Iceberg-cube Computation with PC Cluster

by

Yu Yin

B.Sc., Jilin University, 1993

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

The University of British Columbia

April 2001

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date Oct 10, 2001

Abstract

Iceberg queries constitute one of the most important classes of queries for OLAP applications. This thesis investigates using low cost PC clusters to parallelize the computation of iceberg queries. We concentrate on techniques for querying large, high-dimensional data sets. Our exploration of an algorithmic space considers trade-offs between parallelism, compuation, memory and I/O. The main contribution of this thesis is the development and evaluation of various novel, parallel algorithms for CUBE computation and online aggregation. These include the following: one, the CUBE Algorithm RP, which is a straightforward parallel version of BUC[BR99]; two, the CUBE Algorithm BPP, which attempts to reduce I/O by outputting results in a more efficient way; three, the CUBE Algorithms ASL and AHT, which maintain cells in a cuboid in a skip list and a hash table respectively, designed to put the utmost priority on load balancing; four, alternatively, the CUBE Algorithm PT load-balances by using binary partitioning to divide the cube lattice as evenly as possible; and five, the online aggregating algorithm POL, based on ASL and sampling technique, which gives back instant response and further progressive refinement.

We present a thorough performance evaluation of all these algorithms in a variety of parameters, including the dimensionality and the sparseness the cube, the selectivity of the constraints, the number of processors, and the size of the data set. The key to understanding the CUBE algorithms is in that one-algorithm-does-not-fit-all. We recommend a "recipe" which uses PT as the default algorithm, but may also deploy ASL or AHT in appropritate circumstances. The online aggregation algorithm, POL, is especially suitable for computing a high dimensional query over a large data set with a cluster of machines connected by high speed networks.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Alan Wagner, and professor Dr. Raymond Ng, for helping me in carrying out this research project and for reading the manuscript of my thesis and offering their valuable comments. I also would like to thank Kirsty Barclay for reading the manuscript of my thesis and providing me with her helpful comments.

YU YIN

The University of British Columbia
April 2001

Chapter 1

Introduction

As computing and the Internet advance, we see a massive increase in the raw data available to institutions, corporations, and individuals. For example, large numbers of radiological images have been generated in hospitals and immense product and customer databases have been accumulated [1]. Extracting meaningful patterns and rules from such large data sets is therefore becoming more and more important. In this context, On-line Analytical Processing (OLAP) has emerged as a powerful tool for data analysis. In decision support systems, OLAP enables analysts and managers to obtain insight into data. By interactively posing complex queries, they can extract different views of data.

In many OLAP applications, aggregation queries constitute a large percentage of the computation. Many of these queries are only concerned with finding aggregate values above some specified threshold. We call this kind of query "iceberg queries". Query results consisting of above-threshold aggregate values are typically small compared to the total input data (the iceberg).

Through Structured Query Language (SQL) aggregate functions and the GROUP BY operator, OLAP applications can easily produce aggregates for one group-by, however, most applications need aggregates for a set of group-bys in order to gain more insight into the data. This necessitates generalization of GROUP BY

operator. The CUBE operator, defined by Gray et al [8], provides this generalization. It computes aggregation for every possible combination of a set of specified attributes. For instance, if CUBE operator is applied on 2 attributes, A and B, then the aggregates from GROUP BY on *all*, GROUP BY on A, GROUP BY on B and GROUP BY on AB will be returned together. The computation introduced by the CUBE operator is huge, because for d specified attributes, 2^d GROUP BYs are computed. Furthermore, In each cuboid, there are also numerous cells, or partitions, computed. In CUBE terminology, output for an n-attributes GROUP BY is called an n-dimensional cuboid, also called an n-dimentional group-by. When the CUBE operator is employed to answer a set of iceberg queries, we call it an "iceberg-cube".

In this thesis, we investigate the algorithms for answering iceberg queries, especially for iceberg-cube computation. Recently, several algorithms have been proposed, including the PipeSort and the PipeHash algorithms proposed by Agrawal et al. [2], the Overlap algorithm proposed by Naughton et al [21], the PartitionedCube algorithm proposed by K. Ross and D. Srivastava [14] and the Bottom-Up algorithm (BUC) proposed by Beyer and Ramakrishnan [4]. All these algorithms except BUC are general CUBE computation algorithms, in the regard that they do not specifically target iceberg-cube computation. They proceed in a top-down fashion, that is, computing from more dimensional group-bys to less dimensional group-bys. Many of them try to utilize previous sorting in the top-down traversal. BUC provides another efficient solution, specifically for threshold-set iceberg queries. It proceeds in a bottom-up fashion, trying to prune tuples which do not satisfy threshold as early as possible to reduce computation. We will discuss these two kinds of algorithms in more detail in Chapter 2 .

We based our work on the algorithms mentioned above, however, we were interested in providing parallel solutions. The previous CUBE algorithms mainly proposed for running on stand alone machines were developed to execute on a single processor, so-called sequential algorithms. In this thesis, we propose several par-

allel algorithms for answering iceberg queries, and promote the benefits of using distributed computing platforms to solve problems. Our underlying architecture is a dedicated cluster of PCs. With elegant parallel algorithms, these machines have the potential to achieve the performance of massive parallel machines at a much lower cost. We focused our work 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.

To improve the response time of iceberg queries, two different solutions are explored: precomputing and online querying.

Precomputation is a common technique used by many OLAP applications. Usually, precomputation computes a CUBE operator, extracting multiple aggregates and saving the results on disks. It supports instant response if the precomputed results match a user's queries. Towards efficient iceberg-cube precomputation with PC clusters, this thesis explores different trade-offs between parallelism, computation and I/O. Assuming input data sets fit in main memory on each machine of the cluster, we developed several novel, parallel algorithms for iceberg-cube computation and give a comprehensive evaluation in this thesis. Here is a summery of the parallel algorithms:

- Algorithm RP (*Replicated Parallel* BUC), is a straightforward parallel version of BUC. It is simple and introduces little overhead above its sequential version. However, algorithm RP is poor in distributing tasks and balancing workload. In an attempt to achieve better load-balancing, algorithm BPP (*Breadth-first writing, Partitioned, Parallel* BUC), was developed. BPP differs from RP in two key ways. First, the dataset is range-partitioned and distributed other than replicated in RP; second, the output of cuboids is done in a breadth-first fashion, as opposed to the depth-first writing in RP and BUC. Table 1.1 summarizes the key features of the algorithms.

- Though BPP is better than RP concerning load-balancing, this improvement

Algorithms	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

Table 1.1: Key Features of the Algorithms

is limited when the raw data set skews on some attributes. This is primarily because the task granularity of RP and BPP is relatively large and uneven. To consider load balancing as the utmost priority, algorithm ASL (*Affinity SkipList*) is developed. In ASL each cuboid is treated as a task. ASL uses an affinity task scheduling strategy to seek the relationship among tasks assigned to the same processor and maximize sort sharing among them. Thus ASL resembles to the top-down algorithms. ASL is also unique in the regard that it maintains the cells of a cuboid in a different data structure, namely a skip list.

- Algorithm PT (*Partitioned Tree*) is a hybrid algorithm, combining both the idea of pruning from BUC and affinity scheduling from ASL. It processes tasks of slightly coarser granularity. The idea is to use binary partitioning to divide the cuboids into tasks as evenly as possible, in order to make the load well-balanced. The computation in each task proceeds in bottom-up fashion, however, the task assignment is processed by affinity scheduling in a top-down fashion.

- Two other algorithms based on a hash tree and a hash table were also developed. The implementation based on a hash tree used up memory too rapidly that it fails to process large data set. The hash table based algorithm was implemented much like ASL, in terms of task definition and scheduling. However, its performance is no better than ASL in most cases.

Two questions natually arise at this point: one, which algorithm is the best; and two, do we really need to know about all these algorithms? In considering the first question, we present a thorough performance evaluation of all these algorithms on a variety of parameters. The parameters include the dimensionality, the sparseness of the group-bys, the selectivity of the constraints, the number of processors, and the sizes of the data sets. 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-fits-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.

Putting parallel iceberg-cube algorithmic development and evaluation aside temporarily, we next consider the concept of "truly online". Precomputation can answer users' queries instantly if the query pattern can be predicted. However, if the threshold set by online queries differs from what the precomputation assumed, precomputed cuboids can no longer be used to answer those queries. Therefore, those queries have to be computed online. We posit a scenario that the input raw data set no longer fits in main memory. Only with this precondition will the query computation be large enough to necessitate applying parallelism. In the online aggregation framework proposed and studied by Hellerstein, Haas and Wang, an online query algorithm based on ASL was developed. Using the sampling technique, a user's online query can be responded to instantly. And with more and more data processed, the answer becomes more and more refined and accurate.

Integrating CUBE precomputation and online querying computation together, this thesis gives a relative complete solution for the special problem domain: iceberg query computation.

The outline of the thesis is as follows. Chapter 2 reviews key concepts and the main sequential algorithms for iceberg-cube computation. Chapter 3 introduces the various parallel algorithms we developed. Chapter 4 presents a comprehensive

experimental evaluation of these algorithms, and concludes with a recipe for picking the best algorithms under various circumstances. Chapter 5 discusses online processing. Finally, a conclusion is given in Chapter 6.

Chapter 2

Review

The background material necessary for understanding the parallel algorithms to be introduced in Chapter 3 is presented in this chapter. We first discuss iceberg query, then the CUBE operator. A special CUBE operator, iceberg-cube, is introduced seperately. The last part of this chapter, Section 2.4 presents some sequential algorithms for CUBE and iceberg-cube computation.

2.1 Iceberg Query

An iceberg query is much like a regular aggregate query, except that it eliminates aggregate values that fall below some specified threshold after it performs an aggregate function over an attribute or a set of attributes. The prototypical iceberg query considered in this thesis is as follows for a relation $R(target1, target2, \ldots, targetk, rest, aggregateField)$ and a threshold T.

SELECT	$target1, target2, \ldots, targetk$, SUM($aggregateField$)
FROM	R
GROUP BY	$target1, target2, \ldots, targetk$
HAVING	count($rest$) $\geq T$

If the above iceberg query is applied to the relation R in Table 2.1, with T

target1	target2	rest	aggregate Field
Item	Location	Customer	Sales
Sony 25" TV	Seattle	Joe	700
JVC 21" TV	Vancouver	Fred	400
Sony 25" TV	Seattle	sally	700
JVC 21" TV	LA	sally	400
Sony 25" TV	Seattle	bob	700
Panasonic Hi-Fi VCR	Vancouver	tom	250

Table 2.1: Example relation R

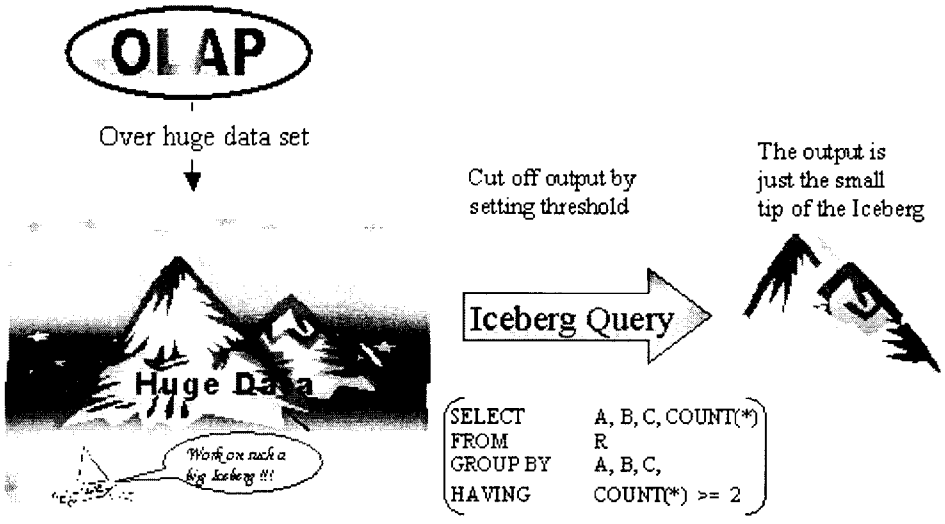


Figure 2.1: Iceberg Query

$= 2$ and $k = 2$, the result would be the tuple <Sony 25" TV, Seattle, 2100>. We notice that relation R and the number of unique *target* values are typically huge (the iceberg), and the answer, that is, the number of frequently occurring targets, is very small (the tip of the iceberg). This situation is pictured in Figure 2.1.

An iceberg query becomes especially important when the amount of input data is tremendous, since data analysts or managers can not possibly go through all the detailed information within a huge data set. Usually, they only note frequently occurring behaviors, which are typically more important than unusual occurrences.

In realistic data analysis, data analysts often execute multiple iceberg queries, which GROUP BY on different number of dimensions. For example, they may want to know more detailed information if the previous query returns too few results. Afterward they might like to "drill-down" by GROUP BY on more attributes. On the other hand, if the previous query gives back too detailed and too much information, they may like to "roll-up" by giving less GROUP BY attributes in the upcoming query. A Generated report containing results from all those queries can be formulated in standard SQL, but its representation is inconvenient. As well as drill-down and roll-up, some other frequently used queries including histogram and cross-tab are also difficult to represent in standard SQL [19].

2.2 CUBE Operator

To exceed the limitation posed by the standard SQL, as mentioned in Section 2.1, the CUBE operator was introduced in [8] by J. Gray et al. It generalizes the standard GROUP BY operator to compute aggregates for every combination of GROUP BY attributes. For instance, consider the following relation SALES(Model, Year, Color, Sales), shown in the lefthand table in Figure 2.2. When CUBE is on R with GROUP BY attributes Model, Year and Color, aggregate on attribute Sales (SUM in this case), the result returned will contain the sum of Sales for the entire relation (i.e. no GROUP BY), for each item: (Model), (Year), (Color), for each pair: (Model, Year), (Model, Color), and (Year, Color), and finally for each (Model, Year, Color). The result is shown in the righthand table in Figure 2.2. Figure 2.3 shows the CUBE in a multi-dimensional array format.

In OLAP terminology, the GROUP BY attributes are called "dimensions", the attributes that are aggregated are called "measures", and one particular GROUP BY, (e.g., (Model, Year)), in a CUBE computation is called a "cuboid" or simply a "group-by".

Three types of aggregate functions are identified in [8]. Consider aggregating

SALES			
Model	Year	Color	Sales
Chevy	1990	red	5
Chevy	1990	white	87
Chevy	1990	blue	62
Chevy	1991	red	54
Chevy	1991	white	95
Chevy	1991	blue	49
Chevy	1992	red	31
Chevy	1992	white	54
Chevy	1992	blue	71
Ford	1990	red	64
Ford	1990	white	62
Ford	1990	blue	63
Ford	1991	red	52
Ford	1991	white	9
Ford	1991	blue	55
Ford	1992	red	27
Ford	1992	white	62
Ford	1992	blue	39

Relation SALES

```
SELECT   Model, Year, Color
         SUM(Sales)
FROM     SALES
CUBE BY Model, Year, Color
```

SALES			
Model	Year	Color	Sales
ALL	ALL	ALL	942
Chevy	ALL	ALL	510
Ford	ALL	ALL	432
ALL	1990	ALL	343
ALL	1991	ALL	314
ALL	1992	ALL	285
ALL	ALL	red	165
ALL	ALL	white	273
ALL	ALL	blue	339
Chevy	1990	ALL	154
Chevy	1991	ALL	199
Chevy	1992	ALL	157
Ford	1990	ALL	189
Ford	1991	ALL	116
Ford	1992	ALL	128
Chevy	ALL	red	91
Chevy	ALL	white	236
Chevy	ALL	blue	183
Ford	ALL	red	144
Ford	ALL	white	133
Ford	ALL	blue	156
ALL	1990	red	69
ALL	1990	white	149
ALL	1990	blue	125
ALL	1991	red	107
ALL	1991	white	104
ALL	1991	blue	104
ALL	1992	red	59
ALL	1992	white	116
ALL	1992	blue	110
All Tuples in Relation SALES			

CUBE of SALES on attributes Model, Year and Color, where aggregate attribute is Sales.

