2008

Multi-Dimensional Partitioning in BUC for Data Cubes

Kenneth Yeung
San Jose State University

MULTI-DIMENSIONAL PARTITIONING IN BUC FOR DATA CUBES

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements of the Degree

Master of Science

by

Kenneth Yeung

November 2008

SAN JOSÉ STATE UNIVERSITY

The Undersigned Project Committee Approves the Writing Project Titled

MULTI-DIMENSIONAL PARTITIONING IN BUC FOR DATA CUBES

by
Kenneth Yeung

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Teng Moh,	Department of Computer Science	Date
Dr. Mark Stamp,	Department of Computer Science	Date
Prof. Frank Butt,	Department of Computer Science	Date

ABSTRACT

MULTI-DIMENSIONAL PARTITIONING IN BUC FOR DATA CUBES

by Kenneth Yeung

Bottom-Up Computation (BUC) is one of the most studied algorithms for data cube generation in on-line analytical processing. Its computation in the bottom-up style allows the algorithm to efficiently generate a data cube for memory-sized input data. When the entire input data cannot fit into memory, many literatures suggest partitioning the data by a dimension and run the algorithm on each of the single-dimensional partitioned data. For very large sized input data, the partitioned data might still not be able to fit into the memory and partitioning by additional dimensions is required; however, this multi-dimensional partitioning is more complicated than single-dimensional partitioning and it has not been fully discussed before. Our goal is to provide a heuristic implementation on multi-dimensional partitioning in BUC. To confirm our design, we compare it with our implemented PipeSort, which is a top-down data cubing algorithm; meanwhile, we confirm the advantages and disadvantages between the top-down data cubing algorithm and the bottom-up data cubing algorithm.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF TABLES (CONT'D)

LIST OF FIGURES

LIST OF FIGURES (CONT'D)

1. Introduction

"Data warehousing is a collection of decision support technologies, aimed at enabling the knowledge worker (executive, manager, analyst) to make better and faster decisions" [3]. Since time is a critical factor in making business decisions, one of the challenges of data warehousing is to provide a responsive summary view out of the tremendous data that is collected from daily operations. A traditional relational database is not capable of doing so because of the lack of performance and its incapability of serving basic data warehousing operations such as rolling-up and drilling-down operations [4]. Since Gray et al. proposed a data cube structure to solve the problem, data cube generation has become one of the active researches in the online analytical processing technology.

A data cube is "a data structure that consists of the results of group-by aggregate queries on all possible combinations of the dimension-attributes over a fact table in a data warehouse" [5]. A fact table can be visualized as a two-dimensional array filled with data values. Each row represents a tuple and each column represents an attribute of the tuple. An attribute can be categorized into two types: dimension and measure. A dimension is a data value that describes a tuple of a fact table and this value is not quantifiable; on the other hand, a measure is a quantifiable data value that describes a tuple of a fact table and we can apply aggregated functions on this type of attributes. For example, a data warehouse of a car dealer stores millions of tuples for car sales (Figure 1). Each

tuple consists of the following attributes: manufacture, model, year, color, and sale price. The dimensions of this tuple are the manufacturer, model, year and color; whereas, the measure of this tuple is the sale price, which can be aggregated to find the average, minimum, maximum, and total of the sale price. The results of a group-by aggregate query are saved in a table known as view, which is labeled by the group-by dimensions. In the car sales example, a view, which label is { manufacturer, year }, contains the summary of the sale prices in terms of manufacturer and year. If a fact table has D number of dimensions, the data cube for this fact table will consist of 2^D number of views.

Dimension Measure

Manufacturer	Model	Year	Color	Sale Price
Honda	Civic	2008	White	14,000
Honda	Accord	2001	Silver	12,400
Toyota	Prius	2007	Yellow	20,100
Mazda	RX7	2005	Red	22,800
Ford	Focus	2006	Blue	11,700

Tuples

Figure 1. A sample fact table in the data warehouse of a car dealer

A data cubing algorithm is an algorithm that generates a data cube from a fact table. A data cube can be visualized as a cube lattice, which is an acyclic graph (Figure 2). Each node of the lattice represents a view of the data cube. There is an edge from node A to B if and only if the number of dimensions of view A is exactly one more than the number of dimensions of view B and the dimensions of view A is the superset of the dimensions of view B. Having this relationship will allow us to compute view B from A. If we arrange the views in a

way that a view is always under its parent and ancestors, the highest node of the lattice will be the finest view that consists of all dimensions; the lowest node of the lattice will be the coarsest view that consists of an empty set of dimensions.

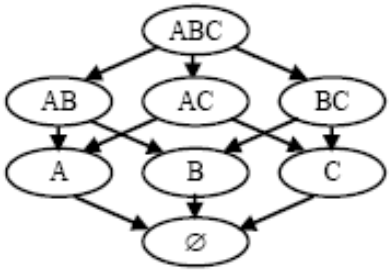

Figure 2. A sample cube lattice [5]

There are primarily two computation styles of data cubing algorithms. A top-down data cubing algorithm generates a data cube from the fine views to the coarse views. The idea is to share the sorting effort by computing a view using the results of its parent. Algorithm GBLP [4], PipeSort [10], Overlap [1] and PartitionCube [8] are categorized as top-down data cubing algorithms. Similarly, a bottom-up data cubing algorithm generates a data cube from the coarse views to the fine views. These kind of algorithms traverse the lattice in depth-first fashion. A bottom-up data cubing algorithm is good at generating an Iceberg-cube, in which the aggregated measures are greater than some value known as minimum support or minsup. A bottom-up algorithm takes advantage of pruning any unqualified tuples in coarse views as early as possible [6]. Algorithm BUC [2], BU-BST [11], and CURE [5] are categorized as bottom-up data cubing algorithm.

The main goal of our research is to provide a heuristic implementation on multi-dimensional partitioning in BUC. One of the limitations of BUC is memory requirement. When input data cannot fit entirely into memory, many literatures suggest partitioning the data by a dimension and run BUC on each the partitions [2]. For clarity, we define single-dimensional partitioning to be an operation that BUC partitions the input data by one dimension only. Two-dimensional partitioning will be an operation that BUC partitions the resulting partition from single-dimensional partitioning. Tuples in such partition will share the same values on two dimensions. For very large sized input data, the single-dimensional partition might still not be able to fit into the memory and partitioning by additional dimensions is required. Unfortunately, we cannot simply apply the same strategy as we do in single-dimensional partitioning on multi-dimensional partitioning because this will lead us to generate an incomplete data cube. To our knowledge, no literature has addressed this issue and laid out a practical implementation in detail for it. Our contribution is to identify the challenges in multi-dimensional partitioning and provide a heuristic implementation for the partitioning.

We have implemented PipeSort to confirm our implementation on multi-dimensional partitioning and our evaluation also gives us an opportunity to compare the top-down data cubing algorithm with the bottom-up algorithm. We analyze the two algorithms by processing input data with different properties. Through our evaluation, we have confirmed the advantages and disadvantages

between the two algorithms. To our knowledge, there is not a side-by-side comparison between these two algorithms.

