Iceberg-cube Computation with PC Clusters

Raymond T. Ng
Univ British Columbia
2366 Main Mall, UBC
Vancouver, BC, V6T 1Z4
rng@cs.ubc.ca

Alan W agner
Univ British Columbia
2366 Main Mall, UBC
Vancouver, BC, V6T 1Z4
wagner@cs.ubc.ca

Yu Yin
Univ British Columbia
2366 Main Mall, UBC
Vancouver, BC, V6T 1Z4
yuyin@cs.ubc.ca

ABSTRACT

In this paper, we investigate the approach of using low cost PC clusters to parallelize the computation of iceberg-cube queries. We concentrate on techniques directed towards online querying of large, high-dimensional datasets where it is assumed that the total cube has not been precomputed. The algorithmic space we explore considers trade-offs between parallelism, computation and I/O. Our main contribution is the development and a comprehensive evaluation of various novel, parallel algorithms. Specifically: (1) Algorithm RP is a straightforward parallel version of BUC [BR99]; (2) Algorithm BPP attempts to reduce I/O by outputting results in a more efficient way; (3) Algorithm ASL, which maintains cells in a cuboid in a skiplist, is designed to put the utmost priority on load balancing; and (4) alternatively, Algorithm PT load-balances by using binary partitioning to divide the cube lattice as evenly as possible.

We present a thorough performance evaluation on all these algorithms on a variety of parameters, including the dimensionality of the cube, the sparseness of the cube, the selectivity of the constraints, the number of processors, and the size of the dataset. A key finding is that it is **not** a **one-algorithm-fit-all** situation. We recommend a "recipe" which uses PT as the default algorithm, but may also deploy ASL under specific circumstances.

Keywords

OLAP, Parallel Computation

1. INTRODUCTION

For many decision support and OLAP applications, CUBE queries constitute one of the most important classes of queries. The CUBE operator, introduced by Gray et al., generalizes the GROUP-BY operator in computing aggregates for every possible combination of the specified attributes [7]. However, for d specified attributes, not only are there 2^d cuboids (i.e., group-bys), but there are also numerous cells, or partitions, computed. Analyzing such a huge amount of output is not easy.

To address this issue, two main approaches have been studied so far. The first approach is to tightly integrate OLAP with data mining techniques. For example, the framework proposed by Sarawagi summarizes partitions into patterns on the one hand, but identifies exceptions to those patterns on the other [14]. The second approach, which is the subject matter of this paper, is to allow user-specified constraints to be imposed on the partitions. Following the spirit of iceberg queries studied in [5], Beyer and Ramakrishnan introduced and studied the Iceberg-cube problem, which computes only those partitions satisfying a given aggregate condition [3, 11]. They developed the BUC algorithm, which proceeds bottom-up by starting from the cuboid with "all", to a cuboid on a single attribute, then on a pair of attributes, and so on.

While user-defined constraints help to reduce the size of the output, the computation is still far away from being truly "online" – in the sense of supporting real time user interactivity. One general solution is to selectively pre-compute or materialize some of the cuboids (e.g., [9, 8, 2, 15]). While materialization is a step in the right direction, we feel that materialization alone is not adequate. To provide for fast answering of iceberg-cube queries, it is more appropriate to materialize partitions, a granularity level finer than cuboids. But selecting the "right" partitions to materialize appear to be harder than selecting cuboids. This is because the "right" partitions depend on the actual constraints, e.g., the type of the constraints, and the thresholds specified. It is an open question how to effectively select partitions to materialize for iceberg-cube queries.

In this paper, we investigate the approach of using PC clusters to parallelize the computation of iceberg-cube queries. We concentrate on techniques directed towards online querying of large, high-dimensional datasets where it is assumed that the total cube has not been precomputed. Furthermore, we focus on practical techniques that could be readily implemented on low cost PC clusters using open source, Linux and public domain versions of the MPI message passing standard.

Towards efficient iceberg-cube computation with PC clusters, this paper explores different trade-offs between parallelism, computation and I/O. The main contribution of this

Algo.	Writing Strategy	Load Balance	Relationship of cuboids	Data Decomposition
RP	depth-first	weak	bottom-up	replicated
BPP	breadth-first	weak	bottom-up	partitioned
ASL	breadth-first	strong	top-down	replicated
PT	breadth-first	strong	hybrid	replicated

Figure 1: Key Features of the Algorithms

paper is the *development and comprehensive evaluation of various novel, parallel algorithms* for iceberg-cube computation. Specifically:

- We develop algorithm RP (*Replicated Parallel* BUC), which is a straightforward parallel version of BUC. Though simple, algorithm RP does a poor job in distributing tasks and workload. As an attempt to correct the situation, we develop algorithm BPP (*Breadth-first writing, Partitioned, Parallel* BUC), which differs from RP in two key aspects. First, the dataset is not replicated, but is range partitioned on an attribute basis. Second, the output of cuboids is done in a breadth-first fashion, as opposed to the depth-first writing that RP and BUC do. Figure 1 summarizes the key features of the algorithms.
- Though an effort has been made, the two algorithms are likely to be weak on load balancing. This is primarily because they follow the BUC-style bottom-up computation too strictly. To consider load balancing as the utmost priority, we pursue a radically different strategy. We develop algorithm ASL (*Affinity SkipList*), which maintains the cells of a cuboid in a skiplist. Furthermore, to allow as much shared computation as possible within each processor, ASL assigns cuboids to processors in a way that amounts to building the cuboids in a top-down fashion.
- While ASL tries to load balance by processing very fine granularity tasks, algorithm PT (*Partitioned Tree*) is a hybrid algorithm combining both pruning and load balancing, and processing tasks of slightly coarser granularity. The idea is to use binary partitioning to divide the lattice of the cuboids as evenly as possible.
- The natural questions to ask at this point are: (i) which algorithm is the best, and (ii) do we really need to know about all these algorithms? For the first question, we present a thorough performance evaluation of all these algorithms on a variety of parameters. The parameters include the dimensionality and sparseness of the cube, the selectivity of the constraints, the number of processors, and the size of the dataset. With respect to the second question, a key finding of our evaluation is that when it comes to iceberg-cube computation with PC clusters, it is **not** a **one-algorithm-fit-all** situation. Based on our results, we recommend a "recipe" which uses PT as the default algorithm, but may also deploy ASL under specific circumstances.
- Algorithmic development and evaluation aside, we consider "truly online" more than just speed and efficiency. We believe that the kind of online aggregation framework proposed and studied by Hellerstein, Haas and Wang is valuable [10]. To this end, we study how well the different algorithms proposed here can be augmented to provide such support.

The outline of the paper is as follows. Section 2 reviews key concepts and the main sequential algorithms for iceberg-cube computation. Section 3 presents the model and the assumptions underlying our study here. Section 4 introduces the various novel parallel algorithms. Section 5 presents comprehensive experimental results, and concludes with a recipe for picking the best algorithms under various circumstances. Finally, Section 6 discusses how well the various algorithms can support online processing.