Figure 2.2: CUBE Operation on Relation SALES [8]

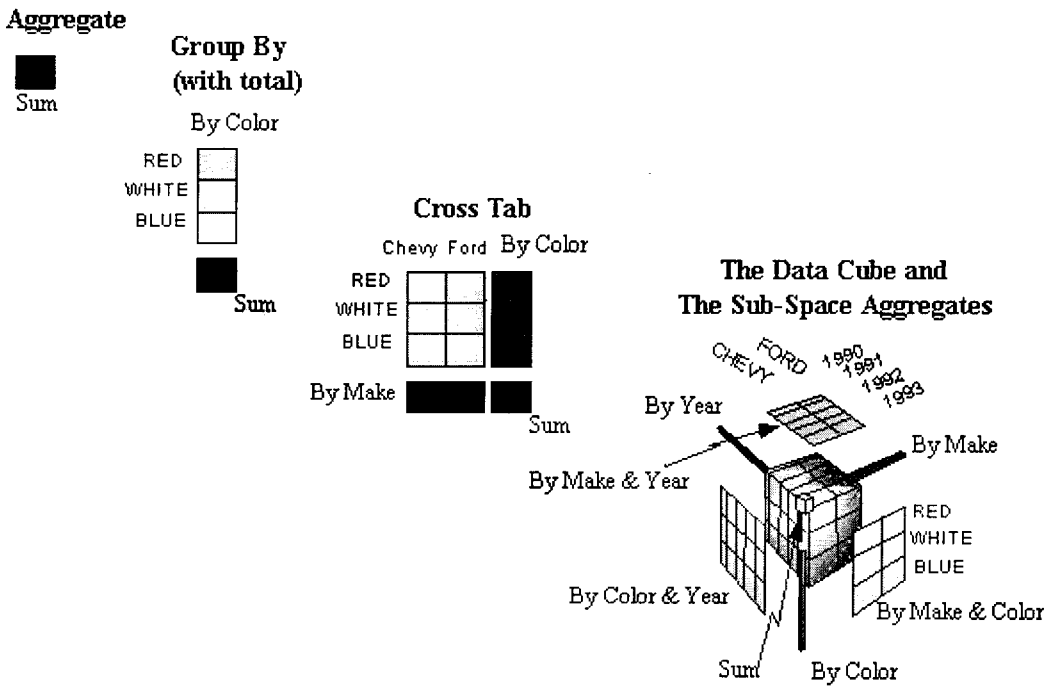


Figure 2.3: Cube in Multi-dimensional Array Format [8]

a set of tuples T. Let $\{S_i \mid i = 1 \ldots n\}$ be any complete set of disjointed subsets of T such that $\bigcup_i S_i = T$ and $\bigcap_i S_i = \{\}$.

- An aggregate function F is *distributive* if there is a function G such that $F(T) = G(\{F(S_i) \mid i = 1 \ldots n\})$. SUM, MIN and MAX are distributive with $G = F$. COUNT is distributive with $G = SUM$.

- An aggregate function F is *algebraic* if there is an M-tuple valued function G and a function H such that $F(T) = H(\{G(S_i) \mid i = 1 \ldots n\})$, and M is constant regardless of $|T|$ and n. All distributive functions are algebraic, as are Average, standard deviation, MaxN, and MinN. For Average, G produces the sum and count, and H divides the result.

- An aggregate function F is *holistic* if it is not algebraic. For example, Median and Rank are holistic.

2.3 Iceberg-cube Computation

The basic CUBE problem is to compute all aggregates as efficiently as possible. Its chief difficulty is that the CUBE computation is exponential with the number of dimensions: for d dimensions, 2^d group-bys are computed. The size of each group-by (cuboid) depends upon the cardinalities of its dimensions, possibly the product of the GROUP BY attributes' cardinalities. When the product of the cardinalities for a group-by is large relative to the number of the cells (partitions) that actually appear in the cuboid, we say the group-by is "sparse". When the number of sparse group-bys is large relative to the number of total number of group-bys, we say the CUBE is sparse. As is well-recognized, given the large result size of the entire CUBE, especially on sparse data set, it is important to identify subsets of interest.

Deriving from this background, the concept of an "iceberg-cube" was introduced in [4, 12].

the iceberg-cube was described as a variant of the CUBE problem, which allows us to selectively compute cells that satisfy a user-specified aggregate condition. It is essentially a CUBE for iceberg queries. For example, an iceberg-cube is easily expressed in SQL with the CUBE BY clause:

```
SELECT    A, B, C, SUM(X)
FROM      R
CUBE BY   A, B, C
HAVING    COUNT(*) ≥ N
```

where N is a count condition, called "minimum support" of a cell, or "minsup" for short. In this thesis, we only discuss this count condition; other aggregate conditions can be handled as well [4].

2.4 Sequential CUBE Algorithms

All CUBE algorithms uses a lattice view for discussion. Figure 2.4(a) depicts a sample lattice where A, B, C and D are dimensions. Nodes in the lattice represent

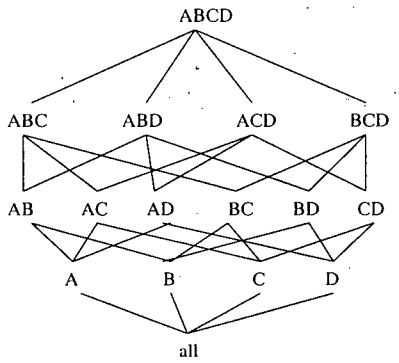
group-bys (cuboids). The group-bys are labeled according to their GROUP BY attributes. The edges in the lattice show potential computing paths. All of the CUBE algorithms in fact convert this lattice into a directed processing tree. Each node in a processing tree therefore has no more than one parent, because it is computed only once from its parent or from the raw data set.

CUBE algorithms are classified into two categories according to their computation fashion. 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 2.4(a), a top-down approach computes from ABCD, to ABC, to AB and eventually to A; a bottom-up approach goes in the opposite direction. Figure 2.4(b) gives a sample processing tree of top-down algorithm. The processing tree of bottom-up algorithm is illustrated in Figure 2.4(c).

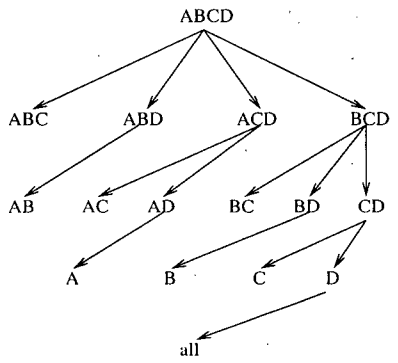
In the following, we will discuss some significant sequential CUBE algorithms proposed. CUBE algorithms can be viewed as having two stages: the planning stage and the execution stage. In the planning stage, the algorithms decide how to convert the lattice into a processing tree; in the execution stage, the algorithm computes cuboids.

2.4.1 Top-down CUBE algorithms

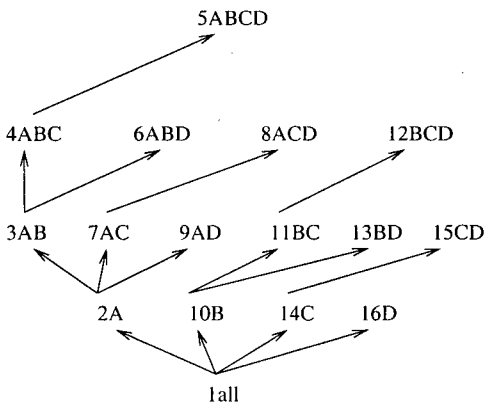
CUBE algorithms always try to discover and take advantage of commonality between a node and its parent in the lattice view. For many top-down algorithms, they recognize that group-bys with common attributes can share, sorts, or partial sorts, and utilize those sharings. Taking the processing tree shown in Figure 2.4(b) as an example, AD represents the cuboid GROUP BY on A and D. If the data set has been sorted with respect to A and D in order to compute AD, then for computing cuboid A, the data set does not have to be re-sorted. We can simply accumulate the sums for each of the values in A. Apparently, cuboid A and AD share sort on



(a) 4-Dimension Lattice



(b) Sample Processing Tree
of Top-Down Algorithms



(c) Processing Tree of Bottom-Up Algorithms

Figure 2.4: Lattice and Processing Trees for CUBE Computation [4]

attribute A.

Besides sort sharing, there are some other commonalities which were exploited by top-down algorithms. Some of these, specified as optimization techniques, are listed by Sarawagi [2]:

- *Smallest-parent*: This aims at computing a group-by from the smallest previously computed group-by. For example, we can compute group-by AB from group-by ABC and ABD. However, among the two potential parents, only the one with smallest size will be selected, because computing from the small parent will lead to lower cost.

- *Cache-results*: This technique tries to compute a group-by when its parent is still in memory, hence, reducing disk I/O.

- *Amortize scans*: This technique also aims at reducing disk I/O by amortizing disk reads by computing as many group-bys as possible together in memory. For instance, during scanning group-by ABCD, we can compute group-bys ABC, ACD, ABD and BCD at the same time.

- *Share-sorts*: Sort-based algorithms use this technique to share sorting cost among multiple group-bys.

- *Share-partitions*: This is specific to the hash-based algorithm. When a hash table can not fit in the memory, data will be partitioned into chunks which do fit in memory. Once a chunk is read in, multiple group-bys will be computed in order to share the partitioning costs.

In the following, we will discuss several sequential top-down algorithms.

PipeSort, PipeHash and Overlap

PipeSort and PipeHash algorithms are among the first algorithms for efficient CUBE computation. They were proposed by Sarawagi et al. in [2]. Both assume the cost

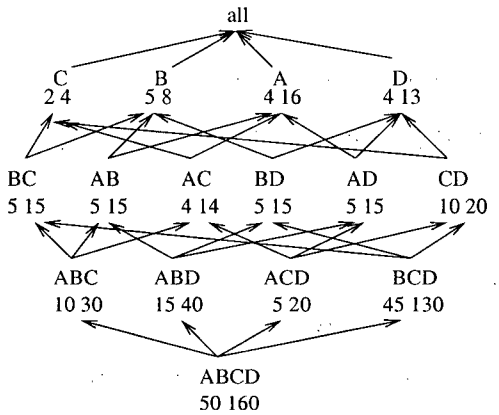
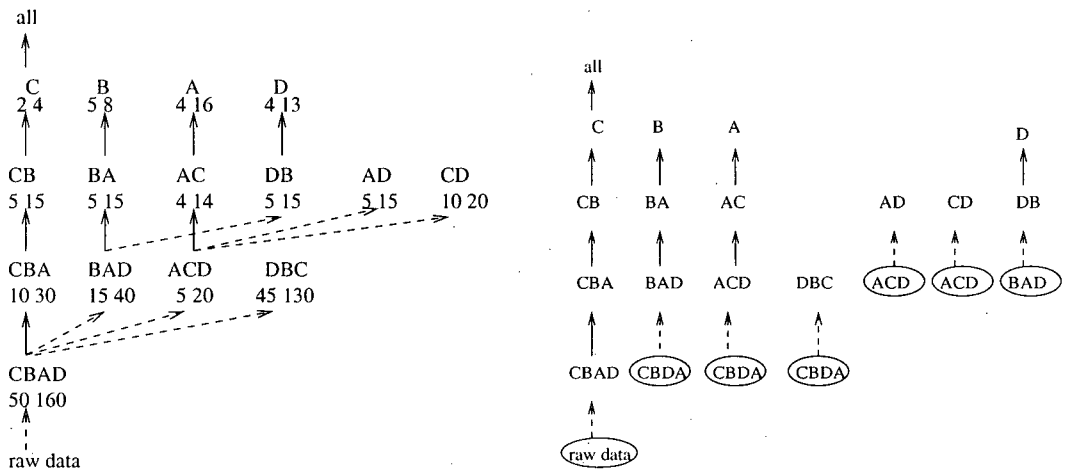


Figure 2.5: An Example of 4-Dimensional Lattice for Algorithm PipeSort [2]

of each node in a lattice proportional to the product of the cardinalities of GROUP BY attributes and try to compute each cuboid from a parent having the smallest cost. However, The data structures of the two algorithms are different: PipeSort uses array and sorting is done prior to aggregation; PipeHash uses hash tables. Furthermore, PipeSort considers share-sorts optimization, trying to minimize the number of sorts, whereas PipeHash focuses on share-partitions optimization.

PipeSort distinguishes between two different costs attached to each node X in the lattice view: cost A(X) and cost S(X). A(X) is induced when one child of X is created through aggregating without any sort on X. Actually only one child of X can be computed with cost A(X). For instance, for cuboid ABCD, only its child, cuboid ABC, can be computed without any sort on ABCD. For other children, if they are computed from ABCD, cost S(ABCD) is induced becasue resort on ABCD is necessary. In this way, the sorting cost is counted by PipeSort. Assuredly, cost S(X) is always greater than or equal to A(X).

In the planning stage, a processing tree with a minimum total cost, taking both A(X) and S(X) into account, is computed in a level-by-level manner, where level N contains all N-dimensinal cuboids. When computing on a level, the algorithm determines what edges between the nodes in this level and the next level in



(a) Minimum Cost Sort Plan

(b) Pipelines Executed

Figure 2.6: An Example of Plan and Pipelines for Algorithm PipeSort [2]

the lattice should be left in the final minimum cost tree. Since each edge has a cost attached, either A(X) or S(X), the problem is converted into finding the minimum cost matching in a bipartite graph. Given the lattice shown in Figure 2.5, the final minimum cost plan becomes that shown in Figure 2.6(a). The pair of numbers underneath each group-by in the figure denote the A(X) and S(X) costs. The detailed plan computation is elaborated in [2].

After a plan is created, in the execution stage each path is computed in a pipeline manner. Figure 2.6(b) shows the pipelines' execution for the generated plan in Figure 2.6(a). The head of each pipeline implies a re-sort, from its parent in the processing tree.

Like PipeSort, PipeHash aims at computing cuboids from their smallest parents. Since PipeHash takes hash tables as its data structure, no sorting is required. Therefore, each node in the lattice has only one cost, which is similar to A(X) in PipeSort. In the planning stage of PipeHash a minimum spanning tree(MST) is computed based on the singular cost of each node. Figure 2.7(a) gives an example

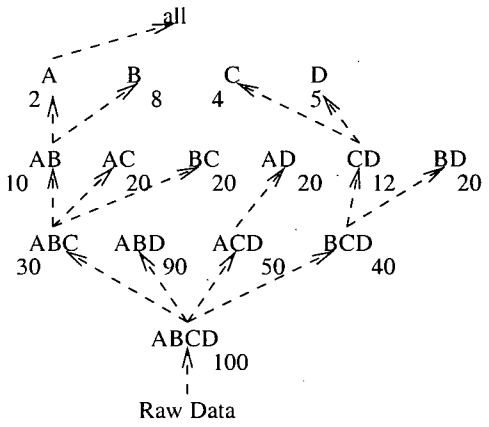
of an MST.

Besides smallest-parent optimization, PipeHash also explores share-partitions and amortized-scans optimizations. It computes as many cuboids as possible if their parents are in memory. If the main memory is big enough to hold hash tables for all cuboids, PipeHash can finish the cube computation in one data scan without any sorting. If no enough memory is available, PipeHash partitions data on some selected attribute, then processes each partition independently.

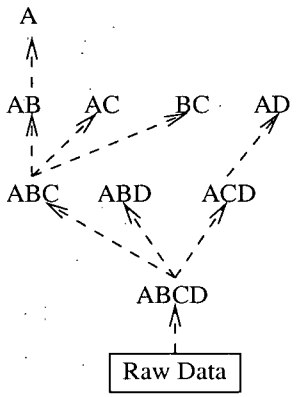
Although data partitioning solve the memory problem, the partitioning attribute limits computations to only include group-bys with that particular attribute. For example, from the MST in Figure 2.7(a), we compute a CUBE on dimensions A, B, C and D. If we partition on A, then the partitions are only used to produce cuboids containing dimension A, including ABCD, ABC, ABD, AB, AC, AD and A. Other cuboids will be computed afterward from cuboids with attribute A. Ideally, they can fit in memory and no further partitioning is necessary. This makes MST divide into subtrees, as shown in figure 2.7(b) and (c). By processing as large a subtree(or a set of subtrees) of the MST as possible in memory, computing all nodes in it (or them) simultaneously, PipeHash favors optimizations cache-results and amortize-scans.

Sarawagi compared PipeSort and PipeHash in [2]. PipeHash suffers two apparent problems, requiring re-hash for every group-by and requiring a significant amount of memory. This makes it can only outperform PipeSort as the data is dense. However, in this thesis, the problem domain is iceberg-cube computation in which data is supposed to be highly sparse; therefore, hash-based algorithms are not our major concern. However, we did implement some hash-based algorithms and they will be discussed in Chapter 3 .