The rest of this paper is organized as below. Section 2 provides a review of PipeSort and BUC. We describe our approach on handling multi-dimensional partitioning in BUC in section 3, followed by the design of our implementation in section 4. We then provide our evaluation in section 5 and our discussion in section 6. Section 7 discusses the possible directions of our research in the future. Finally, we conclude our research in section 8.

2. Review

This section provides a review of PipeSort and BUC.

2.1. PipeSort

PipeSort is a top-down data cubing algorithm introduced by Sarawagi, Agrawal, and Gupta in 1996 [10]. For each edge E_{xy} from view X to view Y of a lattice, PipeSort assigns two computation costs: A_{xy} and S_{xy}. Cost A_{xy} is the computation cost to generate view Y from X by simply reading or scanning the entire tuples of view X. Edges with this cost are called pipeline edges. Cost S_{xy} is the computation cost to generate view Y from X with the sorting in the order of the dimensions of view Y. Edges with this cost are called sort edges. PipeSort goes through the lattice, level by level, to determine the minimum cost for each view to be generated from its parent and removes the edges that come from the

rest of its parents. As a result, each view will have exactly one edge coming from its parent and we call this modified lattice as a processing tree (Figure 3a). A processing path is a set of views such that the views are connected by a set of edges and no two views share the same parent on a processing tree. PipeSort decomposes a processing tree into a set of processing paths, which visit every view on the tree (Figure 3b). For each processing path, PipeSort determines the common order of dimensions of all the views and sorts the source view based on this order such that PipeSort can aggregate the source view in pipeline fashion. It allocates memory in the size of a tuple for each view on the processing path and the aggregation can be performed for all the views simultaneously while the sorted data is being read.

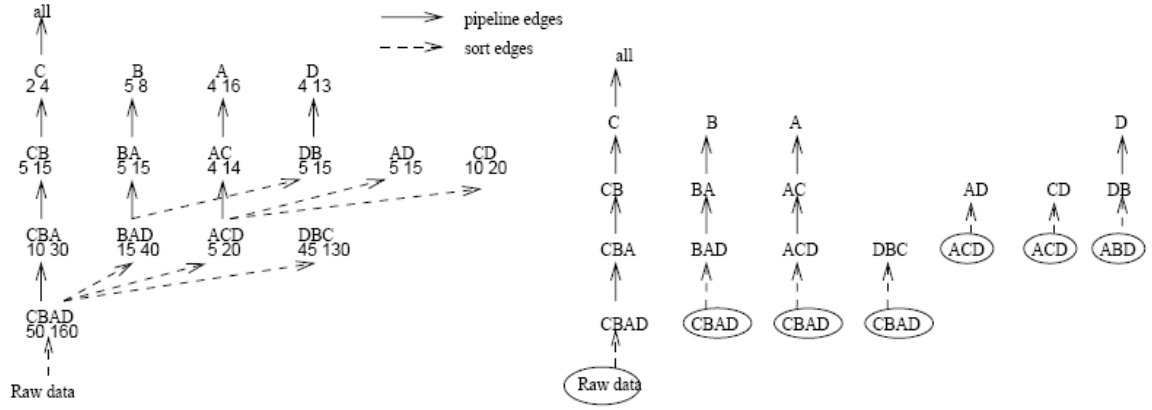

Figure 3. A sample processing tree and processing paths [10]

The disadvantage of PipeSort is that it doesn't scale well as the number of dimensions (D) increases. According to [8], the number of sorting required by

PipeSort has a lower bound at $\binom{D}{\frac{D}{2}}$, which is exponential in D. A simple proof of the lower bound is that the maximum number of views at a level of a lattice is $\binom{D}{\frac{D}{2}}$ and each of the views is visited by a distinct processing path. When a fact table is in spare, many views will not be able to fit inside memory and external sorting is required. This increases a notable amount of I/Os.

2.2. BUC

Algorithm BUC, which stands for Bottom-Up Computation, is a bottom-up data cubing algorithm proposed by Beyer and Ramakrishnan in 1999 [2]. It is designed to generate an iceberg cube, which contains the results of group-by aggregate queries with HAVING(*) > X on all possible combinations of the dimensions of a fact table. In other words, the aggregated measure of all tuples in an iceberg cube must be greater than X, which is known as minimum support or minsup. BUC is good at generating an iceberg cube because of its depth-first fashion on traversing a processing tree. BUC generates a data cube from the coarse views to the fine views and this allows the algorithm to prune any unqualified tuples as early as possible. The advantage of BUC is manifest when a fact table is spare. Many unqualified tuples will be pruned in the early stage and BUC will generate the fine views with less tuples.

The computation logic of BUC is described below. Suppose that we have a fact table with four dimensions: A, B, C and D. The order of the dimensions of each tuple is ABCD; in addition, the domain size of a dimension in lower order is larger or equal to that of the dimension in the higher order. If |X| is the domain size of dimension X, we will have |A| > |B| > |C| > |D|. For the sake of simplicity, we assume that the entire fact table can be fit inside memory. BUC starts from computing the ALL view, which consists of an empty set of dimensions. It then goes through each dimension in ascending order. For each dimension X_i, BUC sorts the tuples with respect to the current dimension and this is known as partitioning by dimension. Tuples in the same partition will have the same value on the specific dimension. For each partition P_i on X_i, BUC aggregates the tuples of the partition and writes the result to view X_i. Instead of moving to the next partition P_{i+1}, BUC performs partitioning by dimension X_{i+1} on P_i and writes the aggregated result to view X_iX_{i+1}. This is a recursive computation and it does not return until it reaches the highest dimension. In our example, after the algorithm computes the ALL view, it partitions A and produces partition a1, a2, a3, and a4 (Figure 4). BUC writes tuple <a1> to view A and partitions a1 by B. It writes <a1, b1> to view AB and partitions a1,b1 by C. Applying the same computation logic, BUC writes the following aggregated tuples in order: <a1, b1, c1>, <a1, b1, c1, d1>, <a1, b1, c1, d2>, <a1, b1, c2>, <a1, b1, c2, d3>, <a1, b2>, etc.

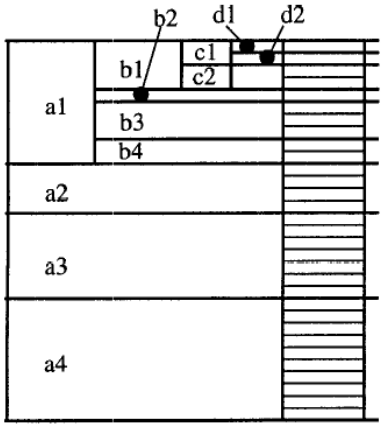

Figure 4. BUC partitioning [2]

When the entire fact table cannot fit into memory, partitioning by a dimension is required. Each partition will be written to disk. We load each partition, one at a time, back to the memory, hoping that the memory can hold the entire partition. We then run BUC on the partition to generate a portion of a data cube. Many literatures have mentioned that partitioning by additional dimensions is required if a single-dimensional partition cannot fit into the memory; however, we haven't seen any publication that describes the logic of the additional partitioning, which is the main goal of our research.

