Parallel Data Cubes On Multi-Core Processors With Multiple Disks *

Frank Dehne
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
frank@dehne.net
www.dehne.net

Hamidreza Zaboli
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
hamedzaboli@gmail.com
www.scs.carleton.ca/∼hzaboli

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 development of *MCMD-CUBE*, a new parallel data cube construction method for multi-core processors with multiple disks. We present experimental results for a *Sandy Bridge* multi-core processor with four parallel disks. Our experiments indicate that *MCMD-CUBE* achieves very close to linear speedup. A critical part of our *MCMD-CUBE* method is parallel sorting. We developed a new parallel sorting method termed *MCMD-SORT* for multi-core processors with multiple disks which significantly outperforms the best previous method (PMSTXXL).

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 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.

---

[*] Research partially supported by the IBM Centre for Advanced Studies Canada and the Natural Sciences and Engineering Research Council of Canada.

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 [10, 14]. 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.

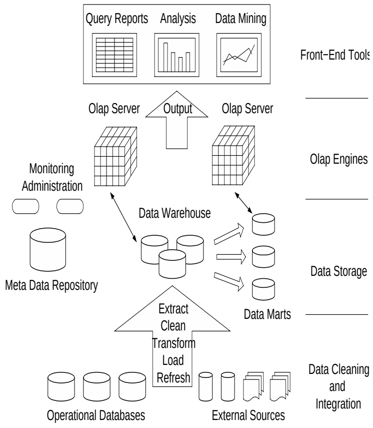

Figure 1: The three-tiered OLAP model.

To support this functionality, OLAP relies heavily upon a data model known as the *data cube* [9, 11]. 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. Figure 2 illustrates a small four-dimensional data cube that might be associated with the automotive industry. In addition to the base cuboid, one can see a number of various planes and points that represent aggregation at coarser granularity. Note that each cell in the cube structure corresponds to one or more *measure* attributes (e.g. *Total Sales*).

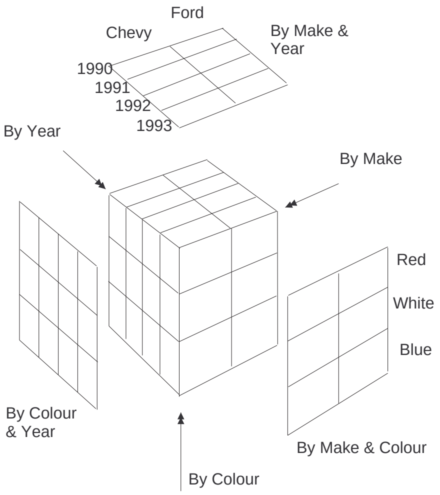

Figure 2: A three dimensional data cube for automobile sales data.

Typically, the collection of cuboids is represented as a *lattice* [11] 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 containing k dimensions can be connected to k views at the next level in the lattice (see Figure 3).

In principle, 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 con-

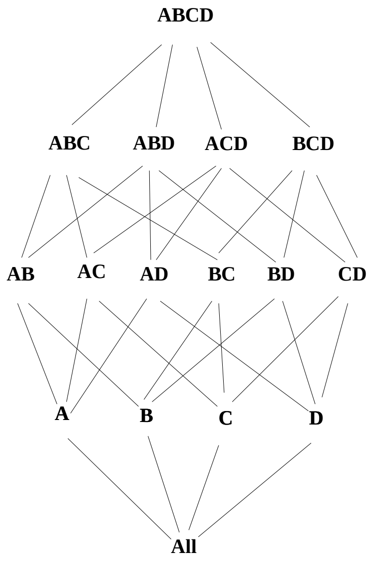

Figure 3: The lattice corresponding to a four dimensional data cube with dimensions A, B, C and D.

junction 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 interrelationships between the various views. Consequently, in 1995, the data cube *operator* (an SQL syntactical extension) was proposed by Gray et al. [9] 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 [4, 6, 11, 12, 13, 15, 16, 17, 18, 20, 21, 22, 23]. The algorithms can 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 on sorting, hashing, and the manipulation of in-memory arrays [4, 17, 23]. 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 [6, 15]. Within each partition, we recursively aggregate at finer levels of granularity until we reach the point where no more aggregation is possible/necessary. Bottom-up algorithms tend to favor views with a larger number of dimensions.

In practice, materialized data cubes can be massive. 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.