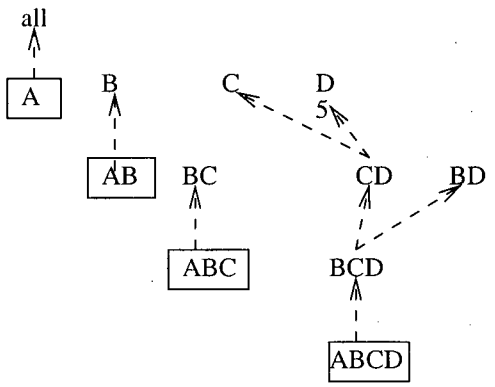
Overlap, proposed in [21], as well as PipeSort, considers sorting cost, but it deals with it in a different way. It tries to overlap as much sorting as possible by computing group-by from a parent with the maximum sort-order overlap. The



(a) Minimum Spanning Tree



(b) Subtree: Partitioned on A



(c) Remaining Subtrees

Figure 2.7: PipeHash on a Four Attribute Group-by [2]

algorithm recognizes that if a group-by shares a prefix of GROUP BY attributes with its parent, then the parent consists of a number of partitions, one for each value of the prefix. For example, since cuboid ABC and cuboid AC share a GROUP BY prefix A, the ABC group-by has |A| partitions that can be sorted independently on C to produce the AC sort order, where $|A|$ is the number of values for attribute A.

Overlap always selects a parent for a cuboid which shares the longest GROUP BY prefix with that cuboid. Then the size of partition is minimized. If several potential parents of a group-by share the same length of prefix with it, and then the smallest one will be picked as the final parent. Overlap chooses a sort order for the root of the processing tree, then all subsequent sorts are some suffix of this order.

The planning stage will build a tree like that shown in Figure 2.4(b). Once this processing tree is formed, Overlap tries to fit as many partitions in memory as possible. If a partition of a group-by can fit in the main memory, then a subtree of the processing tree rooted by that group-by will be computed in a pipeline manner when the partition is scanned in. This is expected to save much I/O costs for writing intermediate results.

The experiments show that Overlap performs consistently better than PipeSort and PipeHash. However, [14] argues that Overlap on sparse CUBES still produces a large amount of I/O by sorting intermediate results.

PartitionedCube and MemoryCube

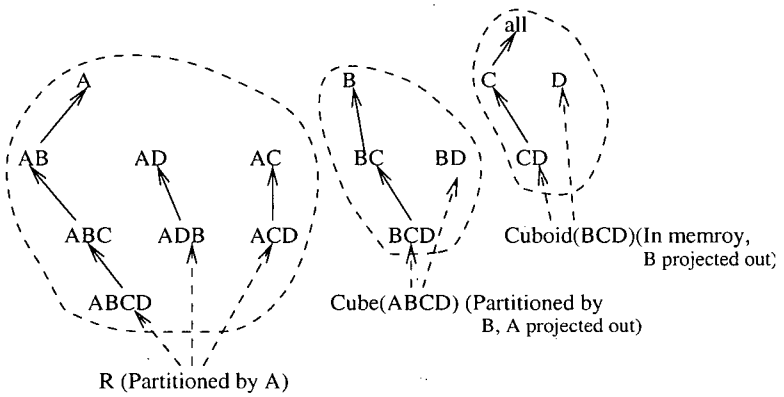
When the above CUBE algorithms are applied to sparse data sets, their performance becomes poor. Group-bys for sparse data sets are more likely to be large; buffering intermediate group-bys in memory requires too much memory. If the main memory is limited, then intermediate group-bys will be written out and read into memory multiple times, which increases I/O dramatically. Moreover, predictation of the size of group-bys becomes very difficult, because the real size of a group-by may not be proportional to the product of cardinalities of the GROUP BY attributes. This

makes the cost of computation in PipeSort and PipeHash no longer feasible.

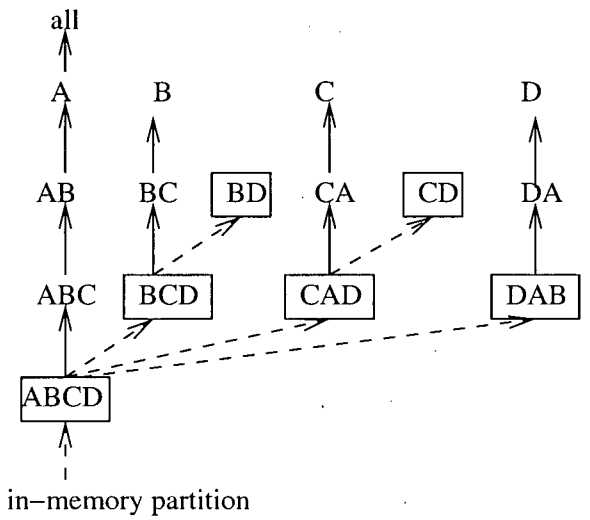
More recently, Ross and Srivastava proposed an efficient top-down algorithm designed for large, high-dimensional and sparse CUBEs [14]. Their algorithm consists of two parts: PartitionedCube and MemoryCube. PartitionedCube partitions the data on some attribute into memory-sized units and MemoryCube computes the CUBE on each in-memory partition.

Partitioning in PartitionedCube is very similar to PipeHash. The algorithm chooses an attribute to partition input into fragments. Then all cuboids containing that attribute will be computed on each fragment separately. For example, if CUBE is to be computed on attributes {A, B, C, D}, we might partition the input relation on attribute A, and get three partitions. Then, we compute cuboids ABCD, ABC, ABD, ACD, AB, AC, AD and A for each partition. By taking the union of corresponding partial cuboids computed from each partition, we get finally the complete cuboids. Then cuboid ABCD can be taken as the input to compute another cuboid. PartitionedCube is called recursively if the fragments or cuboid ABCD is still too big to fit in the memory; in that case, the data will be further partitioned on other attributes. Figure 2.8(a) gives an illustration of this example.

Once the input of PartitionedCube fits in the memory, then MemoryCube can be applied. MemoryCube is a sort-based algorithm, which is its main difference from PipeHash. Like PipeSort, however, MemoryCube algorithm takes advantage of the Pipelining technique. It tries to minimize the number of pipelines and hence, the number of sorts. Its Paths algorithm(not to be discussed in detailed here), guarantees that the number of pipelines(paths) it generates will be the minimum number of paths to cover all the nodes in the lattice. Figure 2.8(b) shows the paths for 4-dimension CUBE computation. There are six pipelines in total built from the input data. The cuboids in boxes are the heads of the pipelines. Sorting is required to create the head of the pipeline, which is shown as dash lines in Figure 2.8(b), however, no sorting is needed in the pipelines.



(a) Partitioning



(b) Paths Found by MemoryCube

Figure 2.8: Examples for PartitionedCube and MemoryCube Algorithms [14]

Since PartitionedCube only considers pipelines in the MemoryCube, this algorithm tries to reduce the amount of I/O for intermediate results, and thus enhance the performance for sparse CUBE computation.

Array-Based Algorithms

When using the array-based algorithms, as one proposed in [13], data sets are stored in a multi-dimension array, where each coordinate matches a CUBE attribute. A tuple's location in the array is determined by its value in each dimension. The algorithm requires no tuple comparison, only array indexing. Unfortunately, if the data is sparse, the algorithms become infeasible, as the array becomes huge. Therefore, we find array-based algorithms are too limited to warrant further discussion here.

2.4.2 Bottom-Up CUBE Algorithm

Our background search revealed only one bottom-up algorithm. It was introduced in [4] by K. Beyer and R. Ramakrishnan, and called the BottomUpCube, BUC for short. It especially targets iceberg-cube computation.

Setting thresholds in iceberg queries always cuts off a lot of cells in general cuboids. For the data set used in [4], and which was also used in our experiments, as many as 20% of the group-bys consisted entirely of cells with support as one. For the iceberg queries with minsup higher than 1, those group-bys do not need to be computed at all. This makes sense to consider a way to prune as early as possible in CUBE computation. Unfortunately, when we traverse a lattice in a top-down fashion, we can not prune cells which have insufficient support in any cuboid, until the last step. For example, suppose the threshold is set by specifying HAVING Count(*) >= 2 in iceberg-cube (the minsup is 2). Before we compute cuboid ABC from cuboid ABCD, we can not prune the cells with support as in 1, for example, a1b1c1d1(minsup:1) and a1b1c1d2(minsup:1). This is because they contribute to the cells in ABC, whose supports are bigger than 1, for example, a1b1c1(minsup

is 2). However if we compute from cuboid ABC to cuboid ABCD in a bottom-up fashion, pruning is possible. Although cuboid ABCD can not be directly computed from cuboid ABC, we can make sure that tuples which do not contribute to cells in cuboid ABC will not contribute to cells in ABCD. We could therefore prune out those tuples in the raw data earlier, before the computation for ABCD proceeds.

Thus, in BUC, a bottom-up approach is adopted. The idea is to combine the I/O efficiency of the PartitionedCube algorithm, with minimum support pruning. The processing tree of BUC is illustrated in Figure 2.4(c). The numbers in Figure 2.9 indicate the order in which BUC visits the group-bys.

A skeleton of BUC is shown in Figure 2.9; 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 2.4(c), $\mathcal{T}_B = \{B, BC, BD, BCD\}$. Prefix in line 9 in Figure 2.9 indicates the current processed cuboid's GROUP BY dimensions.

Take the BUC processing tree in Figure 2.9 as an example: BUC starts with cuboid *all*, and then cuboid with GROUP BY attribute A. For each value v_j in A, the data set is partitioned. Then for those partitions with higher support than minsup, BUC is called recursively in a depth-first manner to process other dimensional group-bys (in lines 14-16). For example, for partition $\mathrm{A}v_1$, in the first further recursion, BUC proceeds partitioning on attribute B, producing finer partitions $\mathrm{A}v_1\mathrm{B}v_1$ to partitions $\mathrm{A}v_1\mathrm{B}v_m$. Afterward, BUC is recursively called on those finer partitions to produce some cells in cuboids ABC, ABCD and ABD. When all recursions for partition $\mathrm{A}v_1$ return, BUC proceeds in the same way on other partitions for $\mathrm{A}v_j$. When all partitions based on A finish, BUC continues on attributes B, C and D in the same way.

Figure 2.10 shows how BUC partitioning proceeds. The arrows shows the partitioning order. The gray area depicts those partitions pruned out based on the constraints(minsup in this case).

Although BUC can exploit pruning, it can not optimize by share-sort or

```
1.  Algorithm BUC-Main
2.  INPUT: Dataset R with dimensions {A_1, A_2, ... 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, ... A_m}.
6.      for each dimension A_i (i from 1 to m) do
          BUC(R, T_{A_i}, Spt, {})
7.  CUBE COMPUTATION:
8.      procedure BUC(R,T_{A_i}, Spt, prefix)
9.        prefix = prefix ∪{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 ≥ 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,T_{A_k}, Spt, prefix)
16.           end for
17.       end for
```

Figure 2.9: A Skeleton of BUC

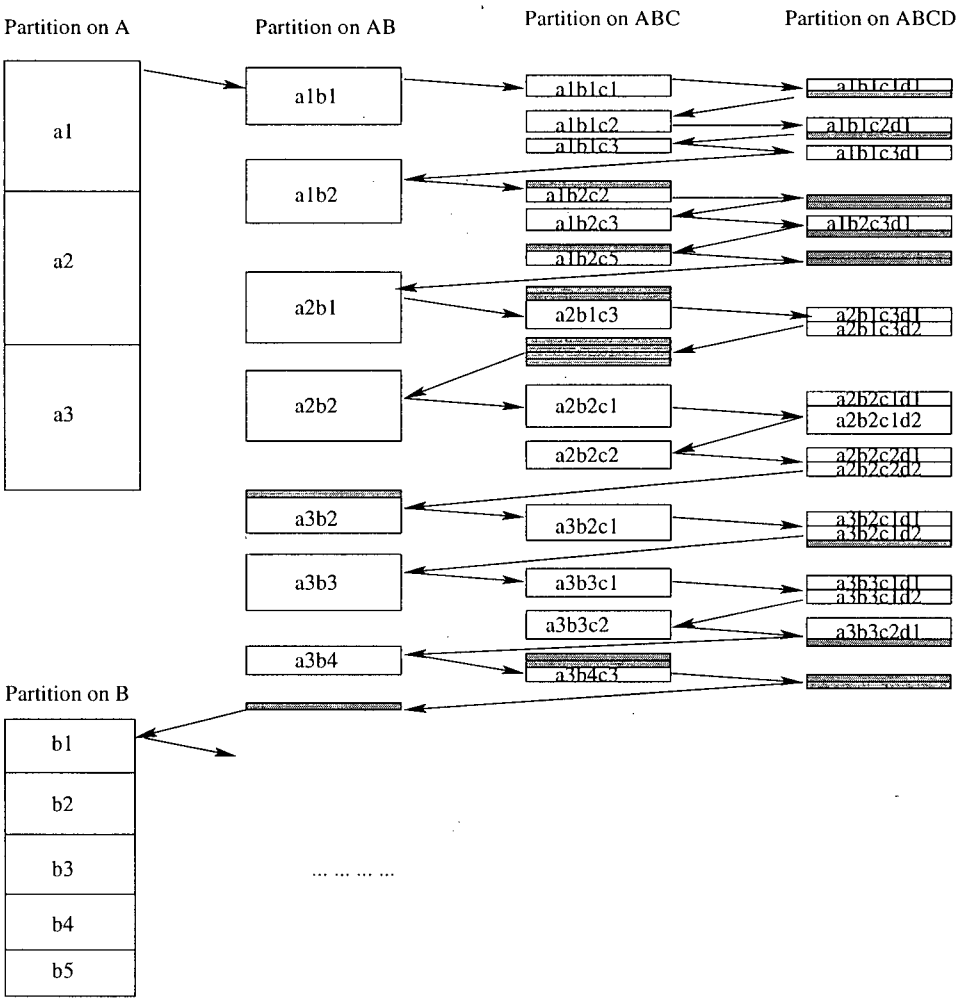


Figure 2.10: BUC Partitioning

smallest parent techniques.

Paper [4] compares BUC with PartitionedCube. It claimes that BUC performs better than PartitionedCube. The pruning significantly reduces running time when the minimum support is above 1. Even with minsup as 1, that is, full CUBE is computed, BUC still outperforms it.

Chapter 3

Parallel Iceberg-cube Algorithms

The key to success for an online system is the ability to respond to queries in a timely fashion. The compute and data intensive nature of iceberg-cube queries necessitates a high performance machine. In the past, this required expensive platforms, such as symmetric multiprocessor machines. In recent years, however, a very economical alternative has emerged: a cluster of low-cost commodity processors and disks. PC-clusters provide several advantages over expensive multiprocessor machines. 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 have significantly improved communication bandwidth and latency. Second, although I/O and the use of commodity disks are weaknesses in these systems, as we show, parallelism can easily be exploited. Third, the affordability of PC-clusters makes them attractive for small to medium sized companies and they have been the dominant parallel platform used for many applications [5], including association rule mining [18].

In the remainder of this thesis, we will discuss various novel algorithms we have developed for parallelizing iceberg-cube computation. Our focus is on practical

techniques that can be readily implemented on low cost PC clusters using open source, Linux and public domain versions of the MPI message passing standard. As our results apply to low cost clusters, the question arises of how much our results may generalize to higher cost systems. In Section 4, 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 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, that is, the cells of cuboids, remains distributed where processors output to their local disks.

In this section, we introduce the algorithms. As shown in Table 1.1, 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 work 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 computation, an important issue is the strategy for traversing the cube lattice. As discussed earlier, top-down traversal may exploit share-sort, whereas bottom-up traversal exploits pruning based on the constraints. Our algorithms consider these possibilities; 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.

3.1 Algorithm RP

Recall from Figure 2.4(c) that the processing tree of BUC consists of 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. The data set is replicated on all machines in a cluster. Each processor reads from its own copy of the dataset, and outputs the cuboids to its local disk. The skeleton of RP is showed in Figure 3.1.

Figure 3.2 gives an example of computing a 4-dimensional CUBE on a cluster of 4 PCs. In total, 4 tasks are created: subtrees rooted by A, B, C and D respectively. Each machine compute one task.