3. Approach

Our approach to handle multi-dimensional partitioning in BUC is to first understand why we cannot simply run BUC on multi-dimensional partitions. Suppose a single-dimensional partition cannot fit into the memory. We partition the data by another dimension, create two-dimensional partitions, and run BUC

on each of the two-dimensional partitions. BUC will not be able to generate the aggregated tuples for the views that project the first dimension but not the second dimension. For example, suppose the input data has four dimensions: A, B, C, and D. After we perform two-dimensional partitioning on AB, we will be able to generate aggregated tuples for the following views from this partition: AB, ABC, ABD, and ABCD. However, we cannot generate tuples for view A, AC, AD, and ACD.

To generate the missing tuples, we modify PipeSort and integrate it with our BUC algorithm. The main difference between the original version and the modified version of PipeSort is that the modified version of PipeSort obtains a set of processing paths from the sub-graph of a lattice. In the original version of PipeSort, we need to generate a set of processing paths that go though every single view of a data cube. The dimension ordering can be different from one path to another. On the other hand, the goal of the modified version of PipeSort is to generate the missing tuples from the results of two-dimensional partitioning. This will work because the missing tuples are always the coarse versions of the tuples that are generated from two-dimensional partitioning. For example, suppose our input data has four dimensions (A, B, C and D). After we perform two-dimensional partitioning on AB and generate the aggregated tuples for each of the partitions, we can obtain the following processing tree by flattening the sub-graph of the lattice (Figure 5). Each node on the tree represents a view. A solid node means that we have already generated the aggregated tuples for this

view by running BUC on the two-dimensional partitioned data; otherwise, the view will be indicated as an empty node. A solid edge means that we can compute the aggregated tuples of a view from its parent without any sorting. A dotted edge means that sorting is required to compute the aggregated tuples of a view from its parent.

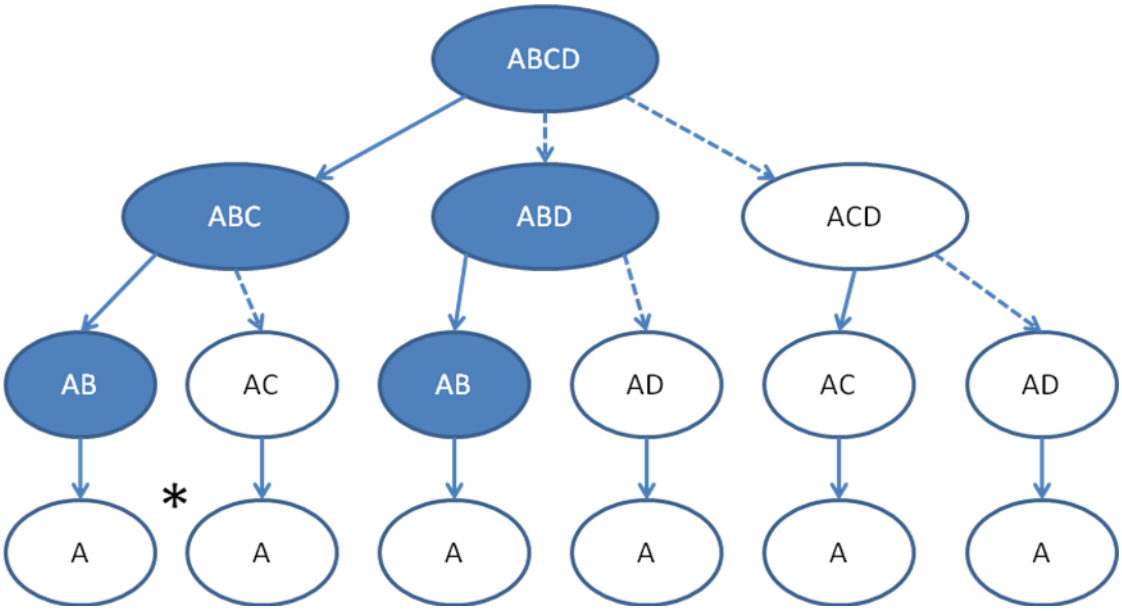

Figure 5. Processing tree for generating missing views

The cost assignment in our modified version of PipeSort is also different to the cost assignment in the original version. Lets pick the graph in Figure 5 as an example. For clarity, we refer missing view to be a set of aggregated tuples for a view that is not generated from two-dimensional partitioning. We can generate the missing view A from the tuples of view AB or view AC. The difference on choosing between view AB and view AC is that the tuples of view AB is already computed but the tuples of view AC are not. Our modified PipeSort will select

view AC since we can compute the missing view A while we are generating the missing view AC. Aggregating the missing view A from the tuples of view AB is cost ineffective because we have to load the tuples of view AB back to memory and this produces additional I/Os. In order to instruct PipeSort to aggregate the missing view A from the tuples of view AC instead of the tuples of view AB, we set the cost of the edge from view AC to view A to be zero. In general, the cost of the edge between two circle nodes will be set to be zero.

After we obtain the optimal set of processing paths, generating the missing views becomes an easy task, which computation logic is similar to the original version of PipeSort. When a single-dimensional partitioned data cannot be fit entirely in memory, we perform two-dimensional partitioning on the partition. During the processing, we determine the boundaries of the aggregated tuples that are generated from two-dimensional partitions. After the processing, we load the aggregated tuples of the views, which are specified in the processing paths, back to memory and generate the missing views.

We have developed two versions of our implementation on multi-dimensional partitioning in BUC. The first version will construct the optimal set of processing paths every time it partitions on a single-dimensional partition; whereas, the second version, which is the improved version, will construct only one set of processing paths for the single-dimensional partitions, which are partitioned by the same dimension. If a fact table has D number of dimensions,

the second version of our implementation will generate at most D number of optimal set of processing paths.

4. Design

We have developed five algorithms to support our research and we will briefly describe the design of each of the algorithms. DataFormatter is a JAVA console application that parses any data in different formats into a binary data file, which becomes the input of our data cubing algorithms. This application is described in section 4.1. In section 4.2, we go though the design of our PipeSort algorithm, which is used in our evaluation. Section 4.3 describes the design of our BUC, which basically follows the design in [2] and supports single-dimensional partitioning. We call this algorithm as BUC1. The design of our implementation on multi-dimensional partitioning in BUC is described in section 4.4 and we call this algorithm as BUC2. BUC3 is the improved and final version of our implementation on multi-dimensional partitioning in BUC and the design is described in section 4.5.

4.1. DataFormatter

DataFormatter is a JAVA console application that parses any data in different formats into a binary data file, which becomes the input of our data cubing algorithms. The resulting data file will have the following properties. 1) Every data value will be replaced with an integer under the following rule. Let X be the integer of value A on a column and Y be the integer of value B on the

same column. Value A is smaller than Value B if and only if X is smaller than Y. 2) Columns in the binary file will be rearranged such that the size of the domain of a column in low order is always larger than or equal to that in high order.

