase of query resolution. Typically, some form of post-processing is then required to fully resolve the original query. An important example of this is partial cube extrapolation. The construction of a partial cube implies that some number of group-bys do not physically exist on disk. There needs to be an efficient mechanism for performing searches in these non-materialized group-bys.

In this section, we describe the implementation of a distributed datacube query engine. A general framework for post-processing is presented, along with a specific algorithm for handling partial cube indexing.

3.1 Distributed RCUBE Query Resolution

As discussed, our distributed RCUBE index has been designed to balance the retrieval of query records across all p processors. Once the records have been obtained, additional OLAP processing is often necessary. The fundamental model, outlined in Algorithm 2, provides the means by which both forms of computation may be carried out in an efficient, load balanced manner.

In Step 1, the query is distributed to all of the p processors, avoiding unnecessary bottlenecks on the frontend. The query usually cannot be executed in its *native* form, however, since the user's request is not likely to match the physical ordering of attributes that was determined by the original datacube build algorithm. For example, the user may request a three-dimensional group-by sorted and presented as A × B × C, while Algorithm 1 may have generated that group-by as C × A × B. In Steps 2 and 3, we identify the group-bys whose dimensions represent a valid permutation of the dimensions of the user request and then transform the original query

Algorithm 2 Outline of *Distributed RCUBE* Query Resolution

```
Input: A set $S$ of indexed group-bys, striped evenly across $p$ processors $P_1, \ldots P_p$, and a query $Q$.
Output: Query result deposited on front-end or distributed across the $p$ processors.
1: Pass query $Q$ to each of the $p$ processors.
2: Locate target group-by $T$.
3: Transform $Q$ into $Q\prime$ according to the attribute ordering of the records in $T$.
4: In parallel, each processor $P_j$ retrieves the record set $R_j$ matching $Q\prime$ for its local data and then reorders the values of each record of $R_j$ to match the attribute ordering of $Q$.
5: Perform a parallel sample sort [13] of $R_1 \cup R_2 \cup \ldots \cup R_p$ with respect to the attribute ordering of $Q$.
6: IF the query result is to be deposited on the front-end THEN collect the result via a MPI_AllGather.
```

to match the attribute order of the index/group-by. This transformed query is passed to the packed R-tree. Since the *retrieved* records are not guaranteed to have the right attribute ordering or the right ordering of records, further processing is necessary. In Step 4, the attributes of each record are permuted, if necessary, via a single linear scan of the query result. In Step 5, the query result is sorted. If the query result is to be deposited on the front-end, it is simply collected via a MPI_AllGather operation. Otherwise, the result remains distributed over the p processors for further parallel processing.

A number of additional performance improvements are included in our solution. Our packed R-tree implementation performs a *prefetch* on all parent pages in the group-by index. Because the pages of level i in the packed R-tree are written contiguously to disk prior to the pages in level $i-1$ (Step 3 of Algorithm 1), the prefetch of all relevant parent pages allows the query engine to minimize the seek time associated with traversing the index.

We also employ a *threshold factor* α to determine whether or not a full parallel sort is required. For very small result sets, a p processor sort would introduce unnecessary communication overhead. If the number of records in the result set is below α, then the partial result sets are sent directly to a single processor for sorting. The threshold factor can be tuned to the physical characteristics of the parallel machine.

3.2 Distributed *Partial* RCUBE Query Resolution

To further improve the scalability of ROLAP with respect to the size and dimension of the data set, we now consider the case where we do not wish to build all group-bys but only a subset. Since the computation of all 2^d group-bys can lead to unacceptable processing and storage requirements, particularly in higher dimensions, a user might want to only build those group-bys that are most frequently used, thereby saving disk space and time for the cube construction. The problem for OLAP query resolution is then to find a way to answer effectively those less frequent OLAP queries which require group-bys that have not been materialized.