3.2 Algorithm BPP

While RP is easy to implement, it appears to be vulnerable in at least two of its aspects. First, the definition of a task may be too simplistic in RP. The division of

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 3.1: A Skeleton of the Replicated Parallel BUC Algorithm

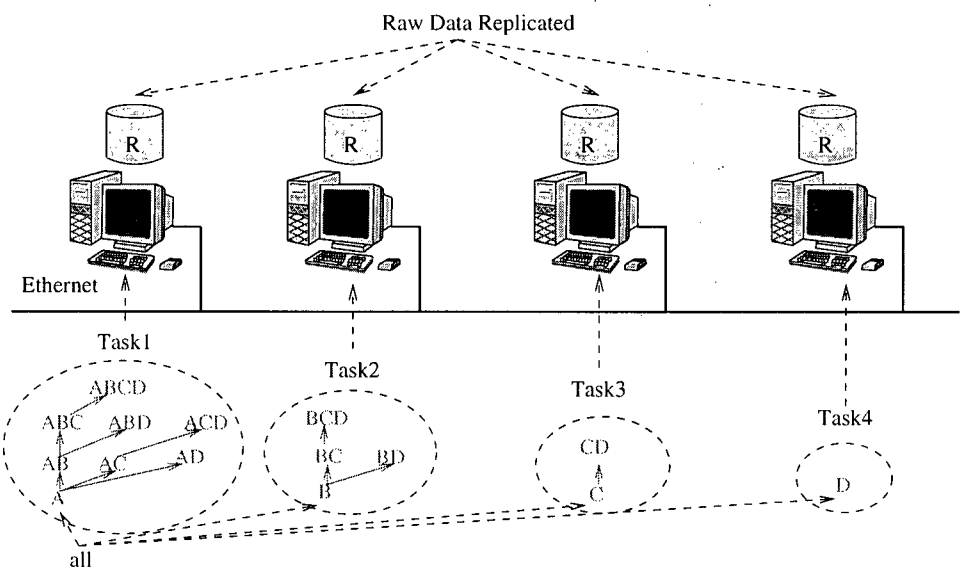


Figure 3.2: Task Assignment in Algorithm RP

the cube lattice into subtrees is coarse-grained. One consequence is that some tasks are much bigger than others. For example, the subtree rooted at A, $\mathcal{T}_A$, is much larger than that rooted at C, $\mathcal{T}_C$. Thus, load balancing is poor. Second, BUC is not optimized in writing performance. To address these problems, we developed the algorithm called Breadth-first writing Partitioned Parallel BUC, or BPP for short.

3.2.1 Task Definition and Processor Assignment

To achieve better load balancing, BPP tries to get finer-grained tasks by range partitioning on each attribute. This is motivated by Ross and Srivastava's design of the Partitioned-Cube, which attemps to partition data into chunks which fit in memory [14]. BPP partitions data in the following way:

- 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 ($R_{1(j)}, \ldots, R_{m(j)}$). 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}, and so on. Notice that as far as BPP execution is concerned, range partitioning is basically a pre-processing step.

If there are m cube attributes, then there will be a total $m \times n$ chunks. Each chunk corresponds to one task. The processor who has the chunk in the local is responsible for processing it. If processor P_j process chunk $R_{i(j)}$, where $R_{i(j)}$ is produced by range partitioning on attribute i, P_j computes the (partial) cuboids in the subtree rooted at A_i. These cuboids are partial because P_j only deals with

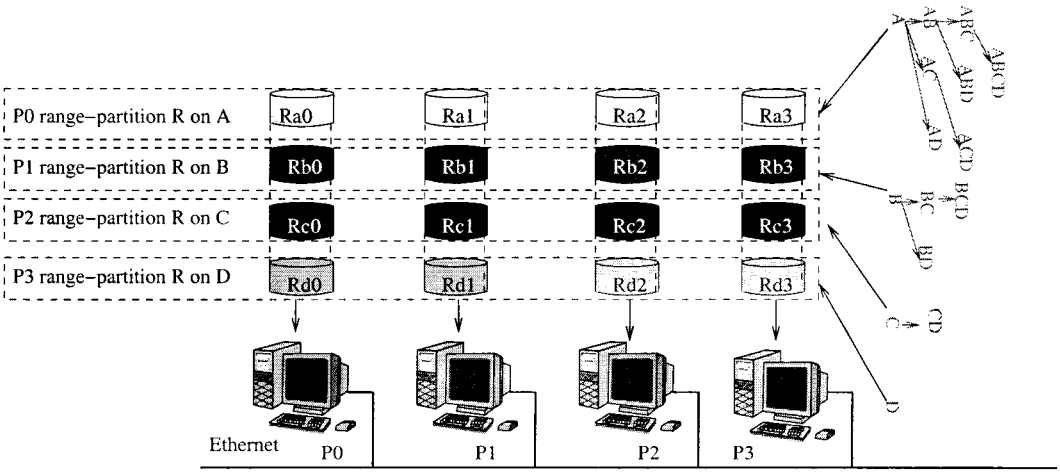


Figure 3.3: Task Assignment in BPP

the part of the data it controls, in this case, $R_{i(j)}$. The cuboids are completed by merging the output of all n processors.

Figure 3.3 illustrates task allocation and process in BPP. Each of the 4 processors in the cluster takes on the responsibility of range partitioning the raw data set R on one dimension and distributing the resulting partitions across the processors. Since there are 4 cube dimensions in total, after data partitioning each processor gets 4 chunks. Data chunks in the same color on the same row are partitioned on the same attribute and have no overlap. However, data chunks located in the same processor are partitioned on different attributes and may have overlap. A processor takes chunk $R_{i(j)}$ to compute subtree $\mathcal{T}_i$, for example, P1 would use $R_{c(1)}$ to compute subtree $\mathcal{T}_C$.

By partitioning data across processors, BPP achieves better load balancing than RP. If data can be evenly distributed among processors, then the load may be very well balanced in a homogeneous environment.

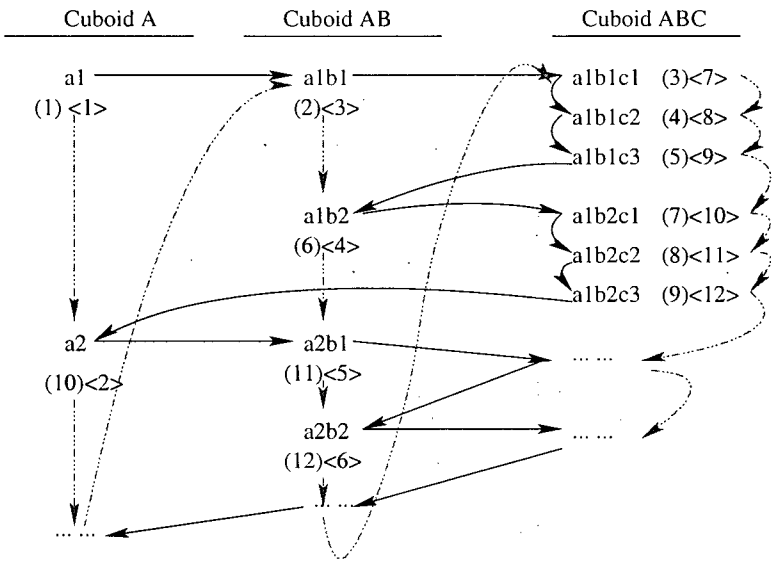


Figure 3.4: Depth-first Writing vs Breadth-first Writing

3.2.2 Breadth-first Writing

BUC computes in a bottom-up manner, and the cells of the cuboids are written out in a depth-first fashion. In the situation shown in Figure 3.4, there are three attributes A, B and C, where the values of A are a_1, a_2, and so on, values of B are b_1 and b_2 , values of C are c_1, c_2 and c_3. As shown in Figure 2.9, the tuples of a_1 are 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 to the cell $a_1b_1c_1$, then to $a_1b_1c_2$, and so on. In Figure 3.4, the number in round brackets beside each node denotes the order in which the cell is processed and the output for depth-first writing; and the black solid lines denote the writing sequence.

Note that these cells 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. Clearly in depth-first writing, the writing to the cuboids is scattered. This incurs a high I/O overhead. We could possibly use buffering to alleviate the

```
1.  Algorithm BPP
2.  INPUT: Dataset R with dimensions {A_1, A_2, ... 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 3.2.1
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), T_Ai, Spt, {}) (with output written on
                                        local disks)
11.         end for
12.     end do

13. Subroutine BPP-BUC(R, T_Ai, Spt, prefix)
14.     prefix = prefix ∪{A_i}
15.     sort R according to the attributes ordered in prefix
16.     R' = R
17.     for each combination of values of the attributes in prefix do
18.         if (the number of tuples for that combination ≥ 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', T_Ak, Spt, prefix)
24.     end for
```

Figure 3.5: A Skeleton of the BPP Algorithm

scattered writing to the disk. However, this requires 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. Returning to the example in Figure 3.4, BPP completes the writing of 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 outputted, and

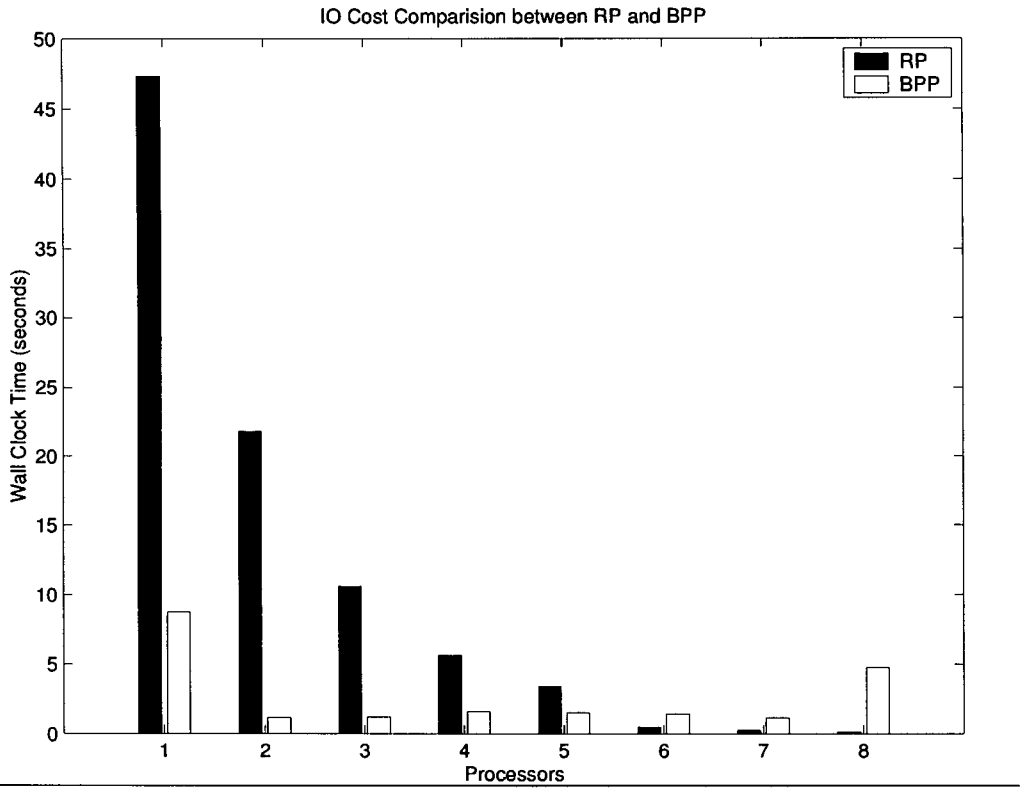


Figure 3.6: I/O comparison between BPP(Breadth-first writing) and RP(Depth-first writing) on 9 dimensions on a dataset with 176,631 tuples, input size is 10Mbyte and output size is 86Mbyte.

so on. In Figure 3.4, the number in angled brackets beside each node denotes the order in which the cell is processed for breadth-first writing, while the red dash lines depict its writing sequence.

Figure 3.5 gives a skeleton of BPP. As described in Section 3.2, the pre-processing step of range partitioning the dataset assigns to each processor P_j of the appropriate tasks.

In the main subroutine BPP-BUC, breadth-first writing is implemented by first sorting the input dataset on the "prefix" attributes in line 15 in the skeleton. In our example, if the prefix is A, meaning that the dataset has already been sorted on A, then line 15 sorts the dataset further on the next attribute B. The loop starting

from line 17 then 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 of the need to begin pruning 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 4, the total I/O time RP took to write the cuboids was more than 5 times greater than the total I/O time for BPP. Figure 3.6 gives the I/O comparison between RP(depth-first writing) and BPP(breadth-first writing).

3.3 Algorithm ASL

Although BPP gives a solution for load balancing, this solution is still not satisfactory under some circumstances. The potential downfall of BPP comes from the fact that the amount of work each processor does is dependent on the initial partitioning of the data. However, the size of the task depends on the degree of skewness in the data set 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. For example, for an attribute named Gender, only two possible values, Female and Male, can be assigned to it. Range partitioning then can produce only 2 chunks. Even if we have more than 2 processors, only two of them will get applied to chunks; the others will be relatively lightly loaded.

This motivates the development of another algorithm, called Affinity Skip List, or ASL for short. ASL tries to create tasks that are as small as the cube

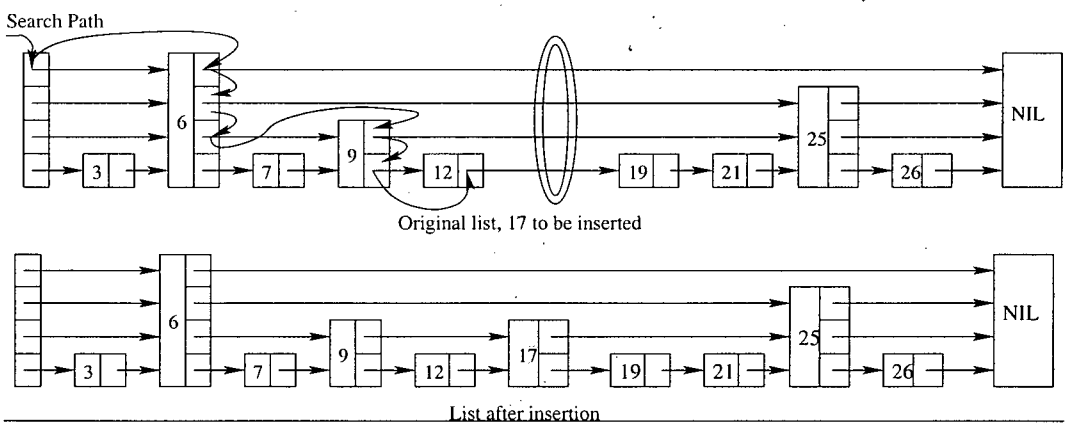


Figure 3.7: Pictorial Description of Steps Involved in Performing an Insertion [22]

lattice allows: each node in the lattice makes a task. This allows efficient use of the processors, quite independent of the the skewness and dimensionality of the data set. In the following, two key features of ASL are presented: the data structure used, and the processor assignment.

3.3.1 Using Skip lists

A skip list is a data structure proposed by W. Pugh [22]. It is much like a linked list with additional pointers. Figure 3.7 is an example of a skip list. The lowest levels of nodes make a linked list, the higher levels of nodes are used for efficient search and insert operations. As showed in Figure 3.7, searching or insertion always starts from the highest level of the head node. If the next link emitted from that level points to a node that contains an element bigger than the element which is to be inserted or searched, we drop one level from the starting node, otherwise, we follow the link to the next node, and try the next step from there. Figure 3.7 shows how an element with key 16 is added into a skip list. The number of levels a new inserted element should have is determined randomly, but not allowed to exceed a threshold set by users.

The benefits of using a skip list are threefold. First, ASL exhibits good average case behavior for insertion and searching, quite similar to that of a balanced tree,

yet the implementation details are much simpler. Second, each node in the structure requires very little storage overhead. Third, skip list incrementally increases as more elements are added, and the sort order of the list is always guaranteed. This is very important, because before sorting the data set need not be entirely loaded.

ASL uses skip lists to maintain the cells in cuboids. While it scans the raw data set, ASL builds skip lists incrementally. If there is a node in the skip list representing the new read-in tuple, then the aggregates and support counts of that node are updated; otherwise a new node will be inserted into the skip list. In theory, if there are k cuboids and if there is enough memory, ASL can maintain all k skip lists simultaneously for one scan of the data set. But for the data sets used in our experiments, this optimization brings minimal gain, so we did not explore that here.

3.3.2 Affinity Assignment

Now, let's consider the task assignment policy of ASL. In order to achieve good load balancing, ASL defines tasks with very fine granularity. It takes each cuboid as a task and assigns it to processors dynamically. During task scheduling, ASL adopts a top-down approach to traversing the cube lattice. It always tries to assign uncomplete high dimensional cuboids to processors, while taking affinity into account. Once a processor finishes one task, it is assigned a new one which has some kind of affinity with the previously one.

