Parallel Processing Letters
Vol. 23, No. 1 (2013) 1350002 (20 pages)

DOI: 10.1142/S0129626413500023

World Scientific
www.worldscientific.com

# PARALLEL CONSTRUCTION OF DATA CUBES ON MULTI-CORE MULTI-DISK PLATFORMS*

FRANK DEHNE

*School of Computer Science, Carleton University, Ottawa, Canada*
*www.dehne.net, frank@dehne.net*

*and*

HAMIDREZA ZABOLI

*School of Computer Science, Carleton University, Ottawa, Canada*
*hzaboli@connect.carleton.ca, zaboli@graduate.org*

Received April 2012
Revised November 2012
Published 28 March 2013
Communicated by I. Stojmenovic

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, multiprocessor 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 outperforms other previous methods.

*Keywords*: OLAP; parallel data cube; multi-core; multi-disk; external sorting.

*Research partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the IBM Center for Advanced Studies Canada.

## 1. Introduction

### 1.1. *Background (Review)*

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. 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 [8, 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.

To support this functionality, OLAP relies heavily upon a data model known as the data cube [7, 10]. 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 $2^d - 1$ subcubes/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). Typically, the collection of cuboids is represented 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

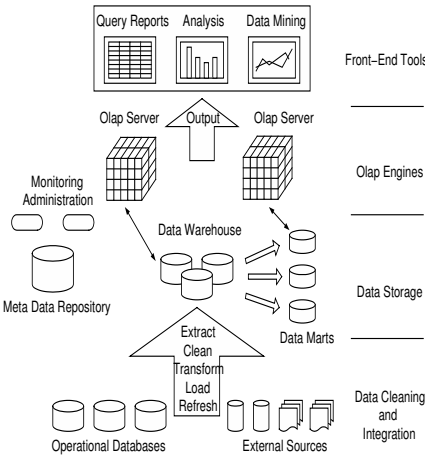



Fig. 1. Three-tiered OLAP model.

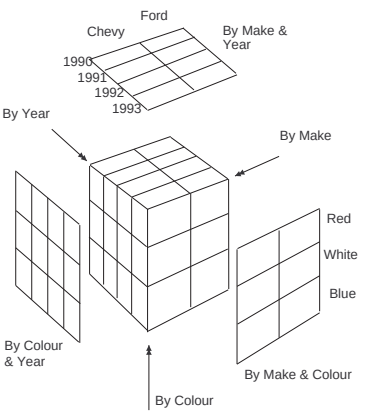



Fig. 2. A three dimensional data cube for automobile sales data.

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. 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 conjunction with 2d sorts of the raw data set to produce all cuboids. However, such an approach would be both tedious to program and immensely inefficient, given the obvious inter-relationships between the various views. Consequently, in 1995,

the data cube operator (an SQL syntactical extension) was proposed by Gray et al. [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, 5, 10, 11, 12, 15, 16, 17, 19, 21, 22, 23, 24]. The algorithms can be divided into two major classes: 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 children. Various techniques have been employed for this purpose, including those based on sorting, hashing, and the manipulation of in-memory arrays [4, 17, 24]. 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 [5, 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.

### 1.2. *Problem Statement*

In practice, materialized data cubes can be massive. Therefore, building data cubes is often a very compute and data intensive operation. With ever increasing corporate databases, this poses a considerable challenge. It is 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 parallel storage. Furthermore, multi-core processors have gained wide acceptance and are now present in nearly all computer systems. This raises an interesting new problem: how to develop *parallel* data cube construction methods that make efficient use of the computational power of multi-core processors and parallel I/O. The aim of our paper is to address this problem.

### 1.3. *Results*

In the remainder of this paper, we discuss the development of MCMD-CUBE, a new parallel data cube construction method for multi-core processors with parallel disks. We present experimental results for a "Sandy Bridge" multi-core processor with four parallel disks. Our experiments indicate that MCMD-CUBE achieves 50% of the theoretically optimal linear speedup. Our parallel data cube construction method is based on the classical Pipesort [4] which decomposes the lattice into a set of chains called pipes, and computes the views in each chain through an external memory sort. Therefore, the performance of our MCMD-CUBE method depends critically on parallel external memory sorting. At the core of our MCMD-CUBE method is a new parallel sorting method termed MCMD-SORT for multi-core processors with parallel disks which significantly outperforms previous methods. We first build

the lattice of data cube and partition the lattice into pipes. Each pipe is a chain of cuboids from top of the data cube toward its bottom. Cuboids in each pipe share one or more dimensions of the data cube so that after computing a cuboid at depth $l$ of the lattice, it is more efficient and easier to build the next cuboid at depth $l+1$ directly from its parent cuboid in depth $l$. The choice of child(ren) cuboids in each level is a matching problem and can be transformed to the weighted bipartite matching problem[4]. Once all cuboids are assigned to their appropriate pipe in the lattice, it is sufficient to apply our MCMD-SORT algorithm only once to each pipe and build the pipe from top to bottom. The remainder of this paper is organized as follows. In the following section, we outline our new parallel sorting method MCMD-SORT for multi-core processors with parallel disks. Our sorting method also includes a new multi-core merging method. In Section 3, we present our parallel data cube construction method MCMD-CUBE for multi-core processors with multiple parallel disks. Section 4 presents discussions and experimental results for MCMD-CUBE and MCMD-SORT, and Section 5 concludes our paper.