2. REVIEW

The data cube on m attributes can be represented as a lattice structure, called the *cube lattice*, where each node in the lattice is called a *cuboid*. Figure 2 shows the cube lattice for four attributes A, B, C and D. The cuboids are labeled according to the attributes values that have been aggregated. For example AB represents the cuboid where each record is the sum of the aggregation field over all distinct combinations of attribute values for AB. If the data set has been sorted with respect to AB then in order to calculate A the dataset does not have to be re-sorted and we can simply accumulate the sums for each of the values in A. This optimization is called *share-sort* and can be used whenever a cuboid is a prefix of another cuboid.

The general iceberg-cube query framework is designed to handle constraints other than count constraints. For the latter constraint, a cell in a cuboid is only returned as part of the answer to the query if there are enough tuples assigned to that cell; the number of tuples assigned is called the *support* of the cell. Like [3], we only consider count constraints here.

Sequential algorithms that compute the entire data cube can be viewed as having two stages: the planning stage and the execution stage. In the planning stage, the algorithm decides how to decompose the lattice into a collection of disjoint sets of nodes; the union of all these sets make up the complete cube. In the execution stage, for each set of nodes, the algorithm computes the actual cuboids. Among the existing algorithms, some are sort-based, while the others are hash-based. In [3], there is a comparison between the two classes of algorithms. It is concluded that hash-based algorithms do not exploit shared computation as much as the sort-based algorithms do, and that hash-based algorithms require a significant amount of memory. For these reasons, in this paper, we focus our attention on sort-based algorithms.

Returning to the planning stage, algorithms which follow paths from the raw data towards the total aggregate value are called "top-down" approaches. Algorithms which compute paths in the reverse direction are called "bottom-up" approaches. For the example shown in Figure 2, a top-down approach computes from ABCD, to ABC, to AB and eventually to A; a bottom-up approach goes in the opposite direction. Bottom-up approaches do not take effective use of share-sort, whereas top-down approaches can. However, for iceberg queries, bottom-up approaches can exploit pruning.

```
1.  Algorithm BUC-Main
2.  INPUT: Dataset $R$ with dimensions $\{A_1, A_2, \ldots A_m\}$,
        the minimum support $Spt$.
3.  OUTPUT: Qualified cells in the $2^m$ cuboids of the cube.
4.  PLAN:
5.      Starting from the bottom, output the aggregate on "all",
          and then a depth first traversal of the lattice
          induced by $\{A_1, A_2, \ldots A_m\}$.
6.      for each dimension $A_i$ ($i$ from 1 to $m$) do
          BUC($R$, $\mathcal{T}_{A_i}$, $Spt$, $\{\}$)
7.  CUBE COMPUTATION:
8.      procedure BUC($R$,$\mathcal{T}_{A_i}$, $Spt$, prefix)
9.        prefix = prefix $\cup\{A_i\}$
10.       for each combination of values $v_j$ of the attributes
            in prefix do
11.         partition $R$ to obtain $R_j$
12.         if (the number of tuples in $R_j$ is $\geq Spt$)
13.           aggregate $R_j$, and write out the
                aggregation to cuboid with cube
                dimensions indicated by prefix
14.           for each dimension $A_k$, $k$ from $i+1$ to $m$ do
15.             call BUC($R_j$,$\mathcal{T}_{A_k}$, $Spt$, prefix)
16.           end for
17.       end for
```

Figure 3: A Skeleton of BUC

Specifically, if there is not sufficient support for a_ib_j, then it can be concluded that sufficient support cannot possibly exist for all of a_ib_jC, a_ib_jCD, and so on.

One of the first algorithms designed for efficient cube creation was PIPESORT [1]. This algorithm attempts to minimize the overall computation by obtaining cost estimates of different ways of covering the lattice with paths. Specifically, it computes a plan in a level-by-level manner, where at each level it uses estimates of the size of the cuboids to determine which cuboid to compute the cuboid at the next lower level. When the plan is executed, each path can be computed in a pipelined manner. While PIPESORT is amenable to parallel implementation, it does so at the expense of communication in the pipeline. For low-cost PC clusters, the overhead of communication easily dominates any savings in the amount of computation. Thus, we do not pursue parallelizing PIPESORT further.

More Recently Ross and Srivastava have designed a very effective top-down algorithm for large, high-dimensional and sparse data cubes [13]. Their algorithm, called Partitioned-Cube, adopts a top-down approach for the planning state. For the execution stage, it tries to make efficient use of available memory to reduce the number of scans of the data. They found that reducing the number of scans was a better measure of the performance of the algorithm on sparse data versus attempting to minimize the amount of work, as was the case with PIPESORT.

As noted by Beyer and Ramakrishnan [3], for iceberg-cube queries, top-down algorithms are not able to prune cells with insufficient support, which can add to the amount of computation and the amount of memory needed during the computation. Thus, in BUC, a bottom-up approach is adopted. For the example given in Figure 2, BUC starts with "all", and then attribute A. For each value v_j in A, the dataset is partitioned and BUC is called recursively in a depth-first manner to process the other dimensions. While it can exploit pruning, BUC does so at the expense of abandoning the use of share-sort.

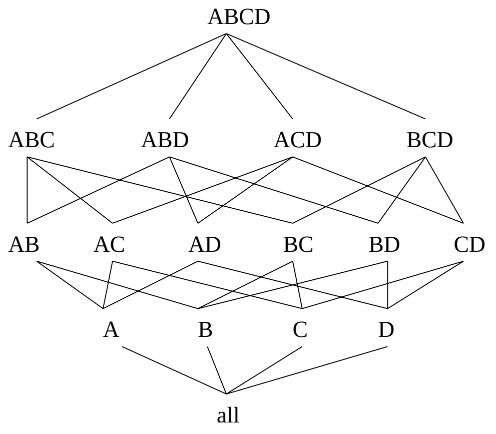
A skeleton of BUC is shown in Figure 3, we use the notation $\mathcal{T}_{A_i}$ to denote the set of all nodes in the subtree rooted at A_i. For the example given in Figure 2, $\mathcal{T}_B = \{B, BC, BD, BCD\}$.

3. MODEL AND ASSUMPTIONS

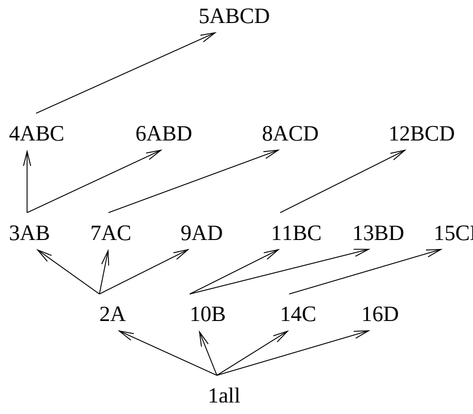
Key to the success of an online system is the ability for the system to respond to queries in a timely fashion. Given the compute and data intensive nature of iceberg-cube queries, it necessitates a high performance machine. In the past, this required expensive platforms, e.g., symmetric multiprocessor machines. In recent years, however, a very economical alternative has emerged – namely, a cluster of low-cost commodity processors and disks. There are several advantages to using PC-clusters. First, in terms of raw performance, processor speeds are similar to and often exceed those of multiprocessor architectures. Recent advancements in system area networks, such as Myrinet, standards like VIA, and 100Mbit or Gigabit Ethernet has significantly improved communication bandwidth and latency. Although I/O and the use of commodity disks is a weakness of these systems, as we show, parallelism can easily be exploited. The affordability of PC-clusters makes them attractive for small to medium sized companies and has been the dominant parallel platform used for many applications [4], including association rule mining [17].