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

In another situation, if a processor has just created the skip list for ABCD, this skip list is still useful if the processor is next assigned the task of computing the cuboid BCD, because now we need only take the counts of each cell in ABCD, and add them to the counts of the appropriate cells in the skip list for BCD. Then

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, as described in Section 3.3.2
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 denote the skip list from 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 denote the skip list from 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 if the support threshold is met

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 the support counts accordingly
25. **end for**
26. Traverse L, and write out the cells if the support threshold is met

Figure 3.8: A Skeleton of ASL

groupings done for the skip list for ABCD are not wasted. For example, suppose in ABCD, a cell corresponds 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 3.8 shows a skeleton of Algorithm ASL. To implement prefix or subset affinity, a processor is designated the job of being the "manager" responsible for dynamically assigning the next task to a "worker" processor. Specifically, the manager does the following:

- first tries to exploit prefix affinity, because if that is possible, the worker processor then has no need to create a new skip list for the current task/cuboid. The previous skip list 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 3.8;

- then tries to exploit subset affinity, if prefix affinity is not applicable. Instead of scanning the dataset, the worker processor can use the previous skip list to create the skip list for the current task. This is executed by the subroutine subset-create in Figure 3.8;

- 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 skip list is created from scratch.

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

The affinity scheduling is very helpful for sort-sharing, especially when the

number of available processors is small. But as more processors are available, the affinity relationship between tasks assigned to the same processor tends to be weak. For example, if we have 2 processors, we may very possibly assign both ABCD and ABC to one machine; however, if we have 16 machines, this possibility becomes light, since we don't want machines to lie idle just to maintain strong affinity. In such a case, one processor may compute ABCD and another may compute ABC, then both would need to sort on ABC. Duplicated sortings then occur.

Since ASL's task scheduling is dynamic, depending on how soon each processor finishes its task, the lattice traversal sequence can not be determined in advance. Different runnings very likely result in different traversal sequences. This makes ASL quite different from other top-down algorithms, such as PipeSort or PipeHash.

3.4 Algorithm PT

By design, ASL does a very good job of load balancing. 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 where prefix or subset affinity cannot be well exploited, and thus not much sort sharing is applicable. Second, ASL's top-down lattice traversal cannot prune those cells which lack minimum support from skip lists. As ASL executes, whether a cell has minimum support or not cannot be determined until the data set has been scanned entirely. Furthermore, at the end of the scan, even if there is a cell 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.

In an effort to combine the advantages of pruning in a bottom-up algorithm on one hand, with load balancing and sort-sharing of top-down lattice traversal on the other, we developed the algorithm called Partitioned Tree, (PT).

Recall that in RP and BPP, tasks are at the granularity level of subtrees rooted at a certain dimension, for example, $\mathcal{T}_{A_i}$. In ASL, tasks are merely nodes

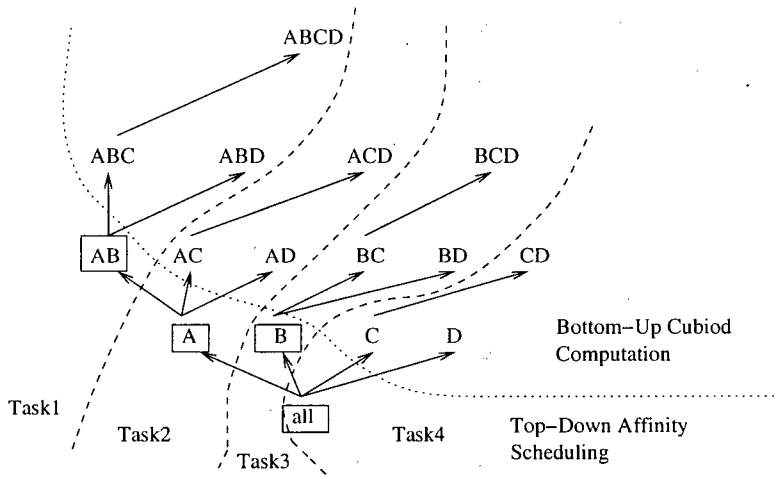


Figure 3.9: Binary Division of the Processing Tree into Four Tasks

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, each having an equal number of nodes. Binary division is achieved by simply cutting the farthest left edge emitted from the root in a BUC processing tree or subtree in recursive callings. For instance, the BUC processing tree shown in Figure 2.4(c) can be divided into two parts: $\mathcal{T}_A$ 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 these two subtrees: $\mathcal{T}_B$ and $\mathcal{T}_{all} - \mathcal{T}_A - \mathcal{T}_B$. Figure 3.9 shows the four subtrees. Each of these four subtree makes a task.

Obviously, each time binary division is applied, two subtrees of equal size are produced. Through binary division, we finally achieve tasks of same size and appropriate granularity. Combining dynamic scheduling and binary division nicely solves the load balancing problem in PT.

Like ASL, PT also exploits affinity scheduling. During processor assignment, the manager tries to assign a task to a worker processor 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

```
1.  Algorithm PT
2.  INPUT: Dataset R cube dimensions {A_1, ..., 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 T
10.         sort R on the root of T (exploiting prefix affinity if possible)
11.         call BPP-BUC(R, T, Spt, {})
12.     end do
```

Figure 3.10: A Skeleton of PT

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.

In Figure 3.9, the roots of each subtree, enclosed in boxes, actually make up a small tree. The scheduling just happens on this small tree, similar to ASL. Once a processor gets a task, that is, a subtree, it computes it in a bottom-up manner, much like computing an RP's task. In this way, we seamlessly combine top-down and bottom-up methods, getting the benefits of both pruning and sort-sharing.

Figure 3.10 shows a skeleton of PT. The step that requires elaboration is line 9, namely the exact definition of $\mathcal{T}$. In general, as shown in Figure 3.9, 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 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 3.5, if a full subtree is given, no change is needed. Otherwise, the loop needs

to skip over the chopped branches.

Since PT treats each final subtree resulting from binary division as a task, in an extreme case binary division will eventually create a task as each node in the cube lattice, as in ASL. Since task granularity in ASL might be too fine, in PT a parameter is used to determine when binary division stops, thus defining how fine tasks can be. The parameter is set as the ratio of the number of tasks to the number of available processors. The higher the ratio, the better the load balancing but the less pruning can be explored in each task. Determining the parameter enables a tradeoff between load balancing and pruning. In Figure 3.9, the dotted line between "Bottom-Up Cuboid Computation" and "Top-Down Affinity Scheduling" depicts this tradeoff. Moving up the line means letter load balancing; moving down the line means more pruning. PT wisely leaves this decision up to applications. For the experimental results presented later, we used the parameter "$32n$" to stop the division, once there are so many tasks (where n is the number of processors).

3.5 Hash-based Algorithms

We also implemented two hash-based CUBE algorithms. In the following, we will briefly discuss them.

3.5.1 Hash Tree Based Algorithm

This algorithm was developed after BPP proved to have poor load balancing. Since BPP's performance is greatly affected by data skewness, which we could not change, it appears there was no way to improve it. However, considering most Association Rules Mining (ARM) algorithms proceed in a bottom-up fashion, also taking advantage of pruning, we then thought about applying the techniques of parallel ARM to CUBE computation.

The prototypical application of ARM is a "market-basket analysis", where the items represent products, and the records in a database represent point-of-sales

Items

Jane Austen	A
Agatha Christie	C
Sir Arthur Conan Doyle	D
Mark Twain	T
P.G. Wodehouse	W

Database

Transaction	Items
1	A C T W
2	C D W
3	A C T W
4	A C D W
5	A C D T W
6	C D T

Frequent itemsets (min_sup = 50%)

Support	Itemsets
100% (6)	C
83% (5)	W, CW
67% (4)	A, D, T, AC, AW CD, CT, ACW
50% (3)	AT, DW, TW, ACT, ATW, CDW, CTW, ACTW

Association rules (min_conf = 100%)

A ⟶ C (4/4)	AC ⟶ W (4/4)	TW ⟶ C(3/3)
A ⟶ W (4/4)	AT ⟶ C (3/3)	AT ⟶ CW(3/3)
A ⟶ CW (4/4)	AT ⟶ W (3/3)	TW ⟶ AC (3/3)
D ⟶ D (4/4)	AW ⟶ C (4/4)	ACT ⟶ W (3/3)
T ⟶ C (4/4)	DW ⟶ C (3/3)	ATW ⟶ C (3/3)
W ⟶ C (5/5)	TW ⟶ A (3/3)	CTW ⟶ A (3/3)

Figure 3.11: Frequent Itemsets and Strong rules for a Bookstore Database [20]

data at large grocery stores or department stores. Each record contains several items. The objective of ARM is to generate all rules with specified confidence and support. An example rule might be, "90% of customers buying product $\{A, B, C\}$ also buy product $\{D, E\}$.", where the confidence of the rule is 90%. In ARM terminoloy, A, B, C, D and E in this rule are called "items"; $\{A, B, C\}$ and $\{D, E\}$ are called "itemset". Later, we will use "k-itemset" to denote an itemset containing k items. As well as CUBE, ARM aslo uses "support" to indicate how frequent an itemset occurs as a subset in transactions. Users can specify a "minimum support" (minsup) and "minimum confidence" (minconf) in their queries.

Most ARM algorithms involve the following steps:

1. Find all frequent itemsets satisfying some specified minimum support.

2. Generate strong rules having minimum confidence from the frequent itemsets.

The first step of ARM is much like a iceberg-cube problem if we imagine items are attributes with only one value. Then, generating all frequent itemsets means generating all different dimensional group-bys above a specified threshold(minsup).

Consider the example bookstore-sales database shown in Figure 3.11. There

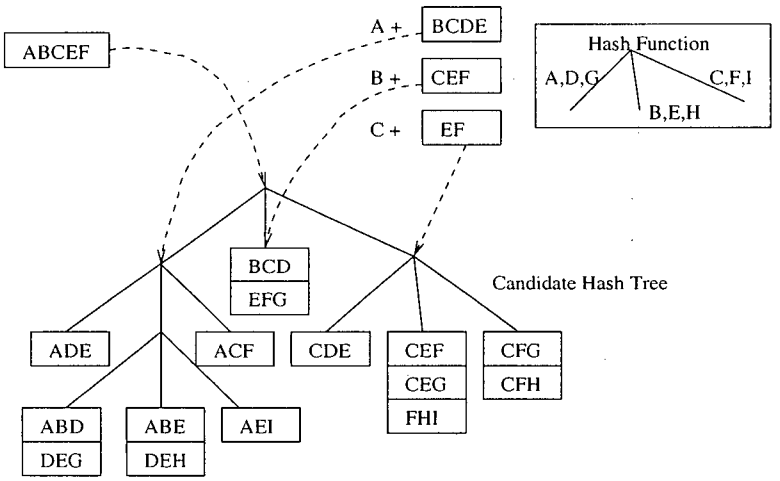


Figure 3.12: Subset Operation on the Root of a Candidate Hash Tree [23]

are five different items (names of authors the bookstore carries), $I = \{A, C, D, T, W\}$. The database comprises six customers who bought books by these authors. Figure 3.11 shows all the frequent itemsets contained in at least three customer transactions, that is, minsup = 50%. The figure also shows the set of all association rules with minconf = 100%.

The Apriori algorithm by Rakesh Agrawal and colleagues [20] has emerged as one of the best ARM algorithms, and also serves as the base algorithm for most parallel algorithms. Apriori uses a complete, bottom-up search, iteratively enumerating from frequent 2-itemsets to higher dimensional frequent itemsets. The algorithm has three main steps:

1. Generate candidates of length k from the frequent (k-1)-itemsets, by a self-join on F_{k-1}. For example, for $F_2 = \{AC, AT, AW, CD, CT, CW, DW, TW\}$, we get $C_3 = \{ACT, ACW, ATW, CDT, CDW, CTW\}$.

2. Prune any candidate that has at least one infrequent subset. For example, CDT will be pruned because DT is not frequent.

3. Scan all transactions to obtain candidate supports.

These three steps are called iteratively from k=2 until no more new frequent itemset can be generated.

Apriori stores the candidates in a hash tree for fast support counting. In a hash tree, itemsets are stored in the leaves; internal nodes contain hash tables (hashed by items) which direct the search for a candidate. The hash tree structure of Apriori is very efficient for candidate searching and insertion. Once a transaction is read in, all of its subsets can be quickly computed and inserted into the hash tree if they are not there already. Figure 3.12 gives an example of subset operation on the root of a candidate hash tree.

Obviously, the bottom-up idea behind both Aprior and BUC is the same, except BUC searches the tree in a depth-first order while Aprior searches in a breadth-first order. From this observation, we developped a CUBE algorithm with a similar hash tree structure as in Aprior, and exploit the breadth-first searching in CUBE computation exactly as in Aprioir. We kept the major structure of the Apriori algorithm and made only little modification to accommodate CUBE computation. For example, since CUBE doesn't assume only a value for each attribute (item in ARM), we built a global index table which counts all values of all attributes as items.

For a small data set, this algorithm is feasible. However, its performace was proved unsatisfactory. Breadth-first searching creates too many candidates to be maintained in the hash tree. This is mainly because the global index table contains too many items, exactly the sum of the cardinalities of all CUBE attributes. This creates a large amount of candidates. If the CUBE is sparse, the situation is even worse. Although we can count on pruning to eliminate many candidates, the hash tree is still a huge burden before pruning, and quickly consumes all available memory.

Unfortunately, we had to admit this attempt failed. Since the performance of this hash tree based algorithm lags far behind other algorithms, we omit it from the following discussion.

3.5.2 Hash Table Based Algorithm

After we finished the implementation of ASL, we tried to use the hash table as an alternative data structure for ASL, to see whether better preformance could be achieved. Then the Affinity Hash Table based algorithm was developed, AHT for short.

As with PipeHash, AHT uses hash tables to maintain cells of nodes in a lattice, group-bys. However, AHT avoids creating a hash table for each cuboid. Once subset affinity becomes applicable, it reuses the hash table created for the previous task. Specifically, AHT builds an index which makes it possible to collapse the previous hash table whenever subset affinity is found.

For this purpose, each CUBE attribute is assigned several bits which, when combined, form the complete index of buckets in a hash table. For example, for a 3-dimensional CUBE with attributes A, B and C, we give A three bits, B two bits and C one bit. Then the hash tables index has 6 bits (in binary) and the size of the hash table will be 2^6. Whenever a tuple (cell) is read in, its location in the hashtable is determined by its values for the CUBE attributes. In this example, for its index, the first three bits are decided by the value for A, the next two bits are decided by the value for B, and the last bit is decided by the value for C.

The number of bits assigned to each attribute depends both on the cardinality of that attribute and on how many tuples are in the raw data set. Originally, the bits assigned to an attribute X is $\log(card(X))$, where $card(X)$ is the cardinality of X. This implies the length of a hash table would be the product of the cardinalities of all attributes. However, if the data set is sparse, this product would be much larger than the size of the data set. In this case, the bits assigned to each attribute would shrink appropriately, in order to define a smaller index. A smaller index, however, may introduce collisions. Here we simply tradeoff memory occupation with run time. This tradeoff would introduce severe bucket collision when many cells need to be maintained by the hash table. It degrades AHT's performance severely, especially

Algorithm AHT

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 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 a subset of the set of dimensions of the previous task, or the set of dimensions of the first task
10. let C denote the hash table from that task
11. call subset-collapse($C, Spt, A_i, \ldots, A_j$)
12. else call subset-newHashTable($R, Spt, A_i, \ldots, A_j$)
13. **end do**

14. Subroutine subset-collapse($C, Spt, A_i, \ldots, A_j$)
15. Collapse C based on $A_i, \ldots, A_j$
16. Write out the cells if the support threshold is met

17. Subroutine subset-newHashTable($C, Spt, A_i, \ldots, A_j$)
18. initialize a hash table$H$
19. **for each** tuple in C **do**
20. find the right cell in H (created if necessary)
21. update the aggregate and the support counts accordingly
22. **end for**
23. Traverse H, and write out the cells if the support threshold is met

Figure 3.13: A Skeleton of AHT

when problem size increases or a high dimensional CUBE need to be computed. We will discuss this further in Chapter 4.