DataFormatter consists of the following components: Main, ColumnInfo, DataParserFactory, DataParser, GenericParser, and CovTypeParser. Main is the main class of this application and the entire execution starts from here. It contains an array of ColumnInfo, which length is in the number of dimensions of the input data file. ColumnInfo consists of two fundamental JAVA collection objects: HashSet and HashMap, which are used to manage the distinct data values of the corresponding column and the mapping between a data value to an integer. Since we would like to support parsing input data file in various formats, we define DataParser as an interface, which allows us to implement different parsers for different data formats. We developed two classes that implement DataParser: GenericParser, which supports parsing files in CSV and CovTypeParser, which supports parsing Forest Covertype data. We adopt the Factory design pattern to construct the corresponding parser and we pass the data format as a parameter to DataParserFactory to get a parser. Figure 6 displays the class diagram of DataFormatter.

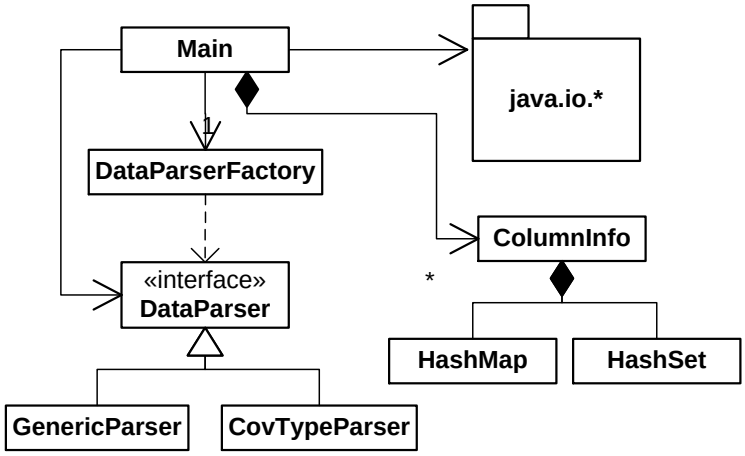

Figure 6. Class diagram of DataFormatter

The execution logic of DataFormatter is described as follows. Given the data format of a data file and its file path, DataFormatter gets the corresponding DataParser from DataParserFactory. For every line that this application reads from the data file, the application inputs the line to the parser that returns an array of Object, each of which represents a data value of a tuple in the data file. We add each value to the HashSet of the corresponding ColumnInfo. After reading the entire data file, we sort each of the HashSets and put the assignment of each distinct data element into a HashMap for each column. We also write the assignments to a file for each column and write statistical information of the data file in a meta file. Lastly, we read the data file again. For every line we read, we get the integer value of each data value from the corresponding HashMap and then write the integer value to the resulting file in binary format. In addition, the order of elements on each line will be based on the size of the column domains in descending order. Figure 7 illustrates the activity diagram of this application.

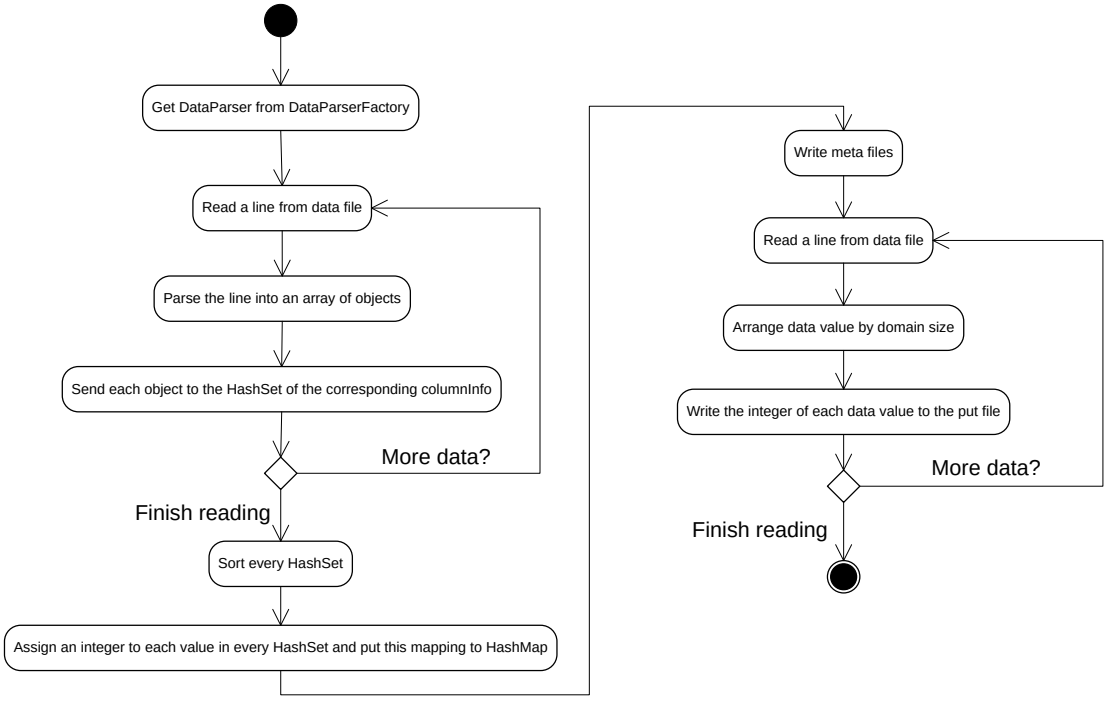

Figure 7. Activity diagram of DataFormatter

4.2. PipeSort

Our design of PipeSort basically follows Sarawagi's design in 1996. Given a binary input file generated from DataFormatter, PipeSort will generate a data cube of a fact table and each view of the data cube will be presented as a file. Let D be the number of dimensions of the input data. There will be 2^D number of files created for the data cube. For simplicity, we will implement only one aggregation function: sum.

Our PipeSort consists of the following basic components: Main, Memory, DataLoader, QuickSort, MergeSort, SizeEstimator, PlanGenerator, and ViewAggregator. Main is the main class, where the execution begins. Memory is

a class that consists of an integer array in fixed length and methods to illustrate a resizable memory table. All data must be loaded into Memory before any in-memory computation such that we can control the memory size of our PipeSort. DataLoader is responsible to load the fact table or generated views into Memory and sort the tuples in ascending order. It will perform external sorting if it cannot load all data into Memory. QuickSort and MergeSort are the algorithms that DataLoader used for in-memory and external sorting. SizeEstimator is a class that calculates the estimated number of tuples of a view by multiplying the size of the dimension domains of the view. The estimation will be used by PlanGenerator, which generates a set of optimal processing paths. For each view on a processing path, a ViewAggregator is used to aggregate tuples and write the results to a file. Figure 8 shows the class diagram of our PipeSort.