Recently, multi-core processors have gained wide acceptance and are now present in nearly all computer systems. This raises an interesting new problem of developing parallel data cube construction methods for such architectures. In this paper we discuss the development of *MCMD-CUBE*, a new parallel data cube construction method for multi-core processors with multiple disks. We present experimental results for a "Sandy Bridge" multi-core processor with four parallel disks. Our experiments indicate that *MCMD-CUBE* achieves very close to linear speedup. Our parallel data cube construction method is based on the classical Pipesort [17] which decomposes the lattice into a sequence of chains called *pipes*, and computes the views in each chain through one external memory sort. Therefore, the performance of our *MCMD-CUBE* methods depends critically on parallel external memory sorting. At the core of our *MCMD-CUBE* method a new parallel sorting method termed *MCMD-SORT* for multi-core processors with multiple disks which significantly outperforms previous methods.

The remainder of this paper is organized as follows. In the following Section 2, we outline our new parallel sorting method *MCMD-SORT* for multi-core processors with multiple disks. In Section 3, we present our new parallel data cube construction method *MCMD-CUBE* for multi-core processors with multiple disks. Section 4 concludes our paper.

2 Parallel sorting on multi-core processors with multiple disks

As discussed above, the performance of our *MCMD-CUBE* data cube computation method depends crucially on parallel external memory sorting. In this section we present an outline of our *MCMD-SORT* algorithm for multi-core processors with multiple disks. Consider a multi-core processors with p cores, M local memory and p disks. We assume a fact table of N data items distributed over those p disks.

2.1 *MCMD-MERGE* algorithm

Our *MCMD-SORT* algorithm is based on a method *MCMD-MERGE* for merging multiple sorted sequences stored on the p disks which is illustrated in Figure 4. For p disk/processor pairs, each disk contains n sorted sequences $S_1, ..., S_n$. The goal is to merge all these sorted sequences into one sorted list L stored on the p disks such that each disk stores $1/p$-th of L. Our *MCMD-MERGE* method is based on an adaptation of deterministic sample sort [19]. As illustrated in Figure 4, each processor/disk pair first independently and in parallel merges its sorted sequences $S_1, ..., S_n$, resulting in p sorted sequences $P_1, ..., P_p$, one on each disk. From each sequence P_i we select p equidistant *local samplers*, collect all p^2 *local samplers* in main memory, sort the p^2 *local samplers*, and then select p *global samplers* from the sorted sequence. As shown in [19], these p *global samplers* define p *well balanced* buckets. Each of the p processors now selects the items in its bucket from the p disks and merges them into one sorted file on its disk. The entire *MCMD-MERGE* procedure can be implemented with two reads and two writes of the entire data set from/to the p disks.

2.2 *MCMD-SORT* algorithm

We now proceed with an outline of our *MCMD-SORT* algorithm. The algorithm proceeds in several stages and is illustrated in Figure 5. We first split the input into N/M blocks of size M, load each block into main memory, sort it using in memory multi-core QuickSort, and write it back to the respective disk. We then select $N^{1/2}$ samples from each block. Here we assume that the total number of samples, $N^{1/2}\frac{N}{M}$, is at most M which implies that $N < M^{3/2}$. If $N > M^{3/2}$, then we will apply an outer level recursion as discussed below. The at most M samples are loaded into main memory, sorted using in memory multi-core QuickSort, and then $M^{1/2}$ equidistant *global samples* are selected. These $M^{1/2}$ *global samples* define $M^{1/2}$ buckets of data consisting of N/M pieces of blocks (sub-buckets). For each bucket, we take the N/M sub-buckets and apply our *MCMD-MERGE* method outlined in Section 2.1.

If $N > M^{3/2}$, then we apply an outer level recursion as illustrated in Figure 6. The entire dataset is partitioned into $N^{1/3}$ sublists of size $N^{2/3}$ and we recurse on each sublist. It is easy to see that the algorithm will not need to recurse for most conceivable cases. For example, for a memory size $M = 2$GB holding 256 million data item, no recursion is required for up to $N = 32$ TB of data.

2.3 *MCMD-SORT* performance analysis

We compared the performance of our *MCMD-SORT* method with the performance of the best currently available multi-core/multi-disk external memory sort (PMSTXXL-SORT) which is part of the PMSTXXL library [1, 3, 2, 5, 7, 8]. More precisely, we implemented our *MCMD-SORT* method in C++ and OpenMP, downloaded PMSTXXL-SORT, and compared their performance on a machine with a *Sandy Bridge* multi-core processor, 16GB of main memory and four parallel disks. In the following, we will first show some more detailed performance data for our *MCMD-SORT* implementation and then show the results of our comparison with PMSTXXL-SORT.