Algorithm 3 Outline of Distributed *Partial* RCUBE Query Resolution

```
Input: A partial set $S'$ of indexed group-bys, striped evenly across $p$ processors $P_1, \ldots P_p$, and a query $Q$.
Output: Query result deposited on front-end or distributed across the $p$ processors.
1: Pass query $Q$ to each of the $p$ processors.
2: Locate a surrogate group-by $T$ containing the attributes in $Q$ and possibly some additional, peripheral, attributes. Among all possible such group-bys select as surrogate group-by $T$ the one with smallest size.
3: Transform $Q$ into $Q\prime$ according to the attribute ordering of the records in $T$ and add "*" values for the peripheral attributes.
4: In parallel, each processor $P_j$ retrieves the record set $R_j$ matching $Q\prime$ for its local data and then reorders the values of each record of $R_j$ to match the attribute ordering of $Q$. While performing the reordering, processor $P_j$ removes from each record the redundant values for the peripheral attributes of $T$.
5: Perform a parallel sample sort [13] of $R_1 \cup R_2 \cup \ldots \cup R_p$ with respect to the attribute ordering of $Q$. While performing the sort, aggregate duplicate records that have been introduced by the peripheral attributes of the surrogate group-by $T$.
6: IF the query result is to be deposited on the front-end THEN collect the result via a MPI_AllGather.
```

It is important to observe that datacube construction costs are skewed heavily towards the upper (high dimensional) portion of the lattice. For example, in a ten dimensional datacube, much of the weight is typically associated with group-bys of five to ten dimensions. In the upper portion of the lattice, little aggregation takes place and the group-bys are very similar to one another. For example, we measured the sizes of group-bys of a data cube for a 10 dimensional data set of 1 Million records. Most group-bys with 6 through 10 dimensions contain almost 97% of all records in the original input set. Therefore, it is not efficient to build all these very similar group-bys. Clearly, a *partial cube* construction and indexing method is required. However, the query engine must then be able to efficiently answer queries on group-bys that do not physically exist. In the following, we present a new method, based on "surrogate group-bys", which answers such queries efficiently. An outline of our method is given in Algorithm 3.

There are a number of key difference between Algorithm 3 and the previous Algorithm 2. First, a *surrogate* group-by T is used as the basis of query resolution for Q. A surrogate is an alternate group-by that will be used to answer the query on the group-by requested by the user, termed the *primary* group-by. To select a surrogate, each processor scans its local disk to find those group-bys whose dimensions represent a superset of the dimensions specified by the user. From the group-bys in this list, it selects the group-by of minimum size. Note that, since this surrogate group-by contains even more detailed information than the original group-by, we can answer *all queries* associated with the original group-by. Furthermore, we note that because Hilbert-based R-tree packing has been used, there is no performance problem due to the different ordering of the records in the group-by, since the Hilbert curve does not favor any particular order. In [22], the authors observe that when XYZ ordering is used, the only alternate group-bys that can be efficiently used for this purpose are the ones in which the attributes of Q represent a prefix of T. Since this situation is unlikely to occur in practice, XYZ ordering makes partial cube query resolution very costly. However, as shown in the experiments in Section 4, such problems do not occur with Hilbert ordering.

Once the surrogate group-by T has been determined, the query is transformed by (i) re-arranging the attributes of the query to match the order of the surrogate and (ii) adding "*" values for the *peripheral* attributes of the surrogate to the the original query. A peripheral attribute is a dimension that is not part of the user query but that must be passed to the packed R-tree query in order to resolve the query on the surrogate. The result of the packed R-tree query is a superset of the records that would have been retrieved had the primary group-by actually existed. However, we note that, since partial cube indexing is most attractive within environments in which data sparsity creates large group-bys of almost identical size, the difference between the sizes of the surrogate result and the actual result are likely to be small in such cases. In addition, since the disk blocks for the packed R-tree are arranged to support contiguous retrieval of disk blocks, the time taken to answer the query will be less influenced by the use of a surrogate because the additional blocks are likely to be accessed within the same disk scans rather than with costly additional disk seeks. These observations are consistent with our experimental results.

When the records have been retrieved, their values must be re-ordered to match the order of attribute values in Q. Furthermore, during this re-ordering, the redundant values for the peripheral attributes of T are removed. Thereby, no additional disk accesses are introduced for the removal of the redundant values.

During the final sort of the query result, it is easy to aggregate, at the same time, the duplicate records that have been introduced by the *peripheral* attributes of the surrogate group-by T. Again, no additional disk accesses are introduced for the removal of the redundant records.

In summary, our partial cube query mechanism is build directly upon the method for completely built datacubes, requiring only very little additional computation. Our experiments, discussed in the following section, show that our distributed query engine is almost as efficient on "virtual" group-bys as it is on ones that actually exist.