## 2. Parallel External Sorting on Multi-Core Processors with Parallel Disks

As discussed earlier, 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 parallel disks. Consider a multi-core processor platform with $p$ cores, a local shared memory of size $M$ and a storage of $d$ disks which can be accessed in parallel by the cores. We assume this platform for the remainder of this paper. We also assume an input data set of size $N$ data records distributed over the $d$ disks.

### 2.1. *Background*

External sorting, also called out-of-core sorting, refers to sorting data items or records residing on external storage. There are numerous fast sorting algorithms presented in the literature, e.g. [3, 2, 6]. Although these sorting algorithms are efficient when running on their specific platforms they may not be efficient when sorting on new parallel platforms. Multi-core processors with shared memory and parallel shared external storage (multiple disks) is a new platform for which only few results have been developed. The best current external memory sorting code available for multi-core processors is a combination of STXXL [6] and MCSTL [1]. STXXL [6] is a standard sorting library including external memory sorting. It was originally designed for single processor platforms but then adapted to work for clusters and later multi-core processors through combination with MCSTL [1], a recent standard library for the new multi-core platforms. In this paper, we propose a new external memory sorting method called MCMD-SORT developed especifically for multi-core processors with parallel disks. We will show that our MCMD-SORT

significantly outperforms STXXL/MCSTL on multi-core processors with parallel disks.

### 2.2. *MCMD-SORT Algorithm*

For single processor platforms and clusters of single processor machines with distributed storage, multi-way merge has been a common method for sorting the data residing on external storage. This method tries to minimize the number of reads and writes to disks since the sorting problem is an I/O-bound problem and needs little computation compared to the slow rate of data transfer from/to disks. In this method, the total data of size $N$ items is partitioned into blocks each of size $M$ that can fit in internal memory. Each of these blocks is loaded from disk into memory and then an internal sorting algorithm sorts each loaded block and writes it back to the disk. After sorting all blocks, a multi-way merge procedure is called and reads data simultaneously from all blocks on disk and merges them into an output buffer. Once the buffer is full, it is written back to disk as the first block including the smallest data records over all $N$ data items. The merge continues until all blocks are completely read and merged. However the merging algorithm will become slow when the total data size ($N$) and consequently total number of blocks is large relative to memory size ($M$). In this case, the method tries to perform a recursive division of $N$ into smaller lists and separately apply the same merging method to blocks of each list. The division continues until the number of lists becomes small enough to be merged using the basic merging procedure. If $N$ becomes large relative to $M$, specially larger than a few hundred gigabytes or terabytes, the number of division-merge steps grows with a rate of $O(\log_{N/M} N)$. This in turn, increases the number of reads and writes to grow logarithmically in $N$. For very large databases, this can become slow and inefficient.

In contrast, our MCMD-SORT presented in this paper uses a scheme similar to deterministic sample sort for clusters [20]. Deterministic sample sort tries to minimize the number of reads/writes to disk(s)and can sort with constant number of reads and writes to disk(s). This is critical for multi-core processors with parallel disks where compute power is abundant and I/O can easily become the bottleneck. In the remainder of this section we present our MCMD-SORT method. We first present our method for a single disk platform and then extend it to parallel multi-disk platforms.

Given an input data set of size $N$ data records, a shared memory of size $M$, and $p$ cores, we consider various cases of data size $N$ compared to $M$. If $N$ is smaller than $M^{3/2}$, $N$ is small enough to be partitioned into $N^{1/3}$(or less) sublists each of size $M$ or smaller. Otherwise, $N$ data records will be divided into sublists each of size $M^{3/2}$. If the number of sublists of size $M^{3/2}$is larger than $N^{1/3}$, then a recursive division on $N$ will be performed until we have nested lists and sublists each of size $M^{3/2}$ and $M$, respectively, with a total number of $N^{1/3}$(or less) sublists in each list and $N^{1/3}$(or less) lists over all $N$ data records. Figure 3 illustrates the division step.

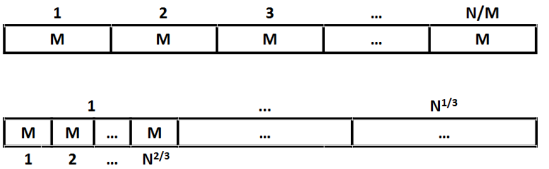



Fig. 3. Input data division into smaller blocks of data.

After the division step, the data is partitioned into sublists of size $M$ and possibly one list of size $N'$ or smaller. $N'$ will be equal to $N$ in case $N \leq M^{3/2}$where only one round of division is needed. According to our calculations, the ratio of the power of $\frac{3}{2}$ is so large that even very large input data sizes (several terabytes) can be sorted with only one round of input data division, which is one advantage of our sorting method over other sorting merthods based on multi-way merge.

An important feature of our method is that even for extremely large data sets, the division step will never need to recurse at all. For example, with an internal memory $M$ of size 2 giga bytes, which is small compared to todays memory sizes, and one round of data division without recursion, MCMD-SORT can sort up to 32 tera bytes of 8-byte data records. With $M = 8$ giga bytes, as available on our test platform, MCMD-SORT can sort up to 256 tera bytes of 8-byte data records with only one round of division (without any recursion). This is very important because one round of division implies a very small, fixed, number of external data accesses.