Figure 7 shows *MCMD-SORT* runtime as a function of the number of processors and parallel disks. Note that this is a log-log curve. The straight line indicates that *MCMD-SORT* shows a very close to linear speedup. Figure 8 shows *MCMD-SORT* runtime as a function of data size (again log-log curve). Here, the straight line indicates a very close to linear performance as data size increases. The main contributing factor is that, since we do not need to recurse, the number of reads and writes of the entire data to/from the parallel disks is fixed, and independent of data size. Figure 9 shows *MCMD-SORT* runtime as a function of main memory size. We observe the performance improvements as main memory size increases.

Figure 10 shows a runtime comparison between *MCMD-SORT* and PMSTXXL-SORT. The upper red curve shows the runtime of PMSTXXL-SORT and the lower blue curve shows the runtime of *MCMD-SORT*, as a function of data size. Figure 10 illustrates that the difference in performance between PMSTXXL-SORT and *MCMD-SORT* is dramatically increasing with growing data size. For a data size of 128 GB, *MCMD-SORT* is more than 30% faster than PMSTXXL-SORT. For a data size of 128 GB, the difference is so large that we can run *MCMD-SORT* in 28K seconds but are unable to run PMSTXXL-SORT in any reasonable amount of time.

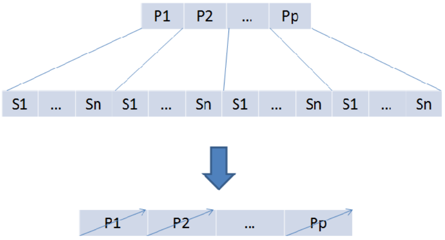

Figure 4: Illustration of our *MCMD-MERGE* algorithm.

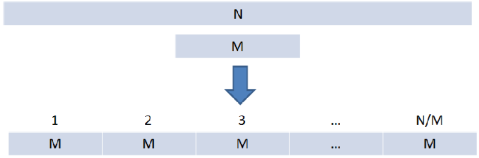

Figure 5: Illustration of our *MCMD-SORT* algorithm.

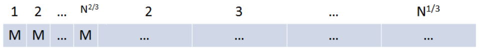

Figure 6: Illustration of *MCMD-SORT* recursion.

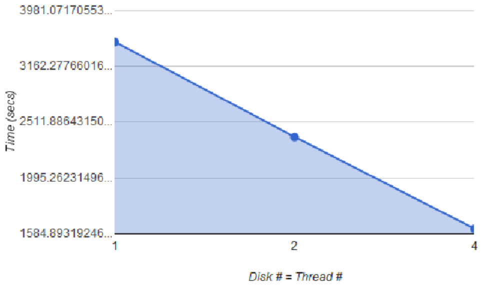

Figure 7: *MCMD-SORT* runtime as a function of number of processors and parallel disks (log-log curve).

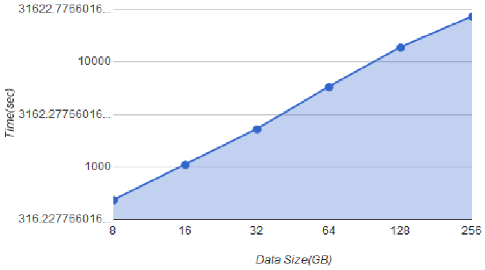

Figure 8: *MCMD-SORT* runtime as a function of data size (log-log curve).

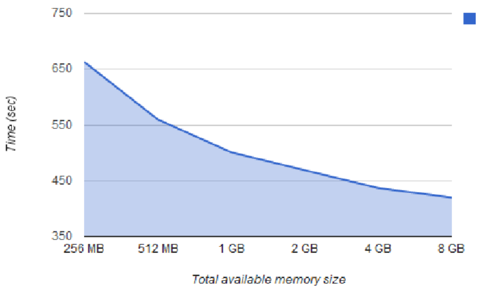

Figure 9: *MCMD-SORT* runtime as a function of main memory size.