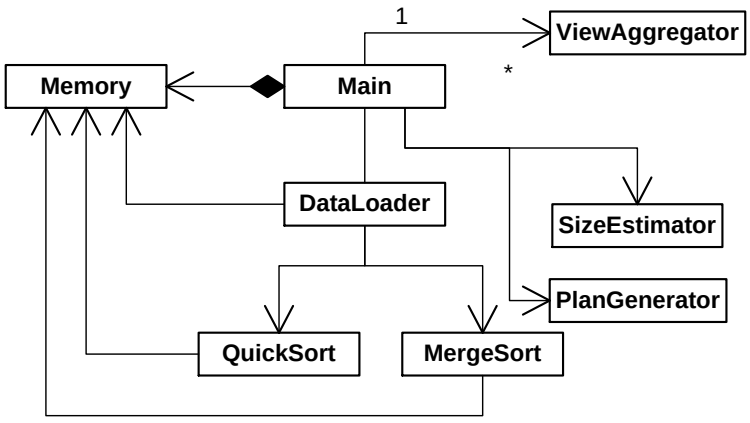

Figure 8. Class diagram of PipeSort

The execution logic of our PipeSort is described as below. Our algorithm starts from initialization on the size of Memory and the basic configuration of our algorithm such as the size of each dimension domain, the total number of dimensions, and the path to the output directory. It then creates SizeEstimator to estimate the size of each view of the data cube and saves the results in an integer array, which is passed to PlanGenerator to generate the optimal set of processing paths. Each processing path is presented by ViewPath, a data structure that contains a pointer to a view and a pointer to the next ViewPath. We will describe ViewPath in detail later. For each ViewPath returned from PlanGenerator, DataLoader loads the first view in the path to Memory and sorts the data in ascending order. PipeSort then asks DataLoader to return every tuple, which will be fed into ViewAggregator to generate tuples for the views specified in the rest of the processing path. Figure 9 shows the activity diagram of our PipeSort.

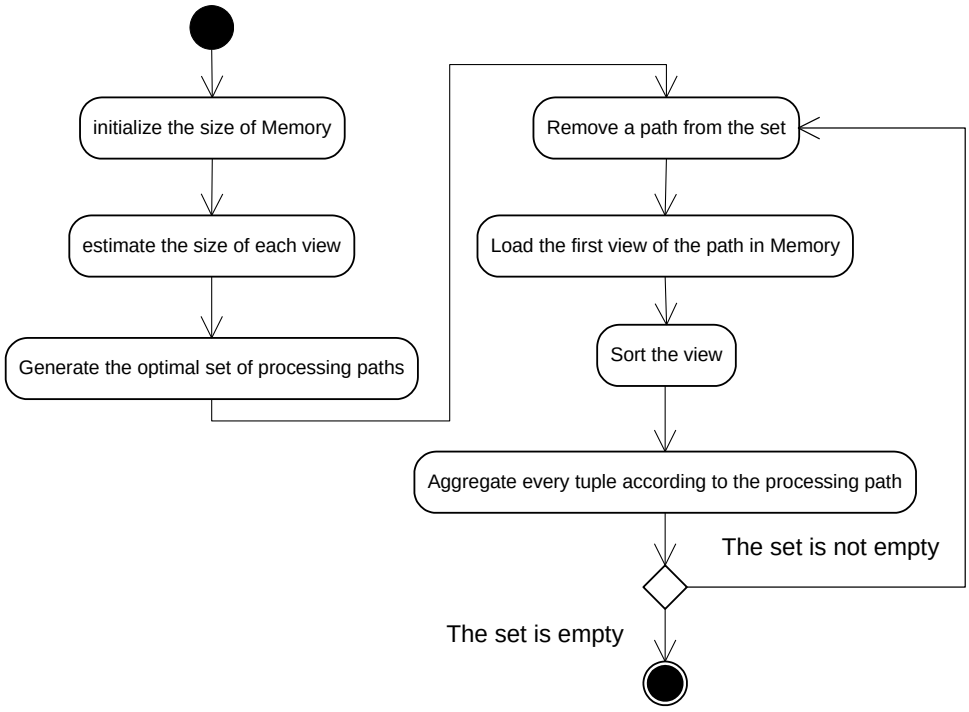

Figure 9. Activity diagram of PipeSort

4.2.1. DataLoader

Given a processing path, DataLoader loads the first element on the path into Memory in a given column arrangement and sort tuples in order. The first element can be the fact table or a file that represents a generated view. Since the second view on the path might not be the prefix of the first element in terms of the order of dimensions, rearranging the dimensions of the data in Memory is required. DataLoader writes the tuples into Memory until either DataLoader finish loading all data of the first element or Memory is full. If Memory can keep all tuples of the first element, DataLoader will sort the tuples by running QuickSort. Otherwise, DataLoader will sort the tuples in Memory by running QuickSort and output them to a temporary file. DataLoader repeats this logic

until it finishes reading all tuples of the first element. At the end, DataLoader will perform MergeSort on the temporary files. The results will be written to a file, which will be loaded back to Memory when it is needed. Figure 10 shows the activity diagram of DataLoader.

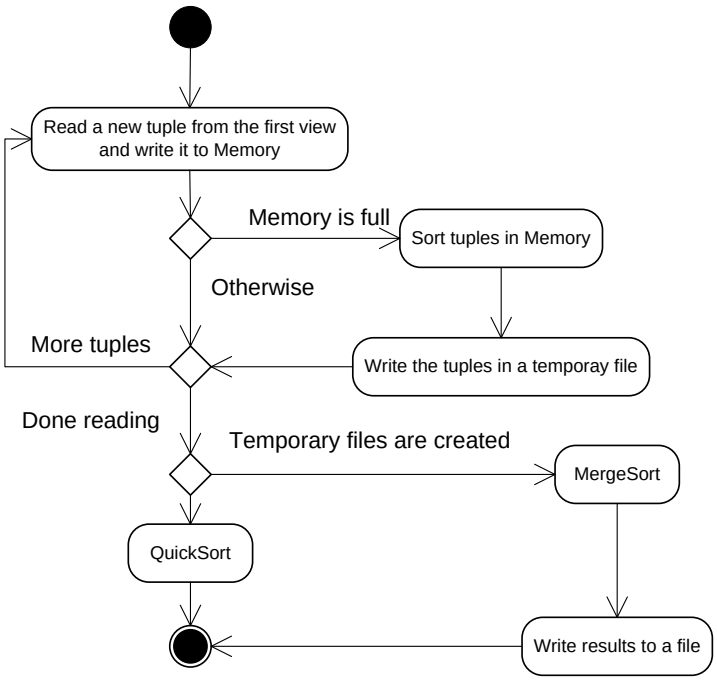

Figure 10. Activity diagram of DataLoader

4.2.2. SizeEstimator

SizeEstimator is a class that estimates the size of each view of a data cube through the mathematical approximations [9]. The reason we chose this approach is that the estimation is simple and fast. Based on the research, we can estimate the size of a view, es(V), by using the following formula:

$$es(V) = ms(V) \times \left[1 - \prod_{i=1}^{|F|} \frac{ms(F) \times d - i + 1}{ms(F) - i + 1}\right]$$

where $d = 1 - \frac{1}{ms(V)}$.