When the division step is complete, we have blocks of data each of size $M$. Each block is read from disk and loaded into internal memory. In this step, the loaded blocks are sorted and written back to disk. Therefore we need to apply an internal memory multi-core sorting method to sort each block. Recently, many sorting methods have been proposed for multi-core processors. STL sort available in the STL library, libmt sort [13],[9], [18] are state-of-the-art sorting methods. We tested these sorting methods on in memory data to choose the fastest of them. We observed for our multi-core platform that the latest STL sort implementation with full multi-core support outperforms the other recent sorting implementations for multi-core. Hence we chose STL sort as part of our MCMD-SORT for sorting blocks of size $M$. Note that in case of any new faster internal memory sorting method, STL sort can be easily replaced by that method which in turn makes MCMD-SORT sort faster as well.

After each block is sorted using STL sort on $p$ cores, before it is written back to disk, $M^{1/2}$ samples with equal distance from each other will be taken from $M$ items of the block and stored in memory. Then the next block of size $M$ is loaded into memory until all blocks on disk have been sorted.

In the next step, the set of samples stored in memory will be merged. Because the samples from each block are in the ascending order, they together build $N^{1/3}$ small sorted buckets. We need to make one sorted list of samples out of them. This can be done using a merging algorithm rather than a complete sorting algorithm.

This is the first instance where we apply our new multi-core merging algorithm to be presented in the next section.

After merging all samples, we will have a sorted list of size $N^{1/3} \times N^{1/3}$ equal to $N^{2/3}$ which is equal to $M$. $M$ samples can be merged in memory without accessing external memory. After merging all samples we choose a subset of samples called *global samples* (GS). More precisely, we choose $N^{1/3}$ equidistant global samples out of the $N^{2/3}$ sorted samples and store them in memory. We observe that the number of global samples is so small that they can remain in memory.

In the next step, we take each global sample $GS_i$, and compute bucket $B_i$ as follows. We read from each sorted sublist in external memory into main memory the data records that are smaller than $GS_i$ (and larger than the the previous $GS_{i-1}$ for later rounds). As shown in [20], the total size of data records in $B_i$ will not exceed $2 \times N^{1/3} \times \frac{N^{2/3}}{N^{1/3}}$. Therefore, if we choose $M$ to be half of the available internal memory size, then the total loaded data will not exceed available memory.

After loading data for $B_i$, we can merge them instead of using a complete sorting method because it consists of $N^{1/3}$sorted sublists. Therefore, we again apply our multi-core merging algorithm presented in the next section. Finally, each $B_i$ is written back to disk. Note that all elements in $B_i$ are larger than all elements in $B_{i-1}$. Hence, after completing the process for all $B_i$, all $N$ data records are sorted in external storage.

### 2.3. *MCMD-Merge Algorithm*

Note that all merging steps of MCMD-SORT are executed on data loaded into internal memory. This is an advantage of our MCMD-SORT because it generates sorted sublists that can fully fit into the internal memory of size $O(M)$. We now outline an internal memory multi-core merging algorithm.

Given a total set of $M$ data items in $B$ sorted sublists, and $p$ processor cores, we assign $\frac{B}{p}$ sublists to each core. Each core merges its $\frac{B}{p}$ sublists using a binary merging process. Note that all cores start merging processes at the same time and they work in parallel on independent data. After this step we will have $p$ sorted sublists. Now we need to merge the $p$ sorted sublists using $p$ cores. Here again we need to divide each of the $p$ sublists into $p$ smaller buckets and then distribute them between the $p$ cores. We need to assign equal work loads to the $p$ cores, i.e. the total number of items assigned to each core should not exceed a maximum threshold. This can be achieved by applying again a deterministic sampling method. Note that randomized sampling is also applicable which will result in a slight reduction of work complexity. However, there would not be any guarantee on the total size of buckets assigned to each core. Therefore, we choose to apply a deterministic sampling method as follows.

We start by assigning one sublist to each core. Next, each core takes $p$ equidistant sample items from its sublist. After that, the $p^2$ selected samples are sorted using any multi-core sorting method. Note that, the total of $p^2$ is so small that does not

require applying any merging methods. In the next step, core $p_i$ takes the $(ip)^{th}$ sample item out of the $p^2$ sorted samples and finds its location in each of $p$ sublists using a parallel binary search algorithm. Next, core $p_i$ takes for each sublist *1...p* all items smaller than the $(ip)^{th}$ sample and larger than the $(i-1)p^{th}$ sample. Therefore, each core $p_i$ obtains $p$ buckets containing all data items smaller than the $(ip)^{th}$ sample and larger than $((i-1)p)^{th}$ sample. Note that data items in each bucket are already sorted. In the last step, core $p_i$ merges its $p$ buckets by applying a binary merge method. After all cores completed their merging processes, all of $M$ items are sorted.

## 3. Parallel Data Cube Computation on a Multi-Core Multi-Disk Platform

Since the appearance of data cube operator in [7], there have been many methods proposed for computation of data cubes. However, computation of data cubes on multi-core platform is still in its infancy. One might consider that a simple task since, after all, a data cube with $d$ dimensions is composed of $2^d$ cuboids which all need to be created. At first sight, there seems to be more than enough parallelism. However, efficient data cube construction methods do not build the $2^d$ cuboids independently but use relationships between cuboids to improve efficiency. Examples include the top-down and bottom-up methods discussed in Section 1.1. Any parallel data cube construction method needs to utilize these relationships between cuboids or risk adding additional work. That makes parallelism at the cuboid level complicated. Another problem for parallelism at the cuboid level is that different cuboids can have very different sizes. Therefore, assigning different cuboid computations to different processors can lead to serious load balancing problems.