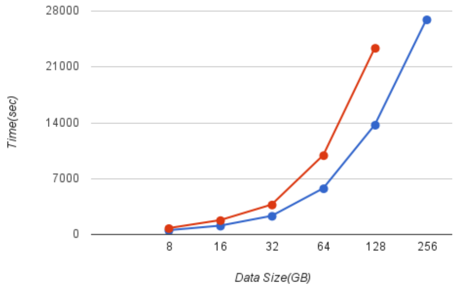

Figure 10: Runtime comparison between *MCMD-SORT* and PMSTXXL-SORT (upper red curve: PMSTXXL-SORT; lower blue curve: *MCMD-SORT*)

3 Parallel data cube construction on multi-core processors with multiple disks

We now turn our attention towards using our *MCMD-MERGE* algorithm for parallel data cube construction on multi-core processors with multiple disks. Our method is based on the classical sequential Pipesort [17] algorithm which decomposes the lattice into a sequence of chains called *pipes*. Figure 11 illustrates the Pipesort algorithm. Given e.g. a five-dimensional fact table with dimensions A, B, C, D, E, the Pipesort algorithm partitions the lattice into sequences of views, called *pipes*, that share the same prefix. For example, Figure 11, one such pipe is ABCDE-ABCD-ABC-AB-A. The full set of pipes in Figure 11 is as follows:

1. ABCDE-ABCD-ABC-AB-A
2. BCEA-BCE-BC-B
3. CDEA-CDE-CD-C
4. DEAB-DEA-DE-D
5. EBDC-EBD-EB-E
6. ADB-AD
7. BDC-BD
8. AEB-AE
9. CEA-CE
10. ACD-AC

For each pipe, the respective views are created by a single sort that creates the first (largest) view of the pipe. The remaining views are then a result of a simple linear scan through the same data because these views are a prefix of the first (largest) view. In fact, the linear scan can be integrated into the sort. Therefore, on a multi-core processor with multiple disks, all views in one pipe can be computed with one single run of our *MCMD-SORT* algorithm. For the example shown in Figure 11, our *MCMD-CUBE* algorithm for building the entire data cube consists of 10 runs of our *MCMD-SORT* algorithm.

Figure 12 shows the *MCMD-CUBE* runtime as a function of the number of processors and parallel disks. Note that this is a log-log curve. The nearly straight line indicates that *MCMD-SORT* shows a very close to linear speedup.

4 Conclusion

In this paper we presented *MCMD-CUBE*, a new parallel data cube construction method for multi-core processors with multiple disks and showed experimental results for a *Sandy Bridge* multi-core processor with four parallel disks.

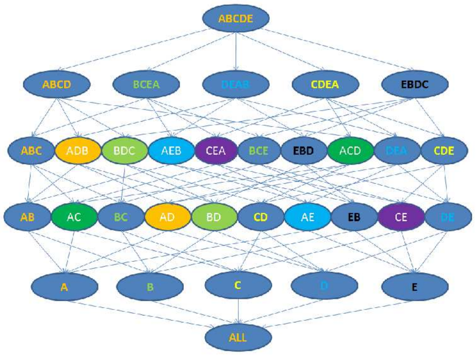

Figure 11: Illustration of the Pipesort algorithm.

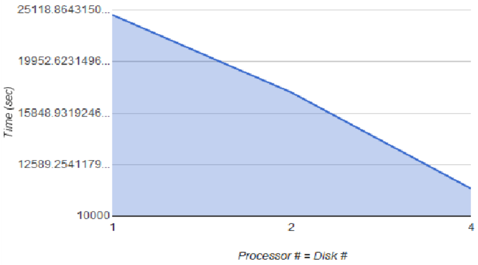

Figure 12: *MCMD-CUBE* runtime as a function of number of processors and parallel disks (log-log curve).

Our experiments indicate that *MCMD-CUBE* achieves very close to linear speedup. A critical part of our *MCMD-CUBE* method is parallel sorting. We developed *MCMD-SORT*, a new parallel sorting method for multi-core processors with multiple disks. Our experiments show that *MCMD-SORT* significantly outperforms PMSTXXL-SORT, the best previous parallel sorting method for multi-core processors with multiple disks.

References

[1] MCSTL: The multi-core standard template library, http://algo2.iti.kit.edu/singler/mcstl/.

[2] PMSTXXL: Multi-core standard template library for extra large data sets (combination of STXXL and MCSTL).

[3] STXXL: Standard template library for extra large data sets, http://stxxl.sourceforge.net/.

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

[5] A. Beckmann, R. Dementiev, and J. Singler. Building a parallel pipelined external memory algorithm library. In *Proc. Int. Parallel and Distributed Processing Symposium (IPDPS)*, 2009.

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

[7] R. Dementiev, L. Kettner, and P. Sanders. STXXL: standard template library for xxl data sets. *Software: Practice and Experience*, 38 (6):58963, 2008.

[8] R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2003.

[9] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Proc. Int. Conference On Data Engineering*, pages 152–159, 1996.

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

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

[12] 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.

[13] 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.

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

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

[16] 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.

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

[18] 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.

[19] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *J. Par. and Dist. Comp.*, 14:362 – 372, 1992.

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

[21] 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.

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

[23] 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.