4 Performance Analysis

We have implemented our distributed datacube indexing prototype using C++, STL and the LAM MPI communication library, version 6.5.6. The current prototype consists of approximately 8,000 lines of code (not including libraries) and was created by a single programmer over a seven month period.

Our experimental platform consisted of a 17 node Beowulf cluster (a frontend and 16 compute nodes), with 1.8 GHz Intel Xeon processors, 1 GB RAM per node and two 40 GB 7200 RPM IDE disk drives per node. Every node was running Linux Redhat 7.2 with gcc 2.95.3. All nodes were interconnected via an Intel 100 Megabyte Ethernet switch. Note that on this machine communication speed is quite slow in comparison to computation speed. We will shortly be replacing our 100 Megabyte interconnect with a 1 Gigabyte Ethernet interconnect and expect that this will further improve performance results obtainable on this machine.

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 to write the output into files. Furthermore, all wall clock times were measured with no other users except us on the Beowulf cluster.

Figure 4a shows, for an input data set consisting of 10 dimensions and 1,000,000 records, the parallel wall clock time observed for *RCUBE index construction* as a function of the number of processors used. We observe that for index construction our method achieves close to optimal speedup; generating, on 16 processor cluster, the RCUBE index for a fully materialized data cube of ≈640 million rows (17 Gigabytes) in just under 1 minute.

Figure 5 shows parallel wall clock time for *distributed query resolution* as a function of the number of processors used, and the corresponding speedup. In this experiment, batches of ten multi-dimensional queries were resolved against random views in a 10 dimensional data cube consisting of 1,000,000 records, where the queries were constructed to return approximately 15% of the corresponding group-bys. We observe that for distributed query resolution our method achieves good speedup. For example, for 16 processors, a speedup of 13.28 is achieved. The source of the difference between this speedup and perfect speedup is interesting. Perhaps surprisingly, it does *not* arise from the queries returning different numbers of data points on different processors. Hilbert ordering combined with round-robin striping almost perfectly balances the query results evenly over the parallel machine. The small work imbalance observed actually results from the parallel sample sort used to order the query results. This suggests that these speedup results might be further improved by simply using a better sort code.

Figure 6a shows the number of *disk blocks retrieved and corresponding number of disk seeks required* in performing distributed query resolution on views of differing sparsity. Each point represents the average of 15 random queries, each of which returns between 5% and 15% of the associated view, drawn from the 10 dimensional data cube described above. The low density (i.e. sparse) views were typically views high in the lattice, while the high density views were typically views low in the lattice. Again, we observe the benefit of using Hilbert ordering combined with round-robin striping in our distributed RCUBE. Even when a large number of blocks need to be retrieved, the number of disk seeks across our parallel machine is very small. This is crucial to achieving good performance, given that contiguous reads are an order of magnitude faster than reads that require an associated disk seek. Figure 6b shows the *relative record imbalance*, that is the maximum percentage variation between the size of query results on different processors computed over the experiments illustrated in Figure 6a. We observe that the Hilbert ordering combined with round-robin striping leads to a maximum imbalance of less than 0.3% with up to 16 processors.

Figure 7a compares parallel wall clock times for distributed query resolution in *primary and surrogate group-bys* as a function of the number of processors used. Figure 7b shows the corresponding relative cost of a surrogate-based query resolution over the same search in the corresponding materialized primary group-by. We observe from Figure7a that the overhead of using surrogates, that is performing query resolution against non-materialized views, is reasonable small, ranging from 3.5 seconds for a batch of 10 queries on a single processor to 0.12 seconds for the same queries on 16 processors. Figure7b illustrates an interesting trend. As the number of processors grows the relative cost of using surrogate group-bys decreases.

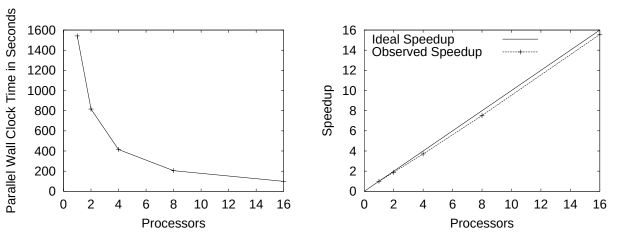

Figure 4: (a) RCUBE index construction, and (b) corresponding Speedup.