In this paper, we choose to create parallelism at a finer level of granularity. We utilize the classical Pipesort sequential method [4] and parallelize each of the pipes generated by Pipesort. A critical part of generating each pipe as outlined in [4] is external memory sorting. That is why our MCMD-SORT method presented earlier will be a critical component of our solution. In the following section, we present our MCMD-CUBE method.

### 3.1. *MCMD-CUBE Computation*

Our method is based on the classical Pipesort sequential top-down computation of data cubes [4]. Given a data cube with $d$ dimensions of cardinalities $< D_1, D_2, D_3, ..., D_d >$ there are $2^d$ cuboids to be computed. These $2^d$ cuboids can be ordered to be computed based on the lattice of the data cube. An example of this lattice for a data cube with 5 dimensions is shown in Figure 4. In this figure, arrows show the order in which pipes are formes and cuboids computed. In the first step, those cuboids that can be computed from other cuboids are specified. This will form a parent-child relationship between each two levels $i$ and $i+1$ of the lattice. Each parent can have many children and each child can be computed from different

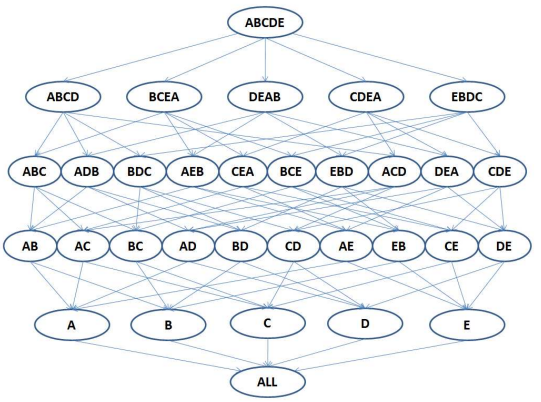



Fig. 4. Lattice of a 5-dimensional data cube.

parents. Therefore, for each child we need to specify from which parent cuboid it should be computed. We need to find the parent that applies the minimum cost of computation for the child node. This cost can be calculated according to the estimated sizes of each cuboid which itself can be calculated using cardinalities of the dimensions in a cuboid. The problem of matching parents and children for each two levels $i$ and $i+1$ can be reduced to a bipartite matching problem. Details are presented in [4].

After assignment of each child to a parent, the lattice of the data cube can be divided into pipes each containing a series of cuboids which can be computed from each other. More precisely, we start building each pipe by computing the first parent cuboid which is the largest cuboid in the pipe and then computing child cuboids using the parent. We continue computing the cuboids in the pipe in parent-child order until all cuboids are computed. When we compute a cuboid, we apply our MCMD-SORT method to compute the cuboid using all cores. This guarantees full usage of all cores while load balancing the computation.

Given a five-dimensional fact table with dimensions A, B, C, D, E, the corresponding lattice shown in Figure 4 is partitioned into pipes. For example, in Figure 4, one such pipe is ABCDE-ABCD-ABC-AB-A. The full set of pipes of the lattice in Figure 4 is listed in Table 1. As shown in Table 1, we obtain 1 pipe of length 5 containing 6 cuboids, 4 pipes of length 3 each containing 4 cuboids, and 5 pipes of length 1 each containing 2 cuboids each. Note that the cuboid "All" does not need to be computed.

For our parallel MCMD-CUBE method, the computation of pipe is parallelized as follows. We apply MCMD-SORT to the first cuboid of the pipe. Then we parallel aggregate with respect to the dimensions not included in the first cuboid. We note that aggregation can be parallelized by partitioning the sorted cuboid into $p$ segment, aggregating independently and in parallel on each segment, and then aggregating in parallel across the $p$ segment boundaries. Next, we extract all the

Table 1. 10 pipes of the lattice of Figure 4.

| Pipes |
|---|
| ABCDE-ABCD-ABC-AB-A |
| BCEA-BCE-BC-B |
| CDEA-CDE-CD-C |
| DEAB-DEA-DE-D |
| EBDC-EBD-EB-E |
| ADB-AD |
| BDC-BD |
| AEB-AE |
| CEA-CE |
| ACD-AC |

remaining cuboids in the pipe by aggregating again in parallel with respect to the other dimensions that are removed for those cuboids. Here, we follow again the classical Pipesort methods [4] but replace each sequential aggregation by a parallel aggregation as outlined above. The major benefit of this approach is that we still utilize the relationships between cuboids that lead to work reduction while obtaining maximum parallelism and work balance.

## 4. Experimental Results

We now present experimental results for our MCMD-SORT and MCMD-CUBE methods. We tested our MCMD-CUBE method by building data cubes on both synthetic databases and the standard TPC database benchmark. We also report experimental results for our MCMD-SORT method and compare it with STXXL sorting package and its parallel STXXL/MCSTL version called PMSTXXL which currently is the fastest external memory sorting method for multi-core platform.

Our experimental setup includes a machine with a Sandy Bridge Intel processor and 16 GB of internal memory shared between 8 cores. ( Note that, not all of the 16 GB can be used due to OS limits.) Our external storage consists of 4 parallel disks that can be accessed independently by each core. All implementations were performed in OpenMP and run on Linux kernel 2.6.38.

Our experiments are divided into two groups: sorting and data cube computation. In the first group of experiments, we tested our MCMD-SORTing method on the above platform with a variety of parameters and configurations. We also ran our MCMD-SORT against PMSTXXL and observed that MCMD-SORT achieved better performance. In the second group of experiments, we compute data cubes with two sets of data (synthetic and TPC benchmark), and observed close to linear speedup with respect to the number of parallel disks and processor cores.