In the remainder of this paper, we develop various novel algorithms for parallelizing iceberg-cube queries. Our focus is on practical techniques that could be readily implemented on low cost PC clusters using open source, Linux and public domain versions of the MPI message passing standard. While our results apply to low cost clusters, a natural question to ask is how much we expect our results to generalize to higher cost systems. As will be shown in Section 5, we examine how the various algorithms would speed up in the presence of more nodes/processors in the cluster. Thus, if the key difference between a low cost and a high cost cluster is only on the number of nodes, then our results will be applicable. However, if the key difference is on the underlying communication layer, then our results may not be applicable.

All of the algorithms to be presented use the basic framework of having a planning stage and an execution stage. In the case of parallel algorithms, the planning stage involves (i) breaking down the entire processing into smaller units called *tasks*, and (ii) assigning tasks to processors, where now the objective is to minimize the running time of the processor that takes the longest time to complete its tasks. To simplify our presentation, we do not include the aggregation for the node "all" as one of the tasks. This special node can be easily handled. Furthermore, it is assumed that the initial dataset is either replicated at each of the processors or partitioned. The output, i.e., the cells of cuboids, remains distributed, where processors output to their local disks.



(a) cube lattice



(b) BUC processing tree

Figure 2: An Example Cube

4. PROPOSED PARALLEL ALGORITHMS

In this section, we introduce the four algorithms. As shown in Figure 1, the algorithmic space that we explore involves the following issues:

- The first issue is how to write out the cuboids. Because BUC is bottom-up, the writing of cuboids is done in a depth-first fashion. As will be shown later, this is not optimized from the point of view of writing performance. This leads us to develop an alternative breadth-first writing strategy.
- The second issue is the classical issue of load balancing. This issue is intimately tied to the definition of what a task is. Different algorithms essentially works with different notions of a task. In general, when the tasks are too coarse-grained, load balancing is not satisfactory. However, if the tasks are too fine-grained, a lot of overhead is incurred.
- When it comes to iceberg-cube queries, an important issue is the strategy to traverse the cube lattice. As discussed earlier, top-down traversal exploits share-sort, whereas bottom-up traversal exploits pruning based on the constraints. Our algorithms consider these possibilities; and in fact, one of the algorithms combines the two strategies in an interesting way.
- As usual, for parallel computation, we explore whether data partitioning is effective.

As an preview, there is a relationship between the four algorithms presented here. We first attempted a straightforward parallel version of BUC. Since the planning stage of BUC divides the lattice into several independent trees, these trees were simply distributed and executed in parallel. This algorithm is called *RP*. But RP exhibits poor load balancing behavior. As a remedy, we developed the algorithm called *BPP*, which adopts breadth-first writing and data partitioning. But it turns out load balancing is not improved as much as we expected. This leads us to the development of the algorithm called *ASL*, which is intended to deal with tasks much smaller than those handled by the previous algorithms. While it uses skiplists to maintain the cuboids directly, it also exploits share-sort as much as possible. This is one way of achieving better load balancing. Another way is to further optimize on BPP (i) to create finer-grained tasks, and (ii) to exploit share-sort. This leads to the development of the algorithm called *PT*.

```
1.  Algorithm RP
2.  INPUT: Dataset $R$ with dimensions $\{A_1, A_2, \ldots A_m\}$ and
              minimum support $Spt$;
3.  OUTPUT: The $2^m$ cuboids of the data cube.
4.  PLAN:
5.      Task definition: identical to BUC, i.e., subtrees
            rooted at $A_i$
6.      Processor assignment: assign a processor, in
            round robin fashion, to each subtree rooted at
            dimension $A_i$ ($i$ from 1 to $m$)
7.  CUBE COMPUTATION (for a processor):
8.      parallel do For each subtree rooted at dimension $A_i$
            assigned to the processor
9.          call BUC($R, \mathcal{T}_{A_i}, Spt, \{\}$)
                (with output written on local disks)
10.     end do
```

Figure 4: A Skeleton of the RP Algorithm

4.1 Algorithm RP

Recall from Figure 2 that BUC breaks down the entire cube lattice into independent subtrees rooted at each of the dimensions. Thus, in the algorithm called Replicated Parallel BUC, RP for short, each of these subtrees becomes a task. In other words, for a cube query involving m attributes, there are m tasks. Processor assignment is simply done in a round robin fashion. With this simple assignment strategy, if there are more processors than tasks, some processors will be idle. Each processor reads from its own copy of the dataset, and outputs the cuboids to its local disk (see Figure 4).

For example, for the cube from Figure 2, if there are two processors, the first processor would be assigned all cuboids with dimensions starting with A and D, and the second processor would compute all cuboids with dimensions starting with B, and those starting with C.

4.2 Algorithm BPP

4.2.1 Task Denition and Processor Assignment

While RP is easy to implement, it appears to be vulnerable in at least two areas. The first problem is that the definition of a task may be too simplistic in RP. The division of the cube lattice into subtrees is coarse-grained. One consequence is that some tasks are a lot bigger than some others. For example, the subtree rooted at A, $\mathcal{T}_A$, is a lot larger than that rooted at C, $\mathcal{T}_C$. Thus, load balancing is poor. The second problem is that BUC is not optimized in writing performance. Below we address the first problem; in Section 4.2.2 we try to solve the second problem.

To address the first problem, the algorithm called Breadth-first writing Partitioned Parallel BUC, BPP for short, tries to get finer-grained tasks by range partitioning on each attribute. This is motivated by Ross' and Srivastava's design objective of Partitioned-Cube in trying to partition the data into chunks that fit in memory [13]. To be more precise:

- For a given attribute A_i, the dataset R is range partitioned into n chunks (i.e., $R_{i(1)}, \ldots, R_{i(n)}$), where n is the number of processors. Processor P_j keeps its copy $R_{i(j)}$ on its local disk.
- Note that because there are m attributes, the above range partitioning is done for each attribute. Thus, processor P_j keeps m chunks on its local disk, i.e., $R_{1(j)}, \ldots, R_{m(j)}$. Any of these chunks may have some tuples in common.
- Range partitioning itself for the m attributes can be conducted in parallel, with processor assignment done in a round robin fashion. For instance, processor i may partition attribute A_i, then A_{i+n}, etc. Notice that as far as BPP execution is concerned, range partitioning is basically a pre-processing step.

As is obvious from the above description, there are a total $m \times n$ chunks. Each chunk corresponds to one task. For example, for the chunk $R_{i(j)}$, processor P_j is assigned the task of computing the (partial) cuboids in the subtree rooted at A_i. These cuboids are partial because P_j only deals with the part of the data it controls, i.e., $R_{i(j)}$. But of course, the cuboids are completed by merging the output of all n processors.