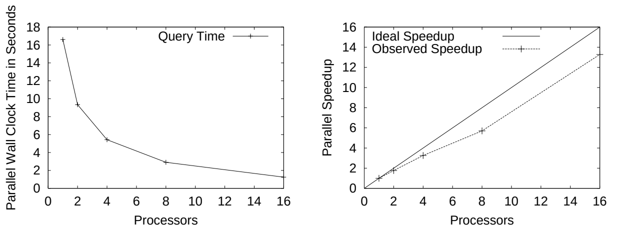

Figure 5: (a) Distributed query resolution, and (b) corresponding Speedup.

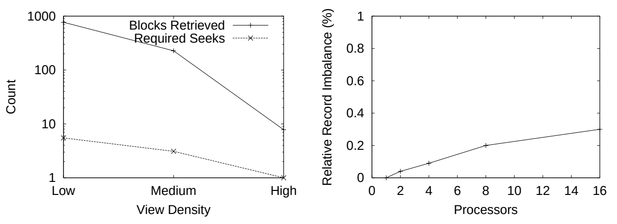

Figure 6: (a) Disk blocks received vs. number of disk seeks required on 16 processors, and (b) Relative record imbalance percentage.

5 Conclusion

In this paper, we have shown that it is possible to build an efficient parallel ROLAP index that is scalable and tightly integrated with the standard relational database approach. Our parallel RCUBE index has the additional advantage of being able to process arbitrary queries on partial datacubes.

References

[1] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and

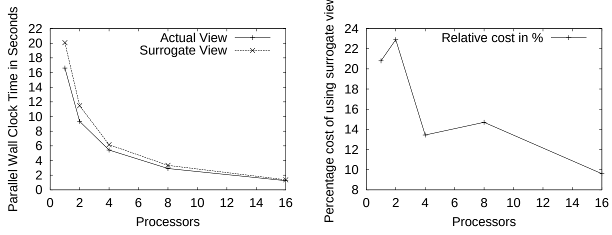

Figure 7: (a) Distributed query resolution in surrogate group-bys, and (b) Relative percentage cost of using surrogate view instead of materialized primary view.

S. Sarawagi. On the computation of multidimensional aggregates. *Proceedings of the 22nd International VLDB Conference*, pages 506–521, 1996.

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

[3] Y. Chen, F. Dehne, T. Eavis, and A. Rau-Chaplin. Parallel rolap data cube construction on shared-nothing multiprocessors. *Dalhousie Faculty of Computer Technical Report*, 2002. http://www.cs.dal.ca/∼arc/publications/2-31/.

[4] F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin. Parallelizing the datacube. *Distributed and Parallel Databases (Special Issue on Parallel and Distributed Data Mining)*, 11(2):181–201, 2001.

[5] F. Dehne, T. Eavis, and A. Rau-Chaplin. A cluster architecture for parallel data warehousing. *IEEE International Symposium of Cluster Computing and the Grid (CCGRid'01)*, 2001.

[6] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. *Symposium on Principles of Database Systems*, pages 247–252, 1989.

[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 subtotals. *Proceeding of the 12th International Conference On Data Engineering*, pages 152–159, 1996.

[9] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index selection for olap. *Proceeding of the 13th International Conference on Data Engineering*, pages 208–219, 1997.

[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] N. Koudas, C. Faloutsos, and I. Kamel. Declustering spatial databases on a multi-computer architecture. In *Proceedings of Extended Database Technologies*, pages 592–614, 1996.

[13] Xiaobo Li, Paul Lu, Jonathan Schaeffer, John Shillington, Pok Sze Wong, and Hanmao Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19(10):1079–1103, 1993.

[14] D. Moore. Fast hilbert curve generation, sorting, and range queries. http://www. caam. rice. edu/ ∼dougm/twiddle/Hilbert.

[15] Oracle9i. RAC (Real Application Clusters). http://otn.oracle.com/products/oracle9i/.

[16] N. Pendse and R. Creeth. The OLAP Report. http://www.olapreport.com/.

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

[18] 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.

[19] 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.

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

[21] B. Schnitzer and S. Leutenegger. Master-client r-trees: a new parallel architecture. *11th International Conference of Scientific and Statistical Database Management*, pages 68–77, 1999.

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