### 4.1. *Experimental Results for MCMD-SORT*

We tested our MCMD-SORT method with respect to the impact of total data size, number of processor cores and disks, internal memory size, and record size. As a

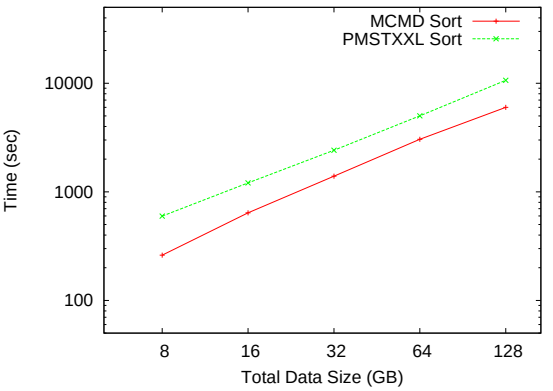
standard configuration, our data includes records of 8 bytes and we use all of 8 bytes as sort key. Where we test the impact of record size, we change our record structure and use a 8-byte key followed by a data field of desired size. All sort keys are generated randomly unless otherwise mentioned. Tests of our MCMD-SORT and PMSTXXL sort are peformed on the same above mentioned machine.

The first set of tests are performed to observe the impact of growing total data size. In this test, the only variable parameter is total size of the data to be sorted. Data can resides on 1, 2, or 4 disks. Where there are more than one disk, we evenly distribute the data between disks. Fixed parameters are record size equal to 8 bytes and memory size equal to 8 Giga bytes. This test includes three parts each performed on a fixed number of disk(s) and processors core(s). The results of this set of tests are shown in Figure 5(a-c).

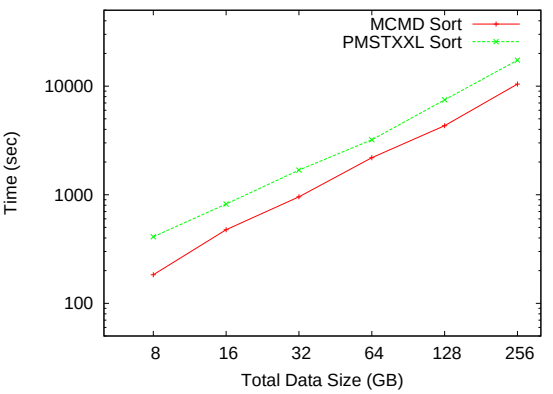
In Figure 5(a-c), both the data size and time axis are logarithmic. Therefore, the difference in performance between our MCMD-SORT and PMSTXXL are substantial. The runtime difference between the two methods on 128 Giga bytes of data on 1-disk-1core, 2-disk-2-core, and 4-disk-4-core configurations are 4635, 2367, and 1785 seconds, respectively. These differences imply 44%, 32%, 48% of lower sorting time for our our MCMD-SORT compared to PMSTXXL.

The second set of tests examines the speedup of our MCMD-SORT method. Figure 6 shows speedup curves for two data sizes, 32 GB and 128 GB, with full memory usage. The only variable parameter in this test is the platform configuration which can vary between 1-disk 1-core, 2-disk 2-core, and 4-disk 4-core. Figure 6 shows that our MCMD-SORT method sorts faster by increasing number of processor cores and disks. We were not able to install more parallel disks on our platform but we expect to see faster sorts with 8 disks, 8 cores. Figure 6 shows that our MCMD-SORT method obtains close to optimal, linear speedup. We observed a 30 percent improvement on total sorting time when moving from a 1-disk 1-core to a 2-disk 2-core configuration. From 2-disk 2-core configuration to 4-disk 4-core, we observed a 40 percent improvement in total sorting time. The curve for 128 GB of data has a larger negative slope compared to the curve for 32 GB. This effect is due to a better utilization of parallel cores for larger data sizes.

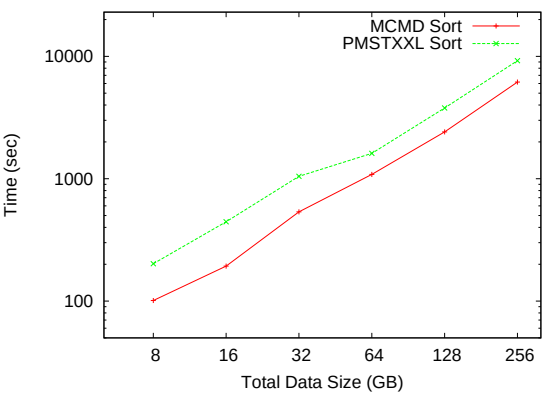
The third set of tests shows the impact of memory size. In this experiment, the only variable parameter is the total memory size available to our MCMD-SORT method. Fixed parameters are total data size (32 Giga bytesor 128 Giga bytes as in the second test) and record size of 8 bytes, using a 4-disk 4-core configuration. The results are shown in Figure 7. By increasing the memory size available to the cores, each time they can load larger blocks of data into memory and sort them. This does have a major impact on the block sorting part of the algorithm. However in the merging part, we also observed a large difference due to the reduced number of blocks to be merged, especially on large data sets. By increasing memory size, sorting time decreases as the number of merging rounds decreases. We observed that on the 32 Giga byte data set, because total data size is not large compared to total memory size (between 4 and 32 times larger), the decrease in total sorting time is




(a)




(b)




(c)

Fig. 5. Impact of growing data size on sorting time for different hardware configurations. Memory size: 8 GBytes, Record size: 8 bytes. (a) 1 disk, 1 processor core. (b) 2 disks, 2 processor cores. (c) 4 disks, 4 processor cores.