As ASL, AHT also takes each group-by as a task. AHT's task scheduling is almost the same as ASL, except AHT does not prcocess prefix affinity differently from general subset affinity. If a new task's GROUP BY attributes make a subset of those of the previous task, then the hash table already built contains all cells needed for the new task. So, we will create no new hash table but shrink the existing one by collapsing some buckets. Further to the example mentioned above, if we've built the hash table for cuboid ABC, we now get a new task for cuboid AC. The buckets xxx 00 x, xxx 01 x, xxx 11 x and xxx 10 x are collapsed into xxx 00 x, with the aggregate and the support upgraded at xxx 00 x. Those attributes missing from the new task (but found in the previous one) determine how many and what buckets will be collapsed. In this example, C is the missing attribute. Since two bits (the forth and the fifth in the index) are assigned to C, then four buckets will be collapsed into one bucket.

Since the hash table does not maintain cells in any particular sorting order, no sorting is needed in AHT. If a sorted cuboid is required by users, the sorting will be done online when users give their queries. We call this post-sorting.

The skeleton of AHT is shown in Figure 3.13.

Chapter 4

Experimental Evaluation

In this Chapter, we give a performance comparison of five algorithms: RP, BPP, ASL, PT and AHT. The hash tree based algorithm is not included in this testing nor in the following discussion, because its performance lags far behind the others.

In order to give a fair evaluation, we investigate the algorithms' memory occupation first before explaining the testing environment, and then give our test results.

4.1 Memory Occupation

In four of the algorithms: RP, ASL, PT and AHT, the raw data set is replicated among processors. Conversely, BPP partitions the raw data set and distributes the partitions among processors. Let's first discuss data replication based algorithms.

In the simplest algorithm, RP, each processor loads the whole replicated data set, R, into its main memory as a large array for later computation, according to the task assigned to it. RP therefore only needs a space the size of R in the main memory for each processor.

Another data replication algorithm, PT, is also an array based algorithm. Like RP, its memory footprint is not much larger than R for each processor.

AHT uses hash tables as its data structure only to maintain cuboids. Since

the cells in a cuboid can be less than tuples in the data set, a hash table may possibly be much smaller than a data array in an array based algorithm. Besides cells, AHT needs also to maintain the index table for the hash table in memory. The index table is fixed-size in AHT; in other words, the number of buckets in the hash table is fixed. This number greatly affects AHT's performance. In order to make the experiment evaluation reliable, we set the number of buckets in the hash table to the number of tuples in R. Therefore, AHT's memory footprint is not much more than R. In an extreme case, such as where the cuboid contains all tuples in the raw data set, each processor of AHT needs space in its main memory for R cells, plus the $|R|$ indices for a hash table.

The memory footprint of ASL is the biggest of all the algorithms. It takes skip lists as its data structure. The memory overhead for each node of a skip list is mainly decided by the maximum number of forward links it allows a node to have. In our algorithm, we allow no more than 16 forward links in each node. Therefore, a node's memory footprint is no more than twice the size of an element of an array in array based algorithms, such as RP.

Like AHT, ASL does not load the entire data set into memory, but only maintains cuboids as skip lists. Thus, a skip list may be smaller than a data array. Even in an extreme case, such that a cuboid contains the whole data set, its skip list size would be no more than twice that of R.

As well as the current working skip list, each processor maintains a "root" skip list in its main memory, to maximize sort sharing among local tasks. Then in an extreme case, ASL's memory footprint will be no more than four times of R, for two skip lists in the memory of each processor.

The data partitioning based algorithm: BPP is the most memory-efficient algorithm. Since each processor only works on local chunks, its memory footprint is the maximum size of its local chunks. Even in an extreme case, where only one chunk gets produced when range partitioning on an attribute, the memory footprint

would be no more than R.

4.2 Experimental Environment

The experiments were carried out on a hetergeous PC cluster, consisting of eight 500MHz PIII processors with 256M of main memory and eight 266MHz PII processors with 128M of main memory. Each machine is attached with a 30Gbyte hard disk and is connected to a 100Mbit/sec Ethernet network.

The CUBE computations were performed on a weather data set containing weather conditions sent by various weather stations on land. The data set is the same as that used by Ross and Srivastava [14], and Beyer and Ramakrishnan [4]. It has 20 dimensions, and is very skewed on some of those dimensions. For example, partitioning the data on the 11^{th} dimension produces one partition which is 40 times larger than the smallest one.

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

- 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});
- with minimum support set at two.

For the dynamic scheduling algorithms ASL, PT and AHT, we overlapped the manager and one worker on one processor. This maximized the usage of the processor on which the manager resided, leading to a reasonable performance evaluation.

In the experiment, we investigated how the algorithms perform under different circumstances. We are concerned with the following issues in CUBE computation:

- load balancing, tested by comparing loads on each processor;
- scalability with processors, tested by varying the number of processors;
- scalability with problem size, tested by varying problem size;
- scalability with dimensions, tested by varying the number of dimensions;
- pruning effects, tested by varying the minimum support;
- accommodation for sparse CUBE computation, tested by varying the sparseness of the data set.

In the following figures, "wall clock" time means the maximum time taken by any processor. It includes both CPU and I/O cost.

4.3 Load Distribution

Figure 4.1 shows the load distribution among processors when testing on the baseline configuration. ASL, AHT and PT have quite an even load distribution while the loads distributed to each processor by RP and BPP vary greatly. For RP, the reason for the uneven load distribution is due to its static task assignment. Although the number of tasks is approximately equal, the amount of computation and I/O for the tasks differs significantly. For BPP, the dataset is partitioned statically across all nodes. Because the data is very skewed on some of the dimensions, the computation is not well balanced. ASL, AHT and PT decrease the granularity of the tasks to a single cuboid in ASL and AHT and to a small subtree in PT. The finer granularity leads to better load balancing, and the use of demand scheduling makes it easier to maintain balanced even when the data set is very skewed.

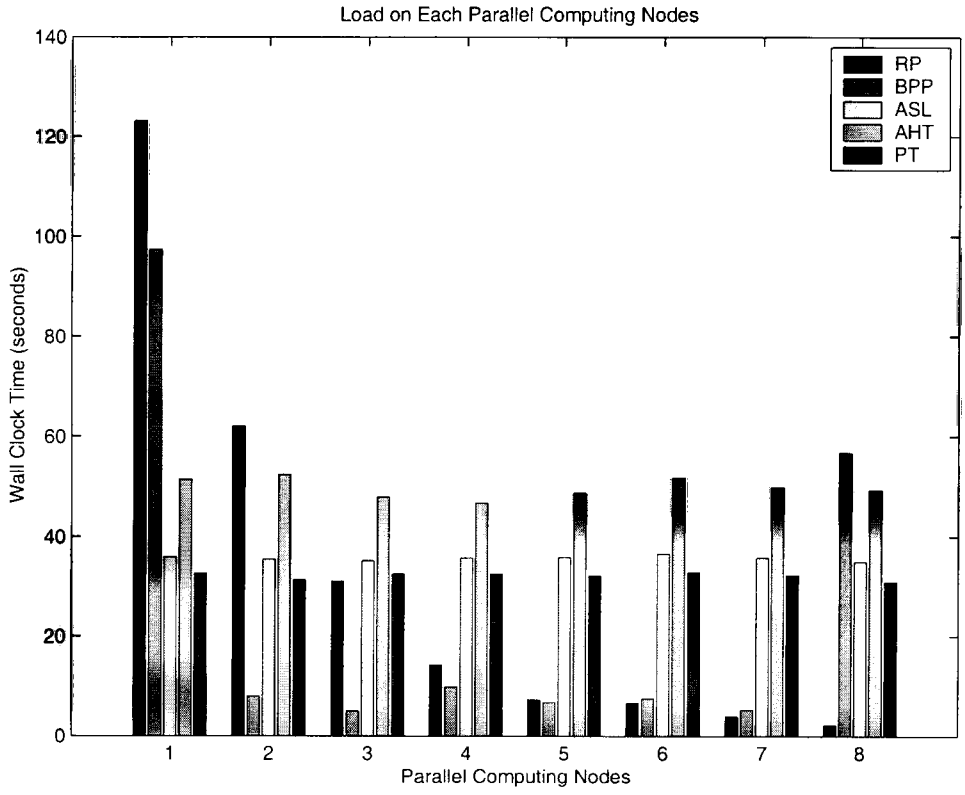


Figure 4.1: Load Balancing on 8 Processors

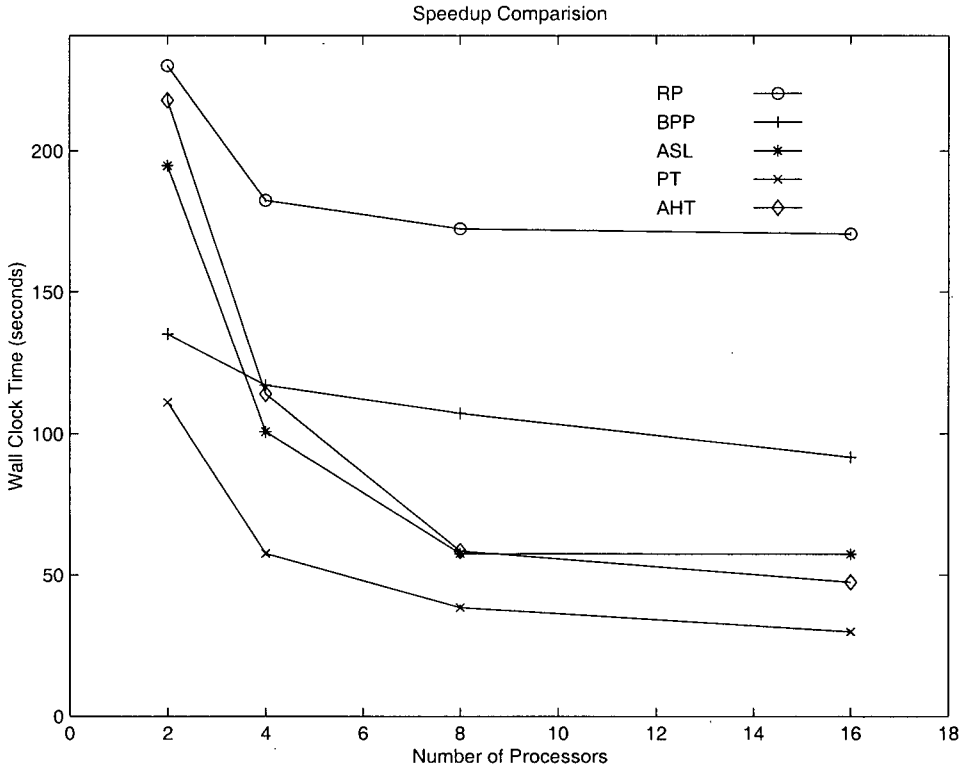


Figure 4.2: Scalability

4.4 Varying the Number of Processors

Figure 4.2 shows the performance of the algorithms when running on different numbers of processors. The performances are largely determined by each algorithms' load balancing ability; generally, the better the load balances, the better the performance.

RP's performance is the worst, no matter how many processors are used. Besides poor load balancing, RP's depth-first writing strategy exacerbates its poor performance as well.

BPP does well when running only on 2 processors, where the data partitioning is done quite evenly. However, as more processors are added to the computing environment, the data partitioning becomes uneven. Uneven tasks with coarse gran-

ularity quickly upset load balancing. BPP is quickly outperformed by ASL when four processors are available.

The performance of ASL is poor when run on only two processors. This is largely due to the overhead from creating and maintaining skip lists. When the number of processors increases, ASL gains from good load balancing and scales very well.

AHT's performance is similar to ASL's, because their task definition and scheduling are almost the same.

PT shows the best performance overall due to both good balancing and pruning. ASL, AHT and PT use affinity scheduling to take advantage of share-sort to reduce computation. As we mentioned in section 3.3.2, the affinity relationship among local tasks on one processor tends to weaken as the number of processors increases. It is interesting to note that the speedup from eight processors to sixteen processors is negligible, relatively.

4.5 Varying the Problem Size

Figure 4.3 shows that with increasing problem size, PT and ASL do significantly better than other algorithms. Both PT and ASL appear to grow sublinearly as the number of tuples increases. This is due to two factors. First, there is an overhead when 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 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 4.3 show that PT and ASL scale well with problem size and indicate that these algorithms could be used, given sufficient memory and disk space, to solve larger problems on larger cluster machines.

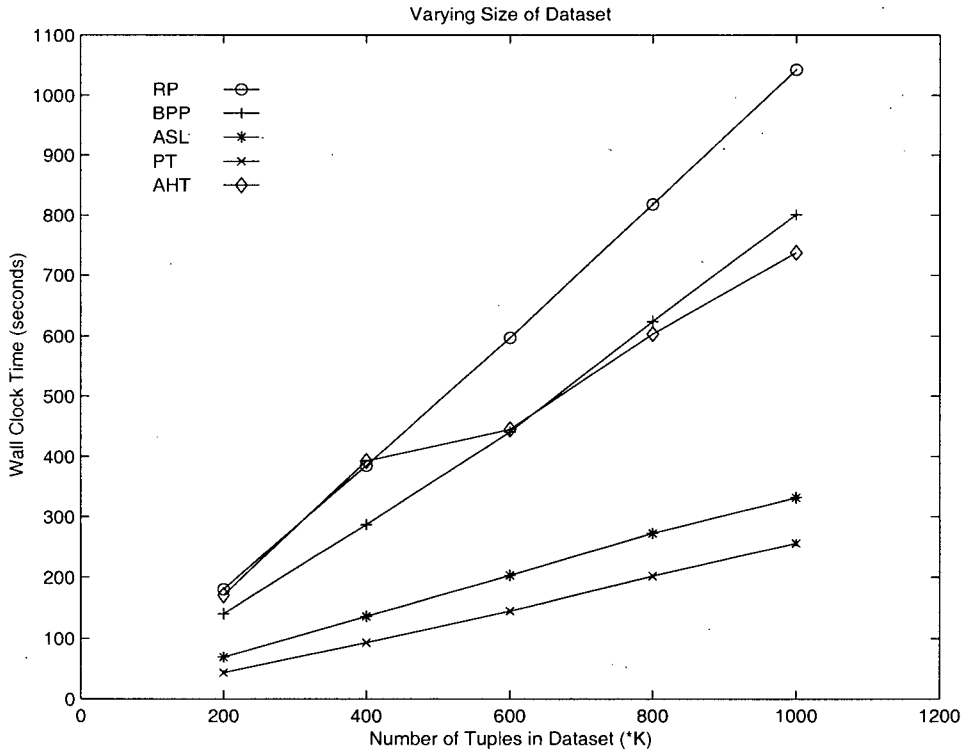


Figure 4.3: Results for varying the dataset size

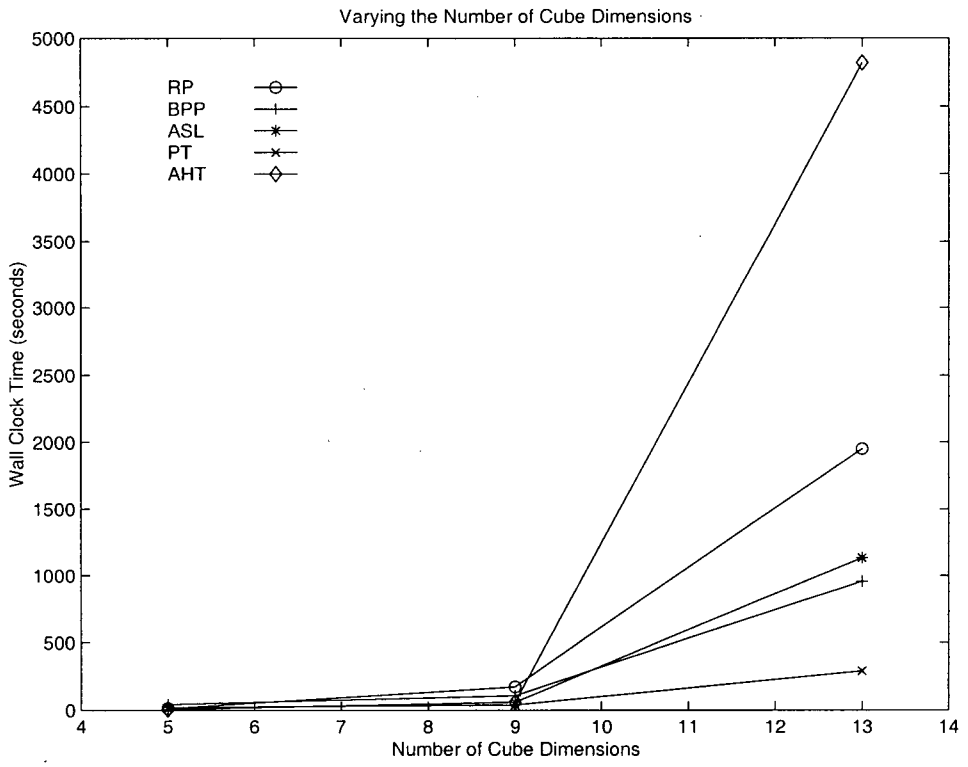


Figure 4.4: Results for varying the Number of Cube Dimensions

Unlike other algorithms, AHT scales unpredictably with problem size. On one hand, this is because collision within a bucket tends to happen more often, as more and more cells are maintained by hash tables. This damages AHT's performance severely. On the other hand, the data distribution in the raw data set dramatically affects how many collisions may occur. This leads to inconsistent scalability in AHT.