Figure 11. The size estimation formula [9]

In this formula, ms(V) is the maximum size of a view and we define it as the multiplication of the size of the dimension domains of the view. |F| is the size of the fact table. SizeEstimator returns an integer array, which contains the estimated size of each view of the data cube. The array will be used by PlanGenerator in the next step.

4.2.3. PlanGenerator

Given the total number of dimensions, the estimated size of each view of a data cube and the size of Memory, PlanGenerator will generate the optimal set of processing paths for PipeSort to generate a data cube. PlanGenerator uses CubeNode to model a view on a processing tree and uses ViewPath to illustrate a processing path. Each ViewPath represents a view on a processing path and maintains a pointer to the next ViewPath. During the path generation processing, PlanGenerator will encounter the minimum cost matching problem, which will be solved by our implementation of Hungarian algorithm that was described by Papadimitriou and Steiglitz in 1982. Figure 12 shows the class diagram of PlanGenerator.

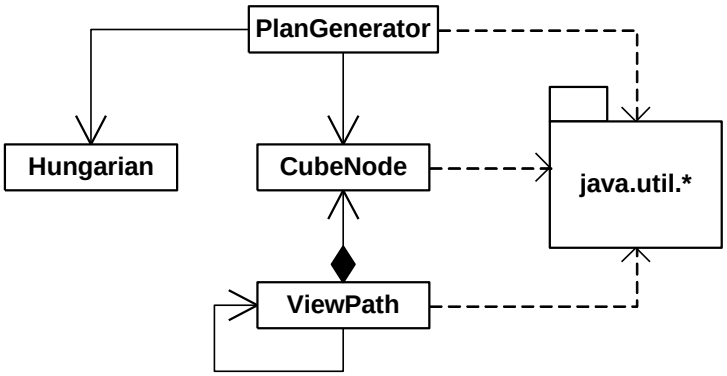

Figure 12. Class diagram of PlanGenerator

The execution logic of PlanGenerator is described as follows. Given a lattice, each of which views is represented by CubeNode, PlanGenerator goes through the views from the bottom level to the top level. For each level k, PlanGenerator works on the views at that level and their parents at the k+1 level. PlanGenerator assigns the corresponding scan cost to every edge from the parents to their children. It then makes k number of copies of the parents and assigns the corresponding sort cost to every edge from the copied parents to their children. A two-dimensional array is used to keep the cost information, where the rows represent the parents and the columns represent the children. PlanGenerator then runs the Hungarian algorithm with the cost table to find the minimum cost assignment for generating the child views from the parent views. However, since the Hungarian algorithm is used to solve the maximum cost matching problem, we need to find the maximum cost in the cost table and subtract it with every cost in the table. When PlanGenerator gets the assignment from the Hungarian algorithm, it removes every edge that goes into the child

views except the one that is mentioned in the assignment. After going through all levels, PlanGenerator will traverse the lattice, which now becomes a processing tree. It then decomposes the tree into processing paths, each of which is represented as ViewPath. Figure 13 shows the activity diagram of PlanGenerator.

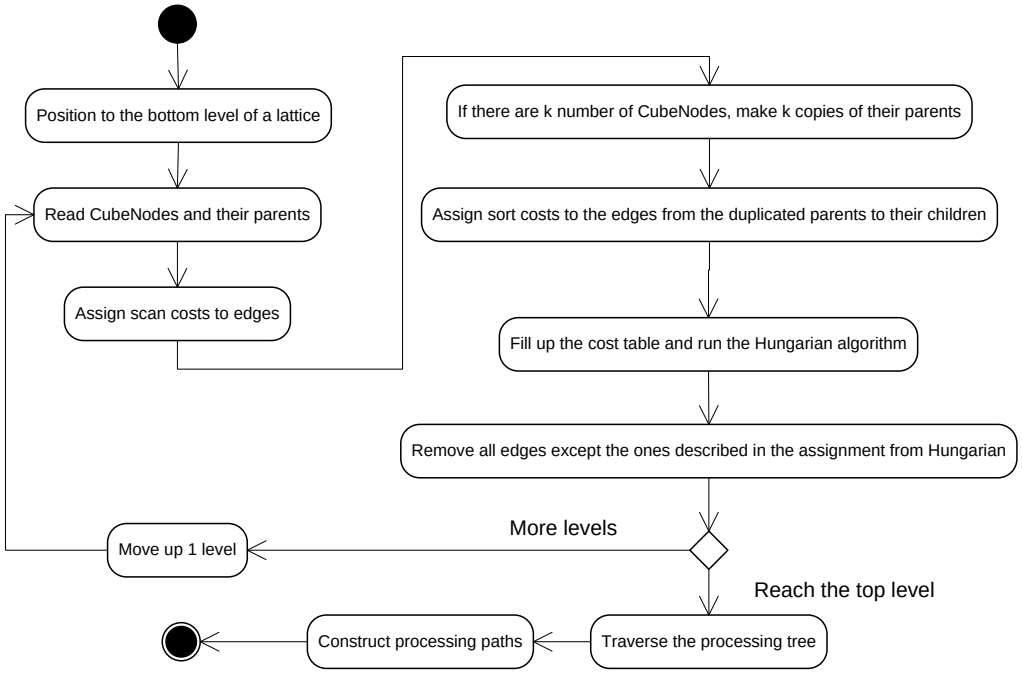

Figure 13. Activity diagram of PlanGenerator

4.2.4. ViewAggregator

ViewAggregator is a class that helps PipeSort to aggregate tuples for a specific view. PipeSort creates an instance of ViewAggregator for every view on a processing path. When PipeSort reads a tuple from DataLoader, it passes the tuple to the ViewAggregators. Since the tuples from DataLoader are sorted in order, it is easy for ViewAggregator to implement the sum aggregation function.

For every tuple that ViewAggregator receives, it extracts the values of the columns that the corresponding view has. If this is the first tuple ViewAggregator has ever read, save it. For the rest of incoming tuples, if the dimension values are the same as the saved tuple, ViewAggregator accumulates the measure of the saved tuple with the measure of the new tuple. Otherwise, ViewAggregator will output the saved tuple to the file that represents the view and then replaces the saved tuple with the incoming tuple. Figure 14 shows the activity diagram of ViewAggregator.