4.2.2 Breadth-First Writing

Because BUC computes in a bottom-up manner, the cells of the cuboids are written out in a depth-first fashion. To illustrate, consider the situation shown in Figure 5. There are three attributes A, B, C, where the values of A are a_1, a_2, values of B b_1, b_2, etc. As shown in Figure 3, the tuples of a_1 is aggregated in line 14 (assuming that the support threshold is met), and the result is output. The recursive call in line 15 then leads the processing to the cell a_1b_1, then the cell $a_1b_1c_1$, then $a_1b_1c_2$, etc. In Figure 5, the number in round

```
1.  Algorithm BPP
2.  INPUT: Dataset $R$ with dimensions $\{A_1, A_2, \ldots A_m\}$ and
            minimum support $Spt$
3.  OUTPUT: The $2^m$ cuboids of the data cube
4.  PLAN:
5.      Task definition: (partial) cuboids of subtrees rooted at $A_i$
6.      Processor assignment: as described in Section 4.2.1
7.  CUBE COMPUTATION (for the processor $P_j$):
8.      parallel do
9.          for each $A_i$ ($i$ from 1 to $m$) do
10.             call BPP-BUC($R_{i(j)}, \mathcal{T}_{A_i}, Spt, \{\}$)
                    (with output written on local disks)
11.         end for
12.     end do

13. Subroutine BPP-BUC($R, \mathcal{T}_{A_i}, Spt$, prefix)
14.     prefix = prefix $\cup\{A_i\}$
15.     sort $R$ according to the attributes ordered in prefix
16.     $R' = R$
17.     for each combination of the values of the attributes
            in prefix do
18.         if (the number of tuples for that combination $\geq Spt$)
19.             aggregate on those tuples, and
                    write out the aggregation
20.         else remove all those tuples from $R'$
21.     end for
22.     for each dimension $A_k$, $k$ from $i+1$ to $m$ do
23.         call BPP-BUC($R', \mathcal{T}_{A_k}, Spt$, prefix)
24.     end for
```

Figure 6: A Skeleton of the BPP Algorithm

brackets beside each node denotes the order in which the cell is processed and output for depth-first writing.

Note that these cells may belong to different cuboids. For example, the cell a_1 belongs to cuboid A, the cell a_1b_1 to cuboid AB, and the cells $a_1b_1c_1$ and $a_1b_1c_2$ belong to ABC. The point is that in depth-first writing, the writing to the cuboids is scattered. This clearly incurs a high I/O overhead. It is possible to use buffering to help the scattered writing to the disk. However, this may require a large amount of buffering space, thereby reducing the amount of memory available for the actual computation. Furthermore, many cuboids may need to be maintained in the buffers at the same time, causing extra management overhead.

In BPP, this problem is solved by breadth-first writing. To return to the example in Figure 5, BPP completes a cuboid before moving on to the next one. For example, the cells a_1 and a_2, which make up cuboid A, are first computed and written out. Then all the cells in cuboid AB are output, and so on. In Figure 5, the number in angled brackets beside each node denotes the order in which the cell is processed and output for breadth-first writing.

Figure 6 gives a skeleton of BPP. As described above, there is the pre-processing step of range partitioning the dataset, and assigning to each processor P_j the appropriate tasks, namely computing the partial cuboids of the subtree rooted at A_i based on the chunk $R_{i(j)}$, for all $1 \leq i \leq m$.

In the main subroutine BPP-BUC, breadth-first writing is implemented by first sorting the input dataset on the "prefix" attributes. For our example, if the prefix is A, meaning that the dataset has already been sorted on A, then line 15

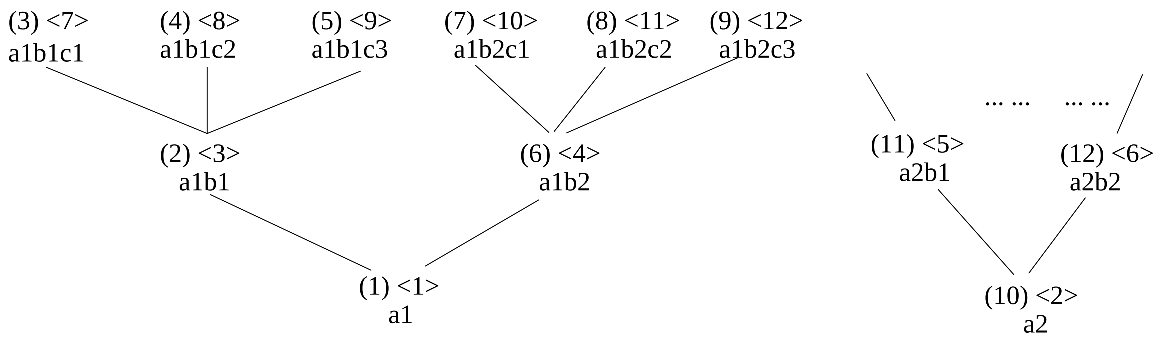


Figure 5: Depth-first Writing vs Breadth-first Writing

sorts the dataset further on the next attribute B. The loop starting from line 17 then basically completes breadth-first writing, by computing and outputting the aggregation of all the cells in the cuboid AB.

Because some cells may not meet the support threshold, there is the extra complication in BPP-BUC to prune as early as possible. This is the purpose of lines 16 and 20. Note that as opposed to what is presented in line 16 for simplicity, in our implementation, we do not actually create a separate copy of the data. Instead, an index is used to record the starting and ending positions in the sorted dataset to indicate that all those tuples should be skipped for subsequent calls to BPP-BUC.

Breadth-first I/O is a significant improvement over the scattering I/O used in BUC. For the baseline configuration to be described in Section 5, the total I/O time for RP to write the cuboids was more than 5 times greater than the total I/O time for BPP.

4.3 Algorithm ASL

The potential disadvantage of BPP is that the amount of work each processor does is dependent on the initial partitioning of the data. The size of the task depends on the degree of skewness in the dataset and the order in which the dimensions are sorted and partitioned. If the skewness is significant, the tasks may vary greatly in size, thereby reducing load balancing. The motivation for the algorithm called Affinity SkipList, ASL for short, is to try to create tasks that are as small as the cube lattice allows. This would allow efficient use of the processors, quite independent of the the skewness and dimensionality of the dataset. As explained below, there are two key features of ASL – namely, the data structure used, and the processor assignment.

Concerning the data structure, ASL uses a skiplist to maintain the cells in one cuboid. More specifically, it iteratively reads in the tuples, inserts each tuple into the cell in the skiplist, and updates the aggregate and support counts. ASL is parallelized by making the construction of each cuboid a separate task. The hope is that this creates a large number of small tasks and leads to better overall load balancing. In theory, if there are k cuboids and if there is enough memory, ASL can maintain all k skiplists simultaneously for one scan of the dataset. But for the datasets used in our experiments, this optimization brings minimal gain, and we did not do that here.

```
Algorithm ASL
1.  INPUT: Dataset $R$ cube dimensions $\{A_1, \ldots, A_m\}$;
            minimum support $Spt$
2.  OUTPUT: The $2^m$ cuboids of the data cube
3.  PLAN:
4.      Task definition: a cuboid in the cube lattice
5.      Processor assignment: a processor is assigned
            the next task based on prefix or subset affinity
6.  CUBE COMPUTATION (for a processor):
7.      parallel do
8.          let the task be with dimensions $A_i, \ldots, A_j$
9.          if ($A_i, \ldots, A_j$ is the prefix of the
                previous task or the first task)
10.             let $C$ be the skiplist of that task
11.             call prefix-reuse($C, Spt, A_i, \ldots, A_j$);
12.         else if ($\{A_i, \ldots, A_j\}$ is a subset of
                the set of dimensions of the previous task, or
                the set of dimensions of the first task)
13.             let $C$ be the skiplist of that task
14.             call subset-create($C, Spt, A_i, \ldots, A_j$)
15.         else call subset-create($R, Spt, A_i, \ldots, A_j$)
16.     end do

17. Subroutine prefix-reuse($C, Spt, A_i, \ldots, A_j$)
18.     Aggregate $C$ based on $A_i, \ldots, A_j$
19.     Write out the cells meeting the support threshold

20. Subroutine subset-create($C, Spt, A_i, \ldots, A_j$)
21.     initialize skip list $L$
22.     for each cell (tuple) in $C$ do
23.         find the right cell in $L$ (created if necessary)
24.         update the aggregate and support counts accordingly
25.     end for
26.     Traverse L, writing out the cells meeting
            the support threshold
```