4.6 Varying the Number of Dimensions

Figure 4.4 shows the effect of increasing the number of dimensions on each algorithm. The wall clock time increases dramatically as the number of dimensions' increases, because the number of cuboids grows exponentially with dimension size.

For example, the 13-dimensional CUBE has 8,192 cuboids.

The scalability of AHT with CUBE dimensions is the worst of all the algorithms. In fact, in our testing, when the number of CUBE dimensions is set as 13, the hash table size was fixed as ten times the size of the input data set, that is ten times larger than that in the baseline configuration. Even then, AHT's performance is very poor. There are two main reasons contributing to this effect. First, as high dimensional CUBE needs to be computed, a large number of cells need to be maintained in the hash table. This introduces a great amount of collisions within in buckets during insertion and searching operations. Second, since the size of the hash table is fixed, the index bits assigned to each CUBE attribute are far from adequate to appropriately collapse the hash table when subset affinity is applicable. If the data set is skewed on some CUBE attributes, the hash function behaves even poorer.

The relative performance for the other four algorithms remains the same except for ASL, where for thirteen dimensions it stops being better than BPP. ASL is affected more than other algorithms because of its comparison operation. The comparison operation used to search and insert cells into the skip list 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.

Figure 4.4 also shows that when the number of dimensions is small, RP, ASL, AHT and PT all give similar performances. Because the size of the output is small for a small number of dimensions, the simple RP algorithm can keep up to the others.

4.7 Varying the Minimum Support

Figure 4.5 shows the effect of increasing the minimum support. As the minimum support increases, there is more pruning, and as a result, less I/O. The total output size for the algorithms given in Figure 4.5 starts at 469Mbyte for a support of

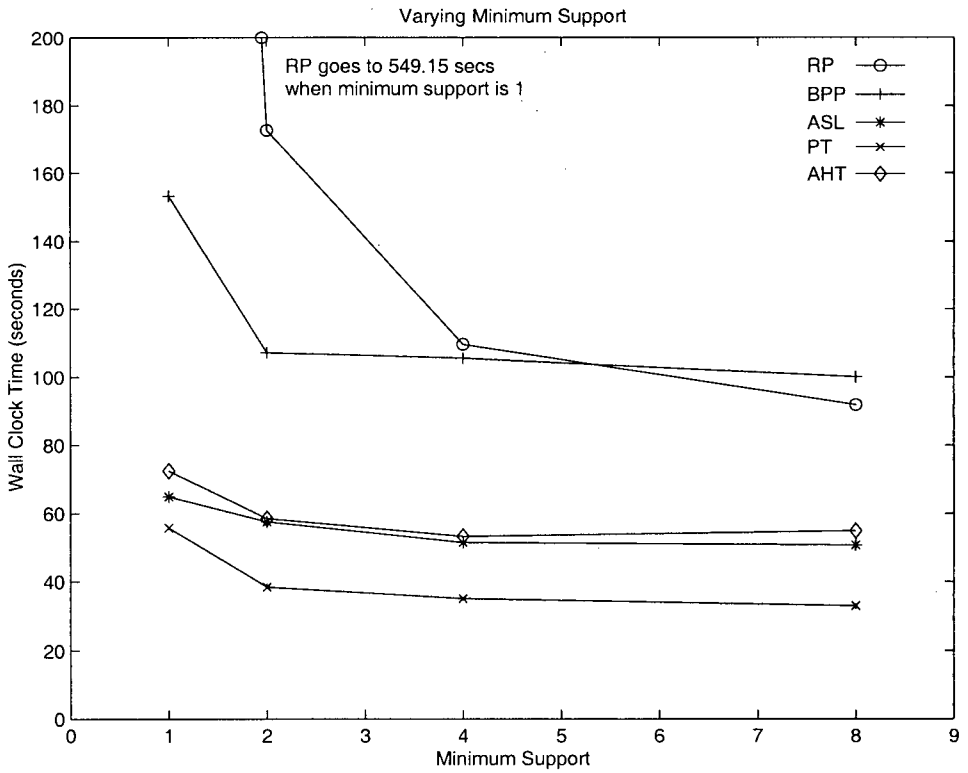


Figure 4.5: Results for varying the minimal support

one, 86Mbyte for a support of two, 27Mbytes for a support of four, and 11Mbytes for a support of eight. After eight, very little additional pruning occurs. Except between one and two, the output size does not appear to have much affect on overall performance. This is surprising since we expected PT to do better as support increased, because more pruning should have led to less computation. The relative flatness of the curve for PT is largely due to the order of the dimensions choosen. For the baseline configuration, the pruning occurs more towards the leaves, where it does not save as much in computation time.

Notice ASL and AHT can not prune during computation; their better performance with higher minimum support is due only to less I/O cost but not to pruning.

4.8 Varying the Sparseness of the Dataset

Figure 4.6 shows the effect of sparseness of the data set on the four algorithms. We consider a data set 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 CUBE. Since the number of tuples in the baseline configuration is fixed, we can vary the sparseness of the data set by choosing smaller dimensions over larger cardinality dimensions. The three data sets chosen for Figure 4.6 consisted of the nine dimensions with the smallest cardinalities, the nine dimensions with the largest cardinalities, and one in between. Note that even for the smallest of the three, there are still about 10^7 possible total cells in the cube.

As shown in Figure 4.6, AHT is apparently more affected by sparseness than the other algorithms. The more CUBE dimensions, the more collisions happen, which badly hamper AHT's performance. If few collisions occurs, as when dimensionality is low, AHT outperforms all others.

AHT and ASL perform well on dense datasets and are more adversely affected by spareness than others. ASL performs well for dense datasets because each cuboid

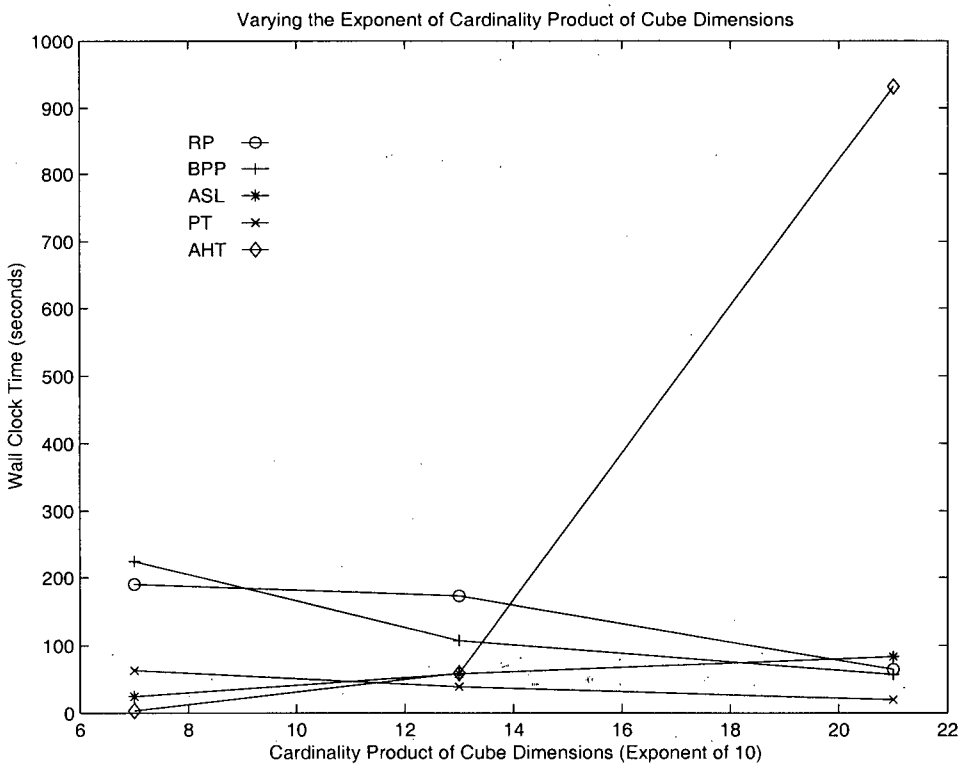


Figure 4.6: Results for varying the sparseness of the dataset

Situations	PT	ASL	RP	BPP	AHT
dense cubes		√			√
small dimensionality (≤ 5)	√	√	√		√
high dimensionality	√				
less memory occupation				√	
otherwise	√√	√			
online support		√			

Figure 4.7: Recipe for selecting the best algorithm

contains relatively few cells, which makes searching or inserting into a skip list relatively fast. The BUC-based algorithms have little opportunity to take advantage of density. In fact, the denser the dataset, the less pruning can be done. 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 worse 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 skip lists.

4.9 Summary

4.9.1 Recipe Recommended

The experimental results shown thus far explores the different parameters affecting overall performance. After careful examination, we recommend the "recipe" shown in Figure 4.7 for selecting the best algorithm in various situations.

It is clear that AHT and ASL dominate all other algorithms when the cube is dense, or when the total number of cells in the data cube is not too high (e.g., $\leq 10^8$). However, AHT is more adversely affected by sparseness and dimensionality. For data cubes with a small number of dimensions (e.g., ≤ 5), almost all algorithms

behave similarly. In this case, RP may have a slight edge in that it is the simplest algorithm to implement. For all other situations, except when the data cube has a large number of dimensions, PT, AHT and ASL are relatively close in performance, with PT typically a constant factor faster than AHT and ASL. For cubes of high dimensionality, there is significant difference among the three, and PT should be used. The last entry in Figure 4.7 concerns online support. This is the topic of the next section.

4.9.2 Further Improvement

There is still room for improvement in some of the algorithms. With the affinity scheduling, the current prefix and subset affinity can be expanded to cooperate with the sorting overlap idea behind the Overlap algorithm, mentioned in Chapter 2. Therefore, even if we can not assign a task to a processor with CUBE dimensions perfectly prefixing the previous task, we can try to assign a task with the longest possible prefix of the previous task. This may improve the performance of ASL.

For AHT, we can attemp more sophisticated hash function instead of the naive MOD hash function currently use. A better hash function may relieve AHT's struggling performance when faced with sparse and high dimensional CUBE computation.

Chapter 5

Online Aggregation

Recall that the CUBE computation is just a precomputation designed to instantly respond to online iceberg queries. However, sometimes a user's query can not be answered by the precomputed CUBE. When the minimum support for the online query is lower than that for the precomputation, it is no longer possible to compute a query, essentially a cuboid, from a precomputed cuboid.

This problem can be solved in two ways. First, we can choose a small minimum support for the precomputation, therefore, most of the queries can be answered by aggregating from a precomputed cuboid. Second, we can simply aggregate from the raw data set to answer an unpredictable query online.

In the following sections, we discuss issues concerning these two separate methods.

5.1 Selective Materialization

CUBE with low constraints usually produces a large body of result for which the computation may take a long time and also may not be saved to disk entirely. To solve this problem, it is natural to consider selecting only one set of cuboids to materialize instead of all the available cuboids. Although our experiments show that in many cases, our parallel algorithms can do well in computing the entire

iceberg-cube query from scratch (e.g., $\leq$ 100 seconds), for truly online processing, selective materialization can still help significantly.

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. If the minimum support was two, as in Figure 4.5, ASL would take approximately sixty seconds to complete the entire CUBE.

The second plan consists of a precomputation stage and an online stage. In the precomputation stage, ASL computes only the leaves 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. In this second stage, ASL can make returns almost immediately; and interestingly, even for the precomputation, it only took fifty seconds for the same example. (The value of fifty seconds was obtained from our additional experiment, not directly from Figure 4.5. The values in Figure 4.5 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.

5.2 Online Aggregate from a Raw Data Set

Besides selective materialization, in this thesis, we also consider computing online aggregates from a raw data set. Thus, we manage to provide a comprehensive solution for the iceberg query problem. Hellerstein, Haas and Wang proposed an online aggregation framework [11], in which a sampling technique is applied for instant response and further progressive refinement. We took this framework for our online aggregate algorithm to allow a user to observe the progress of a query and dynamically direct or redirect the computation. In the case of an iceberg query, the user would see a rough initial cuboid which would become more accurate as more tuples are processed.

Like ASL, we took a skip list as the fundamental data structure, making it possible to construct a cuboid by incrementally inserting tuples into the skip list. Each tuple can therefore be handled independently. In terms of incremently building a cuboid, the hash tables used in AHT provides a good alternative. However, since its performance is too sensitive to dimensionality and data sparseness, as viewed in Chapter 4, a hash table does not make a good data structure for the online aggregate algorithm. The array based algorithms, RP, BPP and PT, are also difficult to be extended to handling online issues, mainly because an array does not efficiently support incrementally insertion. Once query results from new data are computed, they then have to be merged with the results from the old data. Merging operations introduce additional overhead and do not support parallelism well. In fact, the online advantages of ASL over other algorithms was one of the main motivations for its development. In the following section, we present our Parallel OnLine aggregation algorithm(POL).

5.3 Parallel Online Aggregation

5.3.1 Data Partitioning and Skip List Partitioning

Online aggregation implies only one group-by need be computed. Usually, computing one group-by is not time consuming. The computation is much smaller than computing CUBE. To necessitate parallel computation, we assume the raw data set is huge, shown in two aspects. First, a raw data set is range-partitioned across processors without any sorting. If there are n processors in a cluster, n partitions, R_1 to R_n, are produced; processor j gets R_j. Second, neither a processor can load its local data partition entirely into its main memory. A processor has to proceed the computation step by step; at each step, one block of data from its local data partition is loaded and computed. The data block is in fact a sample taken from the unprocessed part of the processor's local data partition.

	Located on P_1	Located on P_2	Located on P_3	Located on P_4
Passed to P_1	$(Chunk_{11})$	$(Chunk_{12})$	$(Chunk_{13})$	$(Chunk_{14})$
Passed to P_2	$(Chunk_{21})$	$(Chunk_{22})$	$(Chunk_{23})$	$(Chunk_{24})$
Passed to P_3	$(Chunk_{31})$	$(Chunk_{32})$	$(Chunk_{33})$	$(Chunk_{34})$
Passed to P_4	$(Chunk_{41})$	$(Chunk_{42})$	$(Chunk_{43})$	$(Chunk_{44})$

Table 5.1: Task Array for 4 Processors

In order to utilize all available machines in a cluster, POL range-partitions a skip list to n partitions as well, where n is the number of processors. Each processor, therefore, maintains only one skip list partition. POL determines boundaries of the skip list partitions assigned to different processors at the beginning of its computation through sampling. Afterward, a processor is only responsible for searching or inserting cells into its skip list partition as delimited by boundaries.

As a processor scans its local data partition, since it is unsorted, the processor finds tuples which should be inserted into skip list partitions maintained by other processors. In such a case, the processor then passes the tuples to other processors appropriately. If there are n processors in the cluster, one processor might pass (n-1)/n of its local data to other processors. The overhead from data communication is then introduced.

5.3.2 Task Definition and Scheduling

As mentioned above, POL proceeds with computation step by step. Within a step, each processor computes a block of data, and data commutation takes place among processors when necessary. POL guarantees that one block of data is loaded only once. Only after all processors complete computing on the tuples in this block, does the loading processor discard the block and move to the next step. Therefore, processors proceed with their computation synchronously, and synchronizations happen amongst processors between every two steps.

Tasks are defined in POL for each step, that is, between synchronizations.