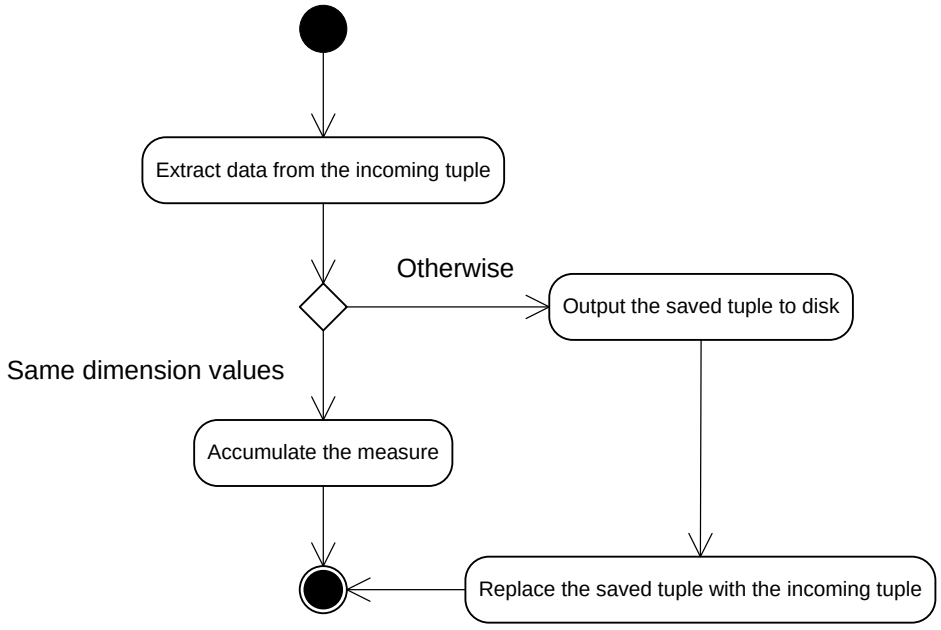

Figure 14. Activity diagram of ViewAggregator

4.3. BUC1

Our design of BUC basically follows the design of Beyer and Ramakrishnan in 1996. Given a binary input file generated from DataFormatter,

our algorithm generates a data cube for the input data. Each view of the data cube will be presented as a file. Let D be the number of dimensions of the input data. There will be 2^D number of files created for the data cube. For simplicity, we will implement only one aggregation function: sum.

We implement three versions of BUC: BUC1, BUC2, and BUC3. The implementation of BUC1 follows the original design of BUC and it supports single-dimensional partitioning. BUC2 is the advanced version of BUC1. We restructure the design of BUC1 and optimize the algorithm on memory and I/O usage. It supports two-dimensional partitioning based on our implementation on multi-dimensional partitioning. BUC3 is the improved version of BUC2 and the final version of our implementation.

BUC1 consists of the following basic components: Main, Memory, DataLoader, DataPartitioner, OutputRec, RecordWriter, and CountingSort. Main is the main class of the algorithm and the execution begins there. Memory illustrates a resizable memory table for the algorithm and allows us to configure the memory usage of the algorithm. Memory is the same component as we discussed in section 4.2. DataLoader is responsible to load the input data into Memory. DataPartitioner divides the input data into numbers of partitions when Memory cannot hold the entire input data. OutputRec is a data class that keeps an integer array for the current aggregated tuple. The length of the integer array equals the total number of dimensions plus one. RecordWriter is a class that

writes the aggregated tuples to files. CountingSort is the countingSort algorithm that sorts the tuples in Memory. Figure 15 shows the class diagram of BUC1.

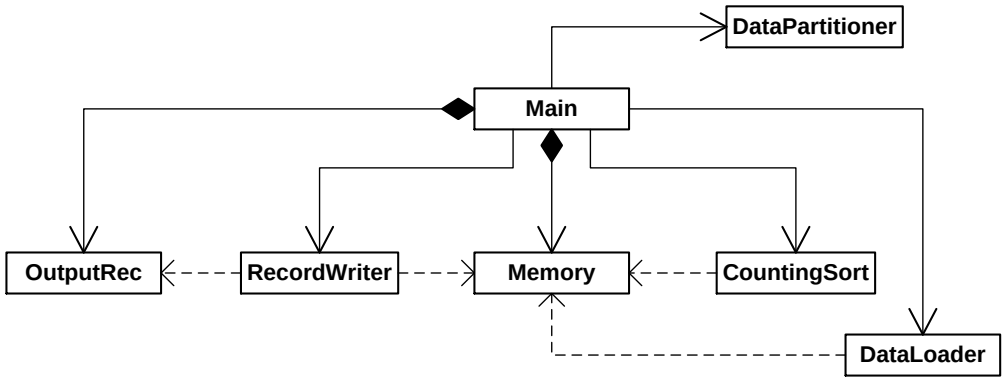

Figure 15. Class diagram of BUC1

The execution logic of BUC1 is divided into two sections. If the input data can fit into Memory entirely, BUC1 executes bucInternal, which is in-memory BUC algorithm that Beyer and Ramakrishnan described in 1996. If Memory cannot hold the entire input data, BUC1 executes bucExternal, which will partition the data and run bucInternal on each partition. Figure 16 shows the global activity diagram of BUC1.

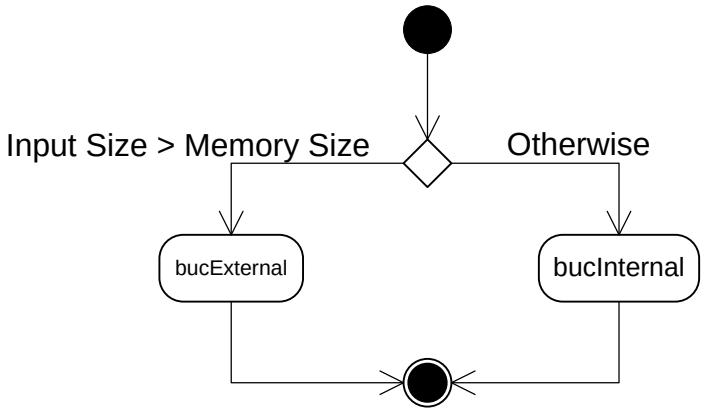

Figure 16: Activity diagram of BUC1 in a global view

The logic of bucInternal can be implemented as a recursive method. During initialization, bucInternal sets every element of the array of OutputRec to be ALL, which represents as -1. If an element contains a value other than -1, this means that bucInternal is working on a view that projects this dimension. Given a set of tuples in Memory, we pass the positions of the boundaries of the set (ie, LEFT and RIGHT) and a column number to bucInternal (ie, D). For example, if we can load the entire input data N into Memory, we will invoke bucInternal with parameter LEFT = 0, RIGHT = N-1 and D = 0. It aggregates the measure of each tuple and saves the result in OutputRec. If the set contains only one tuple, bucInternal will write the tuple to the files of the ancestors of the current view. Otherwise, bucInternal will send OutputRec to RecordWriter to write the aggregated tuple in OutputRec to disk. For each dimension d from D to the total number of dimensions, bucInternal sorts the tuples between LEFT and RIGHT by dimension d. We found that tuples with the same value on dimension d don't need to be sorted and this makes CountingSort shine in BUC. For each group of