Figure 7: A Skeleton of ASL

Skiplists were chosen as the data structure because they exhibit good average case behavior for insertion and searching, can be efficiently implemented, and have the nice property that the sorted cells are stored as a linked list that can easily be used to write out the final cuboid. We also experimented with hash tables. However, they did not perform as well as skiplists and would require the use of special extendible hash functions in order for us to exploit share-sort. They also occupied more space. This is consistent with one of the conclusions given in [1] that hash-based algorithms perform poorer and poorer as the sparenses of the dataset increases.

Next, we turn our attention to the processor assignment policy of ASL. Contrary to the algorithms seen so far, ASL adopts a top-down approach to traverse the cube lattice. The hope is obviously to try to maximize the benefit of share-sort.

For example, if a processor has just created the skiplist for ABCD, then it makes perfect sense for the processor to be assigned the task of computing the cuboid for ABC. The previous skiplist for ABCD can simply be reused to produce the results for ABC. In the following, we refer to this situation as "prefix affinity".

As another situation, if a processor has just created the skiplist for ABCD, this skiplist is still useful if the processor is next assigned the task of computing the cuboid BCD, say. This is because all that needs to be done is to take the counts of each cell in ABCD, and add them to the counts of the appropriate cell in the skiplist for BCD. This may bring about significant savings because the groupings done for the skiplist for ABCD need not be wasted. For example, suppose in ABCD, there is the cell corresponding to the grouping of $a_1b_1c_1d_1$. For the w tuples in the original dataset that belong to this cell, the current aggregate and support counts can readily be used to update the corresponding counts for the cell $b_1c_1d_1$ for BCD. There is no need to re-read the w tuples and aggregate again. In the following, we refer to this situation as "subset affinity".

Figure 7 shows a skeleton of Algorithm ASL. To implement prefix or subset affinity, a processor is designated the job of being the "manager" who has the responsibility of dynamically assigning the next task to a "worker" processor. Specifically, the manager:

- first tries to exploit prefix affinity, because if that is possible, the worker processor has no need to create a new skiplist for the current task/cuboid. The previous skiplist can be aggregated in a simple way to produce the result for the current task. This is executed by the subroutine prefix-reuse in Figure 7.
- then tries to exploit subset affinity, if prefix affinity is not applicable. Instead of scanning the dataset, the worker processor can use the previous skiplist to create the skiplist for the current task. This is executed by the subroutine subset-create in Figure 7.
- assigns to the worker a remaining cuboid with the largest number of dimensions, if neither prefix nor subset affinity can be arranged. In this case, a new skiplist is created from scratch.

Clearly, the last situation ought to be avoided as much as possible. In our implementation of ASL, each worker processor maintains the first skiplist it created. Because ASL is top-down, the first skiplist corresponds to a cuboid with a large number of dimensions. This maximizes the chance of prefix and subset affinity.

4.4 Algorithm PT

By design, ASL tries to do a very good job of load balancing the computation. However, ASL may be vulnerable in two areas. First, the granularity of the tasks may be too fine – to an extent that too much overhead is incurred. This is particularly true in situations where prefix or subset affinity cannot be exploited well, thus reducing the amount of sorting that can be shared. Second, ASL cannot easily prune the tuples not having minimum support. This is a direct consequence of ASL not using a bottom-up traversal. As ASL executes, whether a cell has minimum support or not, cannot be determined until the dataset has been scanned. Furthermore, at the end of the scan, even if there is a cell that is below the minimum support, this cell still cannot be pruned, because its support may contribute to the support of another cell in subsequent cuboid processing.

The algorithm called Partitioned Tree, PT for short, attempts to solve these two problems as follow:

- Recall that in RP and BPP, tasks are at the granularity level of subtrees rooted at a certain dimension, e.g., $\mathcal{T}_{A_i}$. In ASL, tasks are merely nodes in the cube lattice. To strike a balance between the two definitions of tasks, PT works with tasks that are created by a recursive binary division of a tree into two subtrees having an equal number of nodes.

  For instance, for the BUC processing tree shown in Figure 2, it can be divided into two parts, namely $\mathcal{T}_A$ and $\mathcal{T}_{all} - \mathcal{T}_A$. A further binary division on $\mathcal{T}_A$ creates the two subtrees: $\mathcal{T}_{AB}$ and $\mathcal{T}_A - \mathcal{T}_{AB}$. Similarly, a further division on $\mathcal{T}_{all} - \mathcal{T}_A$ creates the two subtrees: $\mathcal{T}_B$ and $\mathcal{T}_{all} - \mathcal{T}_A - \mathcal{T}_B$. Figure 8 shows the four subtrees.

  In the extreme case, binary division eventually creates a task for each node in the cube lattice, like in ASL. In PT, there is a parameter that controls when binary division stops. For the experimental results presented later, we used the parameter $32n$ to stop the division, once there are that many tasks (where n is the number of processors).
- Like in ASL, PT tries to exploit affinity scheduling. During processor assignment, the manager tries to assign to a worker processor a task that can take advantage of prefix affinity based on the root of the subtree. Note that in this case, subset affinity is not applicable. From this standpoint, PT is top-down. But interestingly, because each task is a subtree, the nodes within the subtree can be traversed/computed in a bottom-up fashion. In fact, PT calls BPP-BUC, which offers breadth-first writing, to complete the processing.