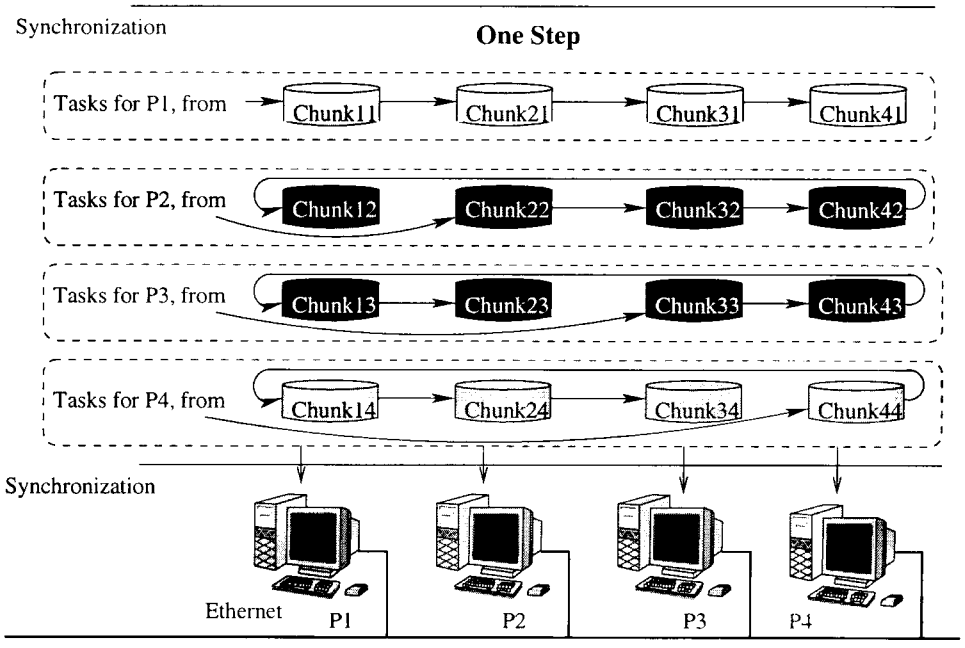


Figure 5.1: Tasks Assignment in POL

Suppose at one step, after the processor P_i loads in a data block from its local data partition R_i, it groups the tuples in the block into n chunks, $Chunk_{1i}$ to $Chunk_{ni}$, according to the partition boundaries set for the skip list partitions, where n is the number of processors. Note that $Chunk_{ji}$ indicates a chunk, which although located in processor P_i, will be passed to processor P_j to maintain P_j's skip list partition. Therefore, for P_i, all but one chunk($Chunk_{ii}$) are passed over network and checked by other processors. For a cluster with 4 processors, the task array created for one step is shown in Table 5.1.

Since there are n processors and each processor has n chunks, $n \times n$ chunks are produced in total. These chunks correspond to $n \times n$ tasks, indicated as task($Chunk_{ji}$)(both i and j from 1 to n). Task($Chunk_{ji}$) is the computation based on chunk $Chunk_{ji}$. Notice that at each step, tasks have to be redefined. Tasks in different steps are separately scheduled.

Like some of CUBE algorithms, in POL, a manager responsible for task

scheduling, and many workers responsible for computing(computing aggregations in this case). The manager initially assigns a number of tasks to each processor. However, once a processor finishes its assigned tasks, it can then help other processors finish their tasks.

Originally, processor P_i, are assigned task($Chunk_{ij}$) (j is from 1 to n). For example, with four processors, P_2 is required to finish task($Chunk_{21}$), task($Chunk_{22}$), task($Chunk_{23}$) and task($Chunk_{24}$). For some of these tasks, P_2 has to fetch appropriate chunks located on other processors. P_2 then needs $Chunk_{21}$ on P_1 to finish task($Chunk_{21}$), $Chunk_{23}$ on P_3 to finish task($Chunk_{23}$), $Chunk_{24}$ on P_4 to finish task($Chunk_{24}$) and local $Chunk_{22}$ to finish task($Chunk_{22}$). The sequence for processor P_i to compute its assigned tasks is this: from task($Chunk_{ii}$) to task($Chunk_{in}$), it then wraps back from task($Chunk_{i1}$) to task($Chunk_{i(i-1)}$). This sequence maximizes the possibility of each processor working on data located on different processors at one time, thus reducing the possibility of a burst of data requests happening on a particular processor. Figure 5.1 illustrates the original task assignment in POL for a computing environment consisting of 4 processors.

To balance the load, a processor is allowed to offload waiting tasks from busy processors after it has finished its own assigned tasks. The manager tries to assign to it those untouched tasks that the processor keep the input data chunk in local. The processor then compute a new skip list for the task. Once it finishes this task, or gets a data request from the processor responsible for the task, it passes the skip list it has already built on to that processor. Then, that processor merges the skip list with its local skip list partition. Apparently, this method of task scheduling does not introduce additional data communication overhead.

To provide a constant update of query results, a set timer periodically gives response back to the user. Whenever the timer expires, the manager collects results from all workers, displays the results on screen, and resets the timer. If the user wishes to discontinue the computation, he or she can interrupt it at any point.

1. Algorithm POL
2. INPUT: Range-partitioned data sets, each located on one processor (processor P_i has partition R_i)
   GROUP BY dimensions $\{A_1, A_2, \ldots A_m\}$ and minimum support Spt
3. OUTPUT: The iceberg query results
4. ONLINE AGGREGATE:
5. The manager takes a sample, and determines the boundaries of skip list partitions assigned to each processor
6. **parallel do**
7. **while (not all data has been processed)**
8. if (worker processors P_i)
9. loads in one block the samples from its local partiton which have not been processed
10. it then groups the samples into n chunks, $Chunk_{1i}$, to $Chunk_{ni}$
11. calls online-slave($Chunk_{1i}, \ldots, Chunk_{ni}, Spt, A_1, \ldots, A_m$)
12. if(the manager)
13. defines $n \times n$ tasks, each for one chunk on workers
14. schedules the tasks, as described in Section 5.3.2
15. synchronize
16. **end while**
17. **end do**
18. Subroutine online-slave($Chunk_{1i}, \ldots, Chunk_{ni}, Spt, A_1, \ldots, A_m$)
19. gets a task from the manger; if there is no uncompleted task at the manager, return
20. if the task is Task($Chunk_{ij}$)
21. asks processor P_j for the chunk $Chunk_{ij}$
22. updates the local skip list based on $Chunk_{ij}$
23. if the task is Task($Chunk_{ji}$)
24. computes a new skip list from the chunk $Chunk_{ji}$
25. sends the skip list to P_j, then P_j merges it into its local skip list parititon.
26. during processing, if any request comes from another process asking for a data chunk, sends it to the processor
27. during processing, if any request comes from the manager for current result, estimates current minimum support, collects result and send them to the manager
28. during processing, if any request comes from the manager for stopping the computation, return

Figure 5.2: A Skeleton of the POL Algorithm

A skeleton of algorithm POL is shown in Figure 5.2.

5.4 Exerimental Evaluation

The testing environment for POL is similar to that for the CUBE algorithms, except that we based our experiments on a larger weather data set, which contains 1,000,000 tuples. Although the data set is larger, it has the same number of dimensions as the smaller one used for testing the CUBE algorithms.

We focused on the following issues in POL during the experiments:

- scalability with the number of processors;
- scalability with the buffer size on each processor.

5.4.1 Varying the Number of Processors

Figure 5.3 shows the performance of POL with different numbers of processors. In testing, a 12-dimensional iceberg query was answered online. The minimum support was set as 2 and the buffer on each processor was set to contain 8000 tuples at each step. The computation created a huge skip list with 924,585 nodes.

The performance of POL was tested on three clusters of machines:

- Cluster1 consists of eight 500MHz PIII processors with 256M of main memory connected by an Ethernet network;
- Cluster2 consists of eight 266MHz PII processors with 128M of main memory connected by an Ethernet network;
- Cluster3 consists of eight 266MHz PII processors with 128M of main memory connected by a higher speed network, Myrinet, which is approximately three times faster than the Ethernet used in the first two clusters.

Data communication among worker processors is the main factor affecting POL's performance. If the data distribution is uniform, for each processor nearly

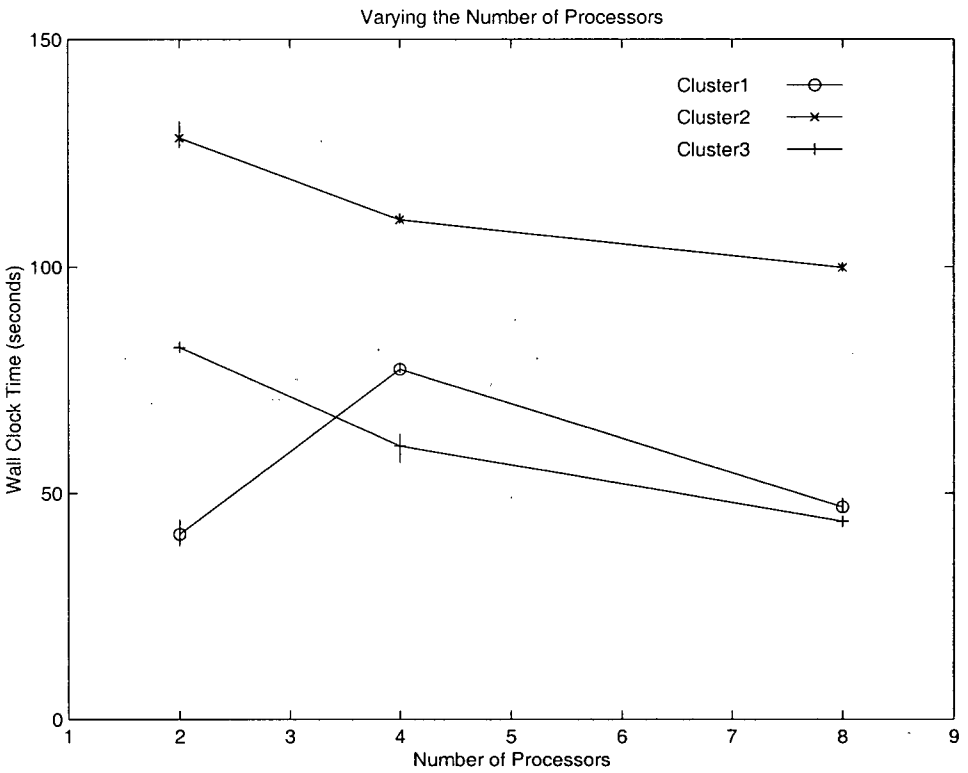


Figure 5.3: POL's Scalability with the Number of Processors

$(n-1)/n$ of data needed are located on other processors, where n is the number of processors. Apparently, the higher n is, the more data needs to be transfered over the network. However, adding more machines decreases the computations carried out at each processor because the work load is shared. Therefore, whether we can achiever better overall performance with more processors or not remains uncertain. It largely depends on how much the computation decreases or the communcation increases on each processor, and which is the dominating factor. Generally, more time spent on computation *versus* the less spent on communication, the better performance can be achieved.

Computing high dimensional queries always implies more computation because a large skip list needs to be maintained. Therefore, we can conclude that POL is feasible especially for computing high dimensional queries.

Figure 5.3 shows the speedup achieved on Clusters2 and Cluster3 is better than on Cluster1, mainly because the computation on the clusters of slow machines takes up more total run time than on the cluster of fast machines.

Concerning load balancing, dynamic offloading from other busy processors can balance uneven load resulted from unevenly distributed data among processors. However, if both the skip list partitioning and the data distribution are uneven, the load may be poorly balanced. Fortunately, in our testings, this adverse situation did not arise.

5.4.2 Varying the Buffer Size

Buffer size limits the amount of data processed at each step. The larger the buffer size, the fewer steps are needed in POL, and thus, less synchronizations and less sampling happens between steps. Usually, synchronization and sampling mean the introduction of overhead. Therefore, as shown in Figure 5.4, as the buffer size increases, performance improves.

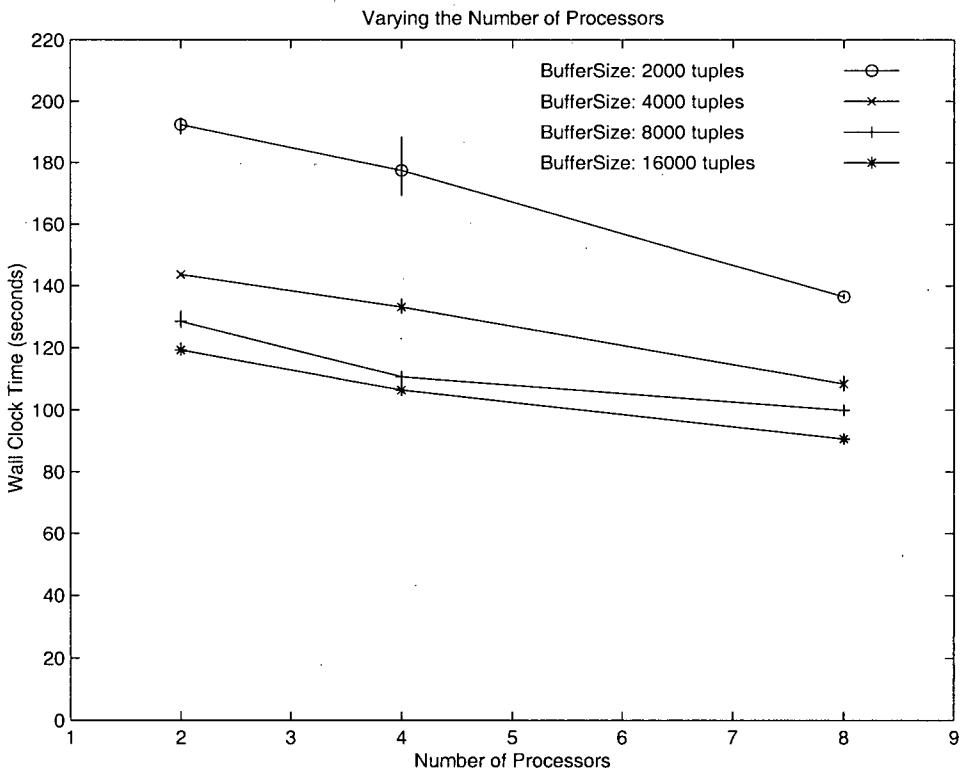


Figure 5.4: Scalability with Buffer Size

Chapter 6

Conclusion

In this thesis we discuss a collection of novel parallel algorithms we developed directed towards online and offline creation of CUBE to support iceberg queries.

We evaluated the CUBE algorithms, RP, BPP, PT, ASL and AHT, across a variety of parameters to determine the best situations for use. RP has the advantage of being simple to implement. However, except for cubes with low dimensionality, RP is outperformed by the other algorithms. BPP is also outperformed; but BPP reveals that breadth-first writing is a useful optimization. As an extension of BPP, PT is the algorithm of choice in most situations. There are, however, two exceptional situations where ASL and AHT are recommended. ASL and AHT are more efficient for dense cubes, whereas ASL supports sampling and progressive refinement especially.

For the online aggregation, we tested our algorithm, POL, for aggregating online over a large data set. Experiments revealed that POL behaves well in a cluster of machines connected with high speed networks, and is valuable in answering high dimensional online queries which require more time to complete computation.

In future work, we would investigate how the lessons we have learned regarding parallel iceberg query computation can be applied to other tasks in OLAP computation and data mining. These include (constrained) frequent set queries [24],

and OLAP computation, taking into account correlations between attributes.

Bibliography

[1] M.J.A. Berry and G. Linoff. Data Mining Techniques: For marketing, Sales, and Customer Support. John Wiley & Sons, New York, 1997

[2] 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.

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

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

[5] 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.

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

[7] 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.

[8] 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.

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

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

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

[12] 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.

[13] YiHong Zhao, Prasad Deshpande, and Jeffrey F. Naughton An Array-based algorithm for simultaneous Multidimensional aggregates.

SIGMOD Conference 1997, pp. 159-170

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

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

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

[17] 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.

[18] 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.

[19] Soroush Momen-Pour Parallel Computation of Data Cubes MSc. Thesis, University of British Columbia, Computer Science Dept., 1999.

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

[21] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R.Ramakrishnan and S. Sarawagi On the Computation of Multidimensional Aggregates. in*Proc. 1996 VLDB*, pp. 506-521.

[22] W. Pugh. Skip Lists: a Probabilistic Alternative to Balance Trees. In *Communications of the ACM* 1990.

[23] Eur-Hong (Sam) Han, George Karypis, Vipin Kumar Scalable Parallel Data Mining for Association Rules Proceedings of the ACM SIGMOD international conference on Management of data May 11 - 15, 1997, Tucson, AZ USA

[24] 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.