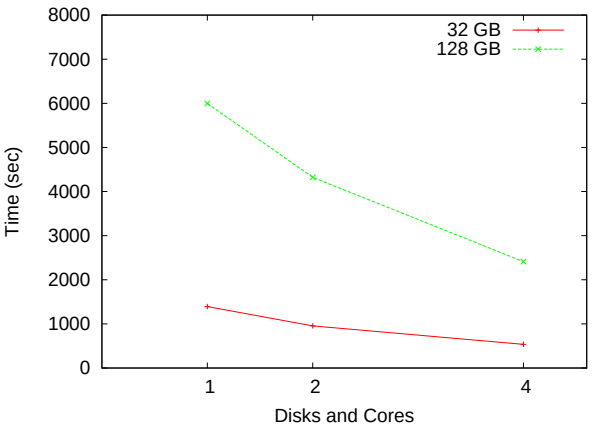



Fig. 6. Speedup curves for two data sizes, 32 GB and 128 GB. Record size: 8 bytes, Memory size: 8 GB.

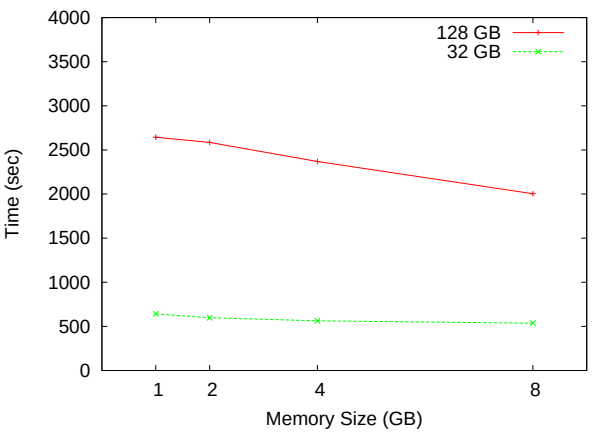



Fig. 7. Impact of memory size on sorting time. Record size: 8 bytes, 4 disks, 4 cores.

slight. However when dealing with larger data sets such as 128 Giga bytes, increasing memory size has a significant impact on the merging part and consequently total sorting time. As shown in Figure 7, for a 128 Giga byte data set with 8 GB of memory, total sorting time is significantly lower than with 1 GB of memory.

The last set of tests for our MCMD-SORT method shows the impact of record size. In this experiment, we changed the structure of each data record and increased the size of data records from 8 bytes up to 64 bytes. Size of data records is the only variable parameter in this test. The sort key is always 8 bytes but the data field changes from 0 to 56 bytes. By increasing the record size and keeping the same total size for the entire data to be sorted, the number of records decreases. The amount of the data to be moved between memory and disks is fixed (128 GB).

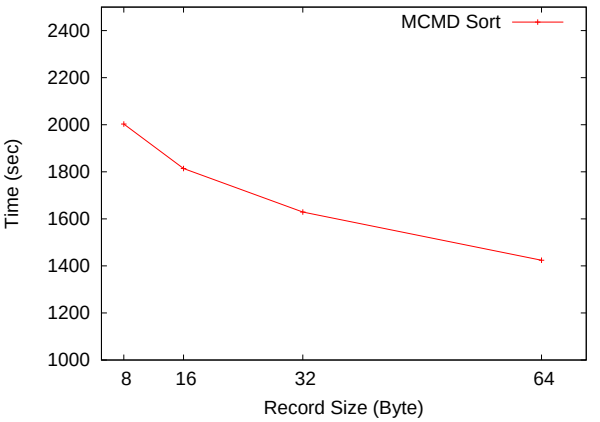



Fig. 8. Impact of record size on sorting time. Total data size: 128 GB. Memory size: 8 GB. 4 disks. 4 cores.

For example, when our record sizes are 8 bytes, we have 16 Giga records equal to $16 \times 2^{30}$records. When we increase the records size to 64 bytes, we have 8-byte keys with 56 bytes for data part in each record. In this case, the total number of records to be sorted is 2 Giga records equal to $2 \times 2^{30}$records. As expected we observe that the total sorting time decreases with increasing the record size because we sort a smaller number of records. However we are moving the same amount of data, and this data I/O prevents us from achieving full speedup. In Figure 8, total sorting time decreases with a smaller slope compared to slopes of the curves in Figure 6. This experiment shows that our MCMD-SORT method is I/O bound like all other sorting methods. It also highlights the importance of the small fixed number of disk accesses in MCMD-SORT achieved by the deterministic sampling method.

### 4.2. *Experimental Results for MCMD-CUBE*

We tested MCMD-CUBE on two databases: a synthetic random generated database and the standard TPC database benchmark. Both data cubes are computed on the same hardware platform. The TPC benchmark includes a variety of different data warehouses. In our experiments, we used the TPC-DS data warehouse.

#### 4.2.1. *Random data sets*

In this set of experiments, we created a data warehouse with 5 dimensions. Data records in dimension tables are generated randomly including primary keys that are used in the fact table. We integrated these primary keys in the fact table and generated the fact table. Each record in the fact table contains primary keys of the 5 dimensions. All keys are generated randomly. We refer to these dimensions as A,B,C,D,E. The cardinalities of the dimensions are 1024, 512, 512, 1024, and 256, respectively.