Figure 9 shows a skeleton of PT. The key step that requires elaboration is line 9, namely the exact definition of $\mathcal{T}$. In general, as shown in Figure 8, there are two types of subtrees handled by PT. The first type is a "full" subtree, which means that all the branches of the subtree are included. For example, $\mathcal{T}_{AB}$ is a full subtree. The second type is a "chopped" subtree, which means that some branches are not included. The subtrees $\mathcal{T}_A - \mathcal{T}_{AB}$ and $\mathcal{T}_{all} - \mathcal{T}_A - \mathcal{T}_B$ are examples. In line 11, depending on which type of a subtree is passed on to BPP-BUC, BPP-BUC may execute in a slightly different way. Specifically, for the loop shown on line 22 in Figure 6, if a full subtree is given, no change is

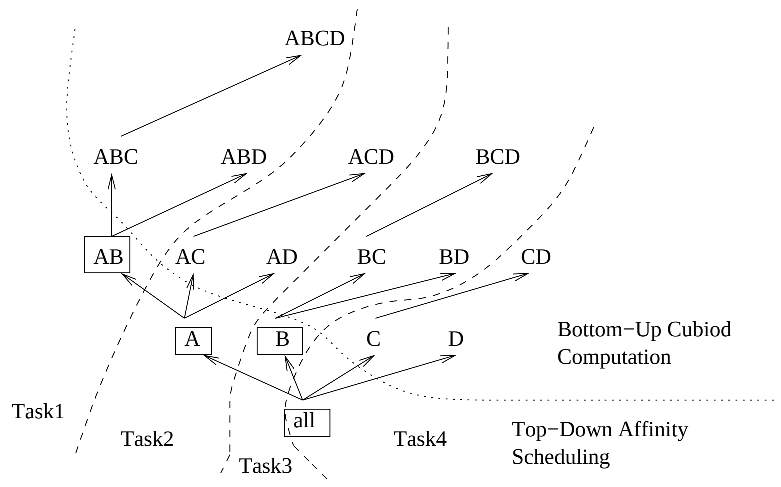


Figure 8: Binary Division of the Processing Tree into Four Tasks

```
1.  Algorithm PT
2.  INPUT: Dataset $R$ cube dimensions $\{A_1, \ldots, A_m\}$;
              minimum support $Spt$
3.  OUTPUT: The $2^m$ cuboids of the data cube
4.  PLAN:
5.      Task definition: a subtree created by repeated
            binary partitioning
6.      Processor assignment: a processor is assigned
            the next task based on prefix affinity
            on the root of the subtree
7.  CUBE COMPUTATION (for a processor):
8.      parallel do
9.          let the task be a subtree $\mathcal{T}$
10.         sort $R$ on the root of $\mathcal{T}$
                (exploiting prefix affinity if possible)
11.         call BPP-BUC($R, \mathcal{T}, Spt, \{\}$)
12.     end do
```

Figure 9: A Skeleton of PT

needed. Otherwise, the loop needs to skip over the chopped branches.

5. EXPERIMENTAL EVALUATION

The different algorithms were evaluated by varying each of the following parameters: number of processors, the size of the dataset, number of dimensions, minimum support, and the sparseness of the data (the product of the cardinalities of each dimension).

5.1 Experimental Setup and Parameters

The testbed for our work is a 16 processor cluster. As a heterogeneous platform, the cluster contains two types of PCs: eight 500MHz PIII processors with 256M of main memory and eight 266MHz PII processors with 128M of main memory. All machines have a 30Gbyte hard disk and were all on the same 100Mbit/sec Ethernet network. In the following, by "wall clock" time, we mean the maximum time taken by any processor. The time taken by a processor includes both CPU and I/O cost.

The cube computations were performed on different subsets of a weather dataset containing the weather conditions at various weather stations on land. The source of the data was the same as the datasets used by Ross and Srivastava [13], and Beyer and Ramakrishnan [3]. There are two datasets. Each has the same 20 dimensions; one has about 170K tuples and the other one has in excess of 1 million tuples. The data is very skewed on some of the dimensions. For example, partitioning the data on the 11^{th} dimension produces one part which is 40 times larger than the smallest part.

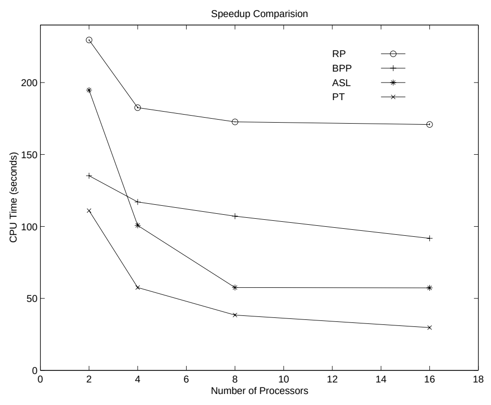
In order to compare the effect of varying the different parameters of the problem we used a fixed setting of the parameters and then varied each of them individually. The fixed setting, or **baseline configuration** for testing the algorithms, used:

- the eight 500MHz processors;
- 176,631 tuples (all from real data);
- 9 dimensions chosen arbitrarily (but with the product of the cardinalities roughly equal to 10^{13}); and
- with minimum support set at two.

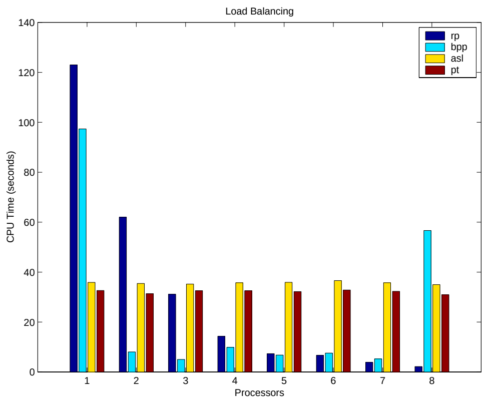
5.2 Varying the Number of Processors

Figure 10 shows the results of varying the number of processors while keeping the other parameters at their baseline values. Figure 10(a) shows a clear distinction between algorithms RP and BPP versus algorithms ASL and PT. While Figure 10(a) shows the wall clock time, Figure 10(b) shows the time taken by each processor for the case when there are 8 processors. It is clear that RP and BPP do not load balance as well as ASL and PT. The task assignment for RP is static and although the number of tasks is about equal, the amount of computation and I/O for the tasks differ significantly. For BPP, the dataset is partitioned statically across all of the nodes. Because the data is very skewed on some of the dimensions, the computation is not well balanced. The load imbalance is the main reason why RP and BPP do not scale well.

ASL and PT decrease the granularity of the tasks to a single cuboid in ASL and a small subtree in PT. The smaller granularity leads to better load balancing and the use of demand scheduling makes it easier to remain balanced even



(a) Runtime



(b) Load balance

Figure 10: Results of each algorithm using 1, 2, 4, 8, and 16 processors

when the dataset is very skewed. Both ASL and PT also use affinity scheduling to take advantage of share-sort to reduce the amount of computation. This leads to an interesting observation. Note that the speedup from 8 processors to 16 processors is below expectation. Initially, we thought that the extra 8 processors were the ones with the slower CPUs. However, when we did a direct comparison between the baseline configuration on the 8 faster CPUs versus the 8 slower CPUs, the difference was small. We also verified that the manager processor was not the bottleneck. Upon further investigation, we now have the following belief/conclusion. For both PT and ASL, the number of tasks increases as the number of processors increases which results in less opportunity for them to take advantage of affinity scheduling. For example, in ASL suppose tasks AB and AC have been assigned to different processors. Both processors compete for tasks A and "all" since both have a prefix affinity with AB and AC. For a larger number of processors, there is a greater chance of tasks being assigned to separate processors resulting in more competition and less opportunity to exploit prefix and subset affinity.

5.3 Varying the Problem Size

Figure 11(a) shows that for increasing problem size, both PT and ASL do significantly better than RP and BPP. Both PT and ASL appear to grow sublinearly as the number of tuples are increased. This is due to two factors. First, there is an overhead in creating the 2^9 cuboids which is independent of the amount of data. Second, doubling the number of tuples does not change the cardinality of the dimensions (except for the date field) and does not imply that there is twice the amount of I/O since more aggregation may take place.

It is possible to use more processors to solve a fixed problem faster or to solve a larger problem in the same amount of time. The results in Figure 11(a) show that PT and ASL scale well with problem size and indicates that these algorithms could be used, given sufficient memory and disk space, to solve larger problems on larger cluster machines.

5.4 Varying the Number of Dimensions

Figure 11(b) shows the effect of each algorithm on increasing the number of dimensions. Note the number of cuboids grows exponentially with the dimension size. For example, the dimension 13 cube has 8192 cuboids.

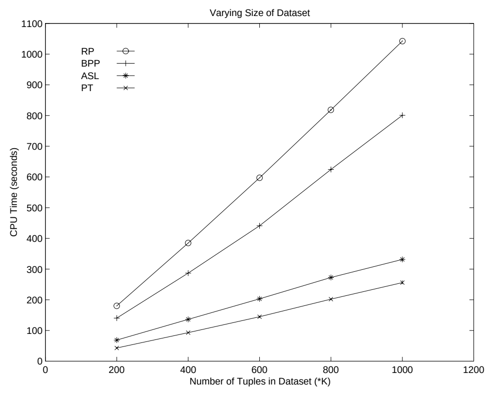
The relative performance for the four algorithms remains the same except for ASL where for 13 dimensions it is no longer better than BPP. The reason why ASL is affected more than the other algorithms is because of the comparison operation. The comparison operation used to search and insert cells into the skiplist becomes more costly as the length of the key increases. The length of the key grows linearly with the number of dimensions. This is a significant source of overhead for ASL. As well, for sparse data as the number of dimensions increases there is less aggregation and the size of many cuboids is close to that of the raw dataset.

Figure 11(b) also shows that when the number of dimensions is small, RP, ASL and PT give similar performance. Because the size of the output is small for a small number of dimensions, the simple RP algorithm, is able to do as well as the others.

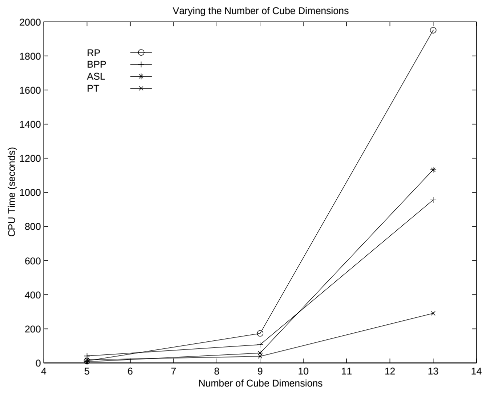
5.5 Varying the Minimum Support

Figure 12(a) shows the effect of increasing the minimum support. As the minimum support increases, there is more pruning and as a result less IO. The total output size for the algorithms given in Figure 12(a) starts at 469Mbyte for a support of 1, 86Mbyte for a support of 2, 27Mbytes for a support of 4, and 11Mbytes for a support of 8. After 8 there is very little additional pruning that occurs. Except between 1 and 2, the output size does not appear to have a large affect on overall performance. This is a surprise since we expected PT to do better as the support increases, since more pruning should have led to less computation. The relative flatness of the curve for PT may be due to the order of the dimensions. Or, it may be due to the fact that pruning occurs more towards the leaves, where it does not save as much in computation time.

5.5.1 Varying the Sparseness of the Dataset

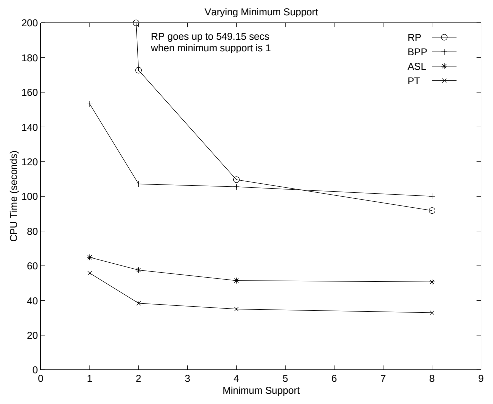


(a) datasets with 200K, 400K, 600K, 800K, and 1M tuples

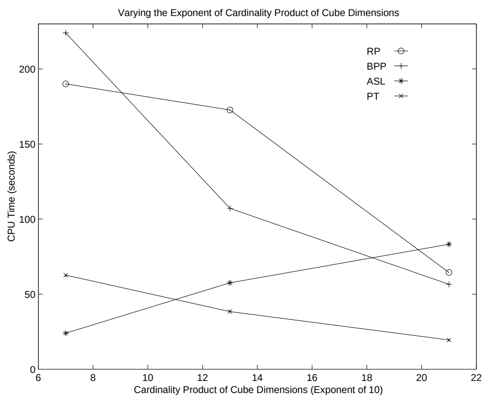


(b) varying dimensions: 5, 9, 13

Figure 11: Results for varying the dataset size and varying the number of dimensions



(a) minimum support: 1, 2, 4, 8



(b) varying sparseness

Figure 12: Results for varying the minimum support and the sparseness of the dataset

Situations	PT	ASL	RP
dense cubes		√	
small dimensionality (≤ 5)	√	√	√
high dimensionality	√		
otherwise	√√	√	
online support		√	

Figure 13: Recipe for selecting the best algorithm

Figure 12(b) shows the effect of sparseness of the dataset on the four algorithms. We consider a dataset to be sparse when the number of tuples is small relative to the product of the number of distinct attribute values for each dimension in the datacube. Since the number of tuples in the baseline configuration is fixed, we can vary the sparseness of the dataset by choosing smaller dimensions over larger cardinality dimensions. The three datasets chosen for Figure 12(b) consisted of the nine dimensions with the smallest cardinalities, nine dimensions with the largest cardinalities, and one in between. Note that even for the smallest of the three, there are still possibly about 10^7 total cells in the cube.

Unlike the other algorithms, ASL performs the best on dense datasets and is more adversely affected by spareness. ASL performs well for dense datasets because each cuboid contains relatively few cells, which makes searching or inserting into a skiplist relatively fast. The BUC-based algorithms have little opportunity to prune the data because of the density of the dataset. As a result, while traversing the lattice, the BUC-based algorithms need to sort almost the entire dataset for many of the cuboids. BPP does particularly poorly for cube dimensions with small cardinalities because BPP cannot partition the data very evenly, which leads to serious load imbalance. ASL does worst than the BUC-based algorithms when the product of the cardinalities is high partly because of the amount of pruning that occurs for the BUC-based algorithms, and partly because ASL has to maintain larger skiplists.

5.6 Varying the Order of the Dimensions

We also experimented with ordering the dimensions in the cube in ascending order, in terms of cardinality, and in descending order. For descending order, having the large cardinality dimensions first should lead to more pruning earlier in the computation for the BUC-based algorithms. In addition, having the larger cardinality dimensions first also should lead to better overall load balancing of the task assuming that the data is not overly skewed. For space reason, we omit the details here.

5.7 Summary: Our Recommended Recipe

The experimental results shown thus far is an exploration of the different parameters that affect overall performance. Upon careful examination of the results, we recommend the "recipe" shown in Figure 13 to select the best algorithm for various situations.