We first built the lattice of the data cube and created all pipes in the lattice. Table 1 summarizes the chosen pipes of the lattice shown in Figure 4. Then we start reading the fact table and keys in each row and apply our MCMD-CUBE method. In our sample data cube, we start with cuboid ABCDE and compute it from the fact table by sorting data records based on the order of concatenated keys in cuboid ABCDE. Next cuboids in ABCDE pipe can be generated from ABCDE by a single pass over it. We continue computing cuboids of Table 1 in the same way until all pipes are built. By building all pipes, data cube is computed.

Although we have 5 dimensions in our database, we added some data fields to the records of the fact table to make it similar to the TPC-DS database. Our fact table records have 164 byte of length which is equal to the average record size in TPC-DS benchmark. When building the data cube, we transfer the whole record to/from internal memory including keys of the 5 dimensions and data part which can contain measures and other attributes of dimensions.

Considering the above approach, our first experiment is to compute our random data cube using different hardware configurations with growing the size of fact table. Figure 9 summarizes this experiment on our 5-dimensinal database.

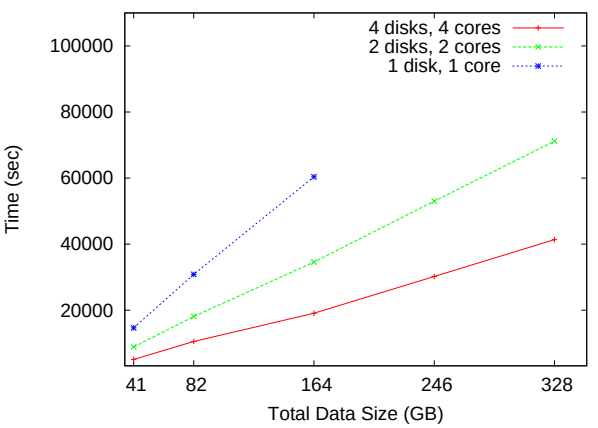



Fig. 9. Computing data cube of randomly generated data warehouse with 5 dimensions. Memory size: 8 GB. Record size: 164 bytes.

Figure 9 shows the impact of total data size on the runtime of MCMD-CUBE as well as the speedup obtained through the use of multiple processor cores and multiple disks. The size of the fact table varies from $0.25 \times 164$ GB to $2 \times 164$ GB. With increasing data size, total time of data cube computation increases smoothly. Figure 9 also shows the impact of different disk and processor core configurations. The configuration with 4 disks and 4 core computes the data cube nearly twice as fast as the configuration with 2 disks and 2 cores, indicating close to linear speedup.

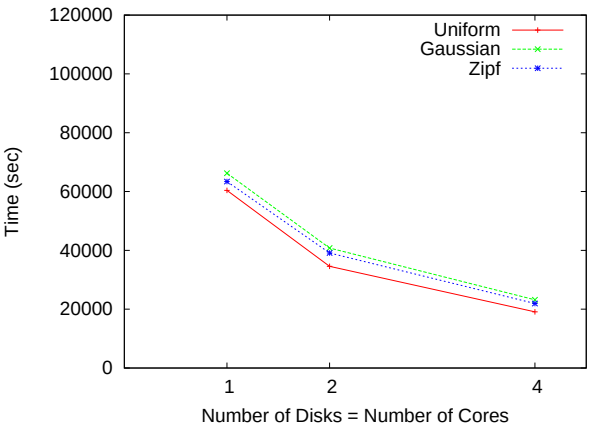



Fig. 10. Computing data cube on different data distributions. Memory size: 8 GB. Record size: 164 bytes.

The impact of different data distributions on the speedup is shown in Figure 10. As we mentioned earlier, our records and keys were first generated with a unifrom distribution. We also generatedd our random data base with Gaussian and Zipf distributions and then used our MCMD-CUBE method to build data cubes on these distributions. As expected, there was only minimal (within measurement noise) fluctuation of MCMD-CUBE's performance in the presence of different distributions. This is due to the fact that MCMD-CUBE uses *deterministic* sample sort which is not influenced by the data distribution. Figure 10 shows that for all three distributions, MCMD-CUBE achieves close to linear speedup. Note that when doubling the number of disks and cores, total computation time will not be half. This is due to bottlenecks that appear when dealing with reading from/writing to parallel disks. Both hardware and OS cause a bottleneck and prevent us to fully exploit the parallel power of disks and cores. However, total data cube computation time with 4-disk 4-core configuration is almost 33 percent of the total time with 1-disk 1-core configuration.

#### 4.2.2. *TPC-DS databse benchmark*

The TPC-DS database benchmark is a well known database benchmark. We chose this benchmark to compute data cubes using our MCMD-CUBE method. The TPC-DS benchmark has 10 dimensions and 7 fact tables. The largest fact table is the table 'Store_Sales'. We chose this fact table and computed its data cube using our MCMD-CUBE. Each row of the table contains attribute values of the 10 dimensions and the measure. Average record size of the fact table is 164 bytes. Out of the 10 dimensions, we chose 5 key dimensions and computed a data cube for these 5 dimensions. These dimensions are: 'Item', 'Customer', 'Store', 'Promotion', and 'Ticket Number'.