It is clear that ASL dominates all the other algorithms when the cube is dense, or when the total number of cells in the data cube is not too huge (e.g., $\leq 10^8$). For data cubes with a small number of dimensions (e.g., ≤ 5), essentially all algorithms behave quite the same. In this case, RP may have the slight edge that it is the simplest algorithm to implement. For all other situations except when the data cube has a large number of dimensions, PT and ASL are relatively close in performance with PT typically a constant factor faster than ASL. For cubes of high dimensionality, there is a significant difference between the two, and PT should be used. The last entry in Figure 13 concerns online support. This is the topic of the next section.

6. DISCUSSION: SUPPORT FOR TRULY ONLINE PROCESSING

Recall that what motivates the development of the various parallel algorithms studied here is the ideal of making OLAP "truly online". Algorithms aside, there are issues other than speedup and efficiency. In this discussion, we consider two important issues, and relate the issues to the algorithms presented here.

- The first issue is the support for sampling and progressive refinement. The online aggregation framework proposed by Hellerstein, Haas and Wang [10] uses sampling to trade-off response time for accuracy. The user is able to observe the progress of a query and dynamically direct or redirect the computation.

  In the case of a iceberg-cube query, the user would see a rough initial cuboid which would become more accurate as more and more tuples have been processed. Among all the algorithms presented here, ASL can easily be extended to fit into this framework. This is because it can construct cuboids incrementally by inserting tuples into the skiplist. For ASL, other than coordinating the access to the structure, each tuple can be handled independently. We can sample the data in parallel and as more data becomes available, the new tuples can be distributed to the different processors to be inserted into the appropriate cuboids. The same procedure is more difficult for PT because the underlying data structure used by PT is an array. It would be necessary to compute the cuboid on the new sampled data and then merge the cuboids corresponding to the same set of attributes. The online advantages of ASL over PT was one of the main motivations for its development.

- So far we have assumed that no part of the entire data cube has been pre-computed. Our experimental results show that in many cases, our parallel algorithms can do well in computing the entire iceberg-cube query from scratch (e.g., ≤ 100 seconds). Clearly, for truly online processing, selective materialization may help significantly. The complication is the existence of the support threshold, or in general the constraints. Specifically, it is no longer possible to compute a cuboid from a pre-computed cuboid when the minimum support of the online query is lower than that of the pre-computed one.

  As an exercise, we compared two different plans for answering online queries using ASL. The first plan is to simply re-compute the query based on the specified minimum support. Let say that the minimum support is 2. According to Figure 12(a), ASL would take around 60 seconds.

The second plan consists of a pre-computation stage and an online stage. In the pre-computation stage, ASL computes only leafs of the traversal tree using the smallest minimum support (i.e., 1). In the online stage, ASL uses top-down aggregation and returns those cells satisfying the new specified support. For the second stage, ASL can return almost immediately; and interestingly, even for the pre-computation, it only took 50 seconds for the same example. (The value of 50 seconds was obtained from our additional experiment, not directly from Figure 12(a). The values there include the total time for the nodes in the tree, not just the leaves.) This suggests that even simple selective materialization can help. It is a topic of future work to develop more intelligent materialization strategies.

7. CONCLUSION

In this paper we have developed a collection of parallel algorithms directed towards online and offline creation of datacubes to support iceberg queries. The four algorithm, RP, BPP, PT and ASL, are novel. They were experimentally evaluated over a variety of parameters to determine the best situation to use them. RP has the advantage that it is simple to implement. However, except for cubes with low dimensionality, RP is dominated by the other algorithms. BPP is also dominated; but BPP shows that breadth-first writing is a useful optimization. As an extension of BPP, PT is the algorithm of choice for most situations. There are, however, two exceptional situations when ASL is the most recommended. ASL is the most efficient for dense cubes, and readily supports sampling and progressive refinement.

In future work, we would investigate how the lessons we have learned for parallel iceberg query computation can be applied to other tasks in OLAP computation and data mining. These include (constrained) frequent set queries [12, 18], and OLAP computation taking into account correlation between attributes.

8. REFERENCES

[1] R. Agrawal, S. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. 1996 VLDB*, pp. 506–521.

[2] E. Baralis, S. Paraboschi and E. Teniente. Materialized view selection in a multidimensional database. In *Proc. 1997 VLDB*, pp. 98–112.

[3] K. Beyer and R. Ramakrishnan. Bottom-Up Computation of Sparse and Iceberg CUBEs. In *Proc. 1999 ACM SIGMOD*, pp 359–370.

[4] M. Eberl, W. Karl, C. Trinitis, and A. Blaszczyk. Parallel Computing on PC Clusters – An Alternative to Supercomputers for Industrial Applications. In *Proc. 6th European Parallel Virtual Machine/Message Passing Interface Conference*, LNCS vol. 1697, pp. 493–498, 1999.

[5] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani and J. Ullman. Computing iceberg queries efficiently. In *Proc. 1998 VLDB*, pp. 299–310.

[6] S. Goil and A. Choudhary. High Performance OLAP and Data Mining on Parallel Computers. In *The Journal of Data Mining and Knowledge Discovery*, 1, 4, pp. 391–418, 1997.

[7] J. Gray, A. Bosworth, A. Layman and H. Pirahesh. Datacube: A relational aggregation operator generalizing group-by, cross-tab and sub-totals. In *Proc. 1996 ICDE*, pp. 152–159.

[8] H. Gupta, V. Harinarayan, A. Rajaraman and J. Ullman. Index selction for OLAP. In *Proc. 1997 ICDE*, pp. 208–219.

[9] V. Harinarayan, A. Rajaraman and J. Ullman. Implementing data cubes efficiently. In *Proc. 1996 ACM SIGMOD*, pp. 205–216.

[10] J. Hellerstein, J. Haas and H. Wang. Online Aggregation. In *Proc. 1997 SIGMOD*, pp. 171–182.

[11] M. Kamber, J. Han and J. Chiang. Metarule-guided mining of multi-dimensional association rules using data cubes. In *Proc. 1997 KDD*, pp. 207–210.

[12] R.T. Ng, L.V.S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. 1998 SIGMOD*, pp. 13–24.

[13] K. Ross and D. Srivastava. Fast Computation of Sparse Datacubes. In *Proc. 1997 VLDB*, pp. 116-125.

[14] S. Sarawagi. Explaining differences in multidimensional aggregates. In *Proc. 1999 VLDB*, pp. 42–53.

[15] A. Shukla, P. Deshpande and J. Naughton. Materialized view selection for multidimensional datasets. In *Proc. 1998 VLDB*, pp 488-499.

[16] A. Srivastava, E. Han, V. Kumar and V. Singh. Parallel formulations of decision-tree classification algorithm. In *The Journal of Data Mining and Knowledge Discovery*, 3, 3, pp. 237–262, 1999.

[17] M. Tamura and M. Kitsuregawa. Dynamic Load Balance for Parallel Association Rule Mining on Heterogeneous PC Cluster System. In *Proc. 1999 VLDB*, pp. 162–173.

[18] M. Zaki. Parallel and distributed association mining: a survey. In *IEEE Concurrency*, 7, 4, pp. 14–25, 1999.

[19] YiHong Zhao, Prasad Deshpande, and Jeffrey F. Naughton An Array-based algorithm for simultaneous Multidimensional aggregates. SIGMOD Conference 1997, pp. 159-170