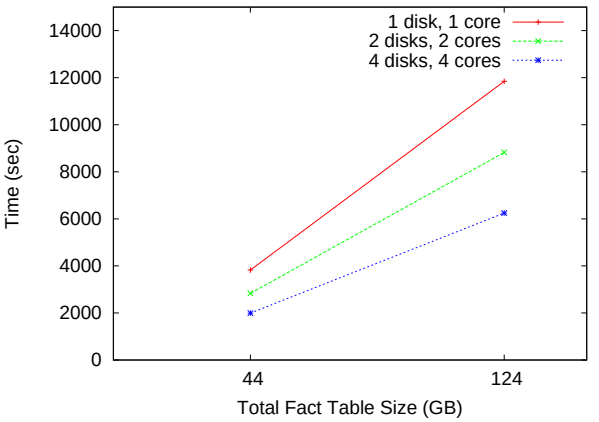



Fig. 11. Computing data cube of TPC-DS benchmark. Memory size: 8 GB. Record size: 164 bytes.

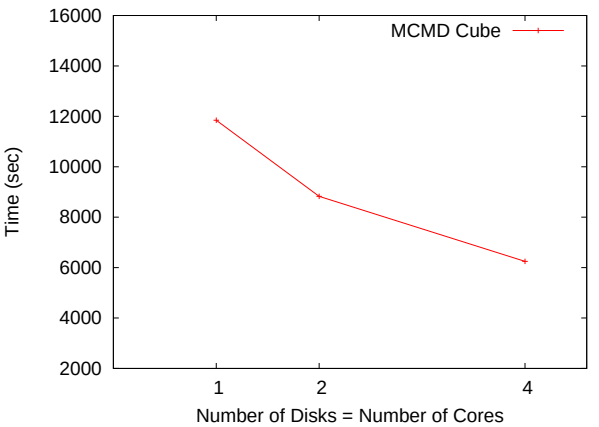



Fig. 12. Speedup curve for computing data cube of TPC-DS benchmark. Fact table size: 124 GB. Memory size: 8 GB. Record size: 164 bytes.

Figure 11 shows the impact of total data size on the runtime of MCMD-CUBE as well as the speedup obtained through the use of multiple processor cores and multiple disks. (Note that the TPC-DS benchmark is only available in specific sizes .) With increasing data size, total time of data cube computation increases smoothly. Figure 11 also shows the impact of different disk and processor core configurations. As also shown in Figure 12, the configuration with 4 disks and 4 cores computes the data cube nearly twice as fast as the configuration with 1 disk and 1 core, indicating approximately 50% of optimal linear speedup.

Figure 13 shows the impact of internal memory size on the performance of MCMD-CUBE. Using larger memory sizes allow the method to load more data

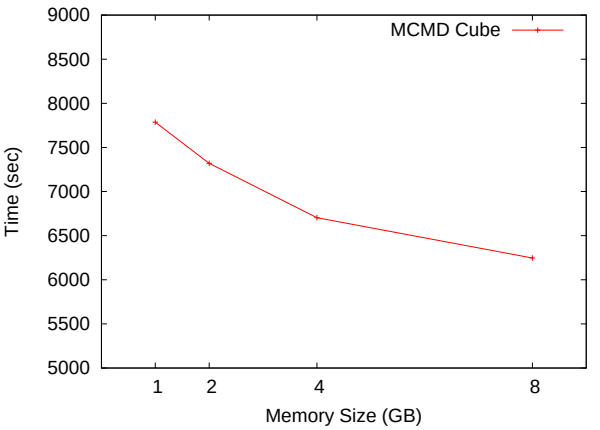



Fig. 13. Impact of memory size on computing time of TPC-DS data cube. Fact table size: 124 GB. Record size: 164 bytes.

records at once and creates larger pipes with only one pass of disk read/write. This effect is a consequence of the lower work load of the MCMD-SORT when sorting with larger memory sizes. Therefore, when increasing the memory size available to MCMD-CUBE, the total time for building data cubes decreases.

## 5. Conclusion

On-line Analytical Processing (OLAP) has become one of the most powerful and prominent technologies for knowledge discovery in VLDB (Very Large Database) environments. 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 discussed the development of MCMD-CUBE, a new parallel data cube construction method for multi-core processors with parallel disks. Our experimental results for a "Sandy Bridge" multi-core processor with four parallel disks indicate that MCMD-CUBE achieves 50% of the theoretically optimal linear speedup.

## 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] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. *Proc. ACM SIGMOD Conference*, pages 359–370, 1999.
[6] R. Dementiev, L. Kettner, and P. Sanders. Stxxl: standard template library for xxl data sets. *Software: Practice and Experience*, 38 (6), 2008.
[7] 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.
[8] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.
[9] Song Hao, Zhihui Du, Bader D.A., and Yin Ye. A partition-merge based cache-conscious parallel sorting algorithm for cmp with shared cache. *Parallel Processing, 2009. ICPP '09. International Conference on*, pages 396–403, 2009.
[10] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes. *Proceedings of the 1996 ACM SIGMOD Conference*, pages 205–216, 1996.
[11] 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.
[12] L.V.S. Lakshmanan, J. Pei, and Y. Zhao. Qc-trees: An efficient summary structure for semantic OLAP. *Proceedings of the ACM SIGMOD Conference*, pages 64–75, 2003.
[13] C multithread library (libmt). http://www.libmt.sourceforge.net.
[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] Nadathur Satish. Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort. *Proceedings of the 2010 international conference on Management of data*, 2010.
[19] 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.
[20] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *J. Par. and Dist. Comp.*, 14:362 – 372, 1992.
[21] Y. Sismanis, A. Deligiannakis, N. Roussopolos, and Y. Kotidis. Dwarf: Shrinking the petacube. *Proceedings of the 2002 ACM SIGMOD Conference*, pages 464–475, 2002.
[22] 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.
[23] 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.
[24] 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.