tuples that share the same value on dimension d, bucInternal sets the value to the d-th element of the array of OutputRec, and then runs itself on this group with D = d + 1. After going through every group, bucInternal resets the d-th element of the array of OutputRec to ALL. Figure 17 and 18 shows the algorithm of bucInternal.

```
Procedure bucInternal(LEFT, RIGHT, DIMENSION)
Inputs:
     LEFT: the position of the most-left tuple in Memory
     RIGHT: the position of the most-right tuple in Memory
     DIMENSION: the current dimension that we are working
on
Global:
     OUTPUTREC: the aggregated tuple
     NUM_OF_DIMENSIONS: the total number of dimensions
Begin
     Aggregate(LEFT, RIGHT)
     If RIGHT = LEFT
          Write OUTPUTREC to ancestors
          Exit
     End If
     Write OUTPUTREC
     For each d from DIMENSION to NUM_OF_DIMENSIONS
          C = Cardinality(d)
          dataCount[] = Partition(LEFT, RIGHT, d)
          k = LEFT
          For each i from 0 to C - 1
               C = dataCount[i]
               If c >= 1
                    OUTPUTREC[d] = k
                    bucInternal(k, k+c-1,d+1)
                    k = k + c
               End If
          End For
          OUTPUTREC[d] = ALL
     End For
End
```

Figure 17. Algorithm of bucInternal

```
Procedure Aggregate(LEFT, RIGHT)
Inputs:
     LEFT: the position of the most-left tuple in Memory
     RIGHT: the position of the most-right tuple in Memory
Global:
     OUTPUTREC: the aggregated tuple
     MEMORY: the memory that stores the tuples
Begin
     Accumulate the measure of the tuples between LEFT and
RIGHT
     Save the sum in OUTPUTREC
End

Procedure Partition(LEFT, RIGHT, d, C)
Inputs:
     LEFT: the position of the most-left tuple in Memory
     RIGHT: the position of the most-right tuple in Memory
     d: the current dimension
     C: the cardinality of the current dimension
Output:
     dataCount: an integer array in length C.  It tells the
                the number of tuples with the same value on
                dimension d
Global:
     MEMORY: the memory that stores the tuples
Begin
     Perform CountingSort on Memory from LEFT to RIGHT on
d.
     For each i from LEFT to RIGHT
          dataCount[Memory[i][d]] += 1
     End For
End
```

Figure 18. Procedures that are used in bucInternal

After we develop bucInternal, the design of bucExternal is simple. The execution logic of bucExternal is described below. During initialization, all dimension values of OutputRec are set to be ALL. Suppose D is the total number of dimensions of the input data. For each dimension d from 0 to D – 1, bucExternal initializes Memory with (D – d + 1) columns. The first (D – d)

columns are used to keep the dimension values from d to D -1. The last column is used to keep the measure. bucExternal calls DataLoader to load the tuples from the input to Memory. During the loading process, DataLoader discards any dimension values before d. When Memory is full, bucExternal reads the tuples from Memory and sends them to DataPartitioner for partitioning. This step is repeated until all input data has been partitioned. If the aggregated tuple for the ALL view hasn't been output yet, bucExternal writes it to disk. For each partition p that we've created, bucExternal assigns the d-th dimension value of OutputRec to be the d-th dimension value of the tuples in p. Memory is resized according to p and DataLoader loads the partition to Memory. bucExternal runs bucInternal on this partition. Before bucExternal moves forward to the next dimension, it reset the d-th dimension value of OutputRec to ALL. Figure 19 shows the activity diagram of bucExternal.

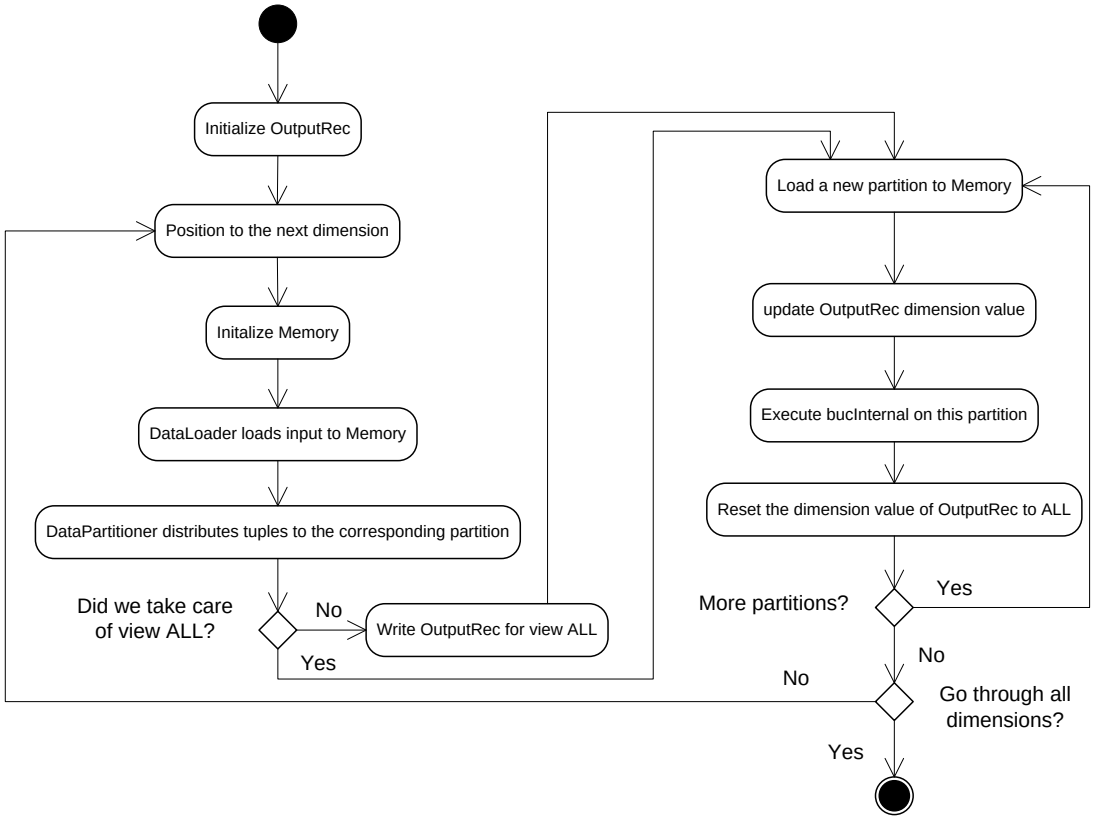

Figure 19. Activity diagram of bucExternal

4.3.1. OutputRec

OutputRec is a data class that holds the current aggregated tuple. It is composited by an integer value for storing the aggregated measure and an integer array for storing the dimension values. The length of the integer array equals the total number of dimensions of the input data. We can determine which view the current aggregated tuple belongs to by reading the integer array. If a dimension value in the integer array does not equal to -1 (e.g. ALL), the corresponding view is projecting the corresponding column. For example, Suppose the input data has four dimensions (A, B, C, and D) and the integer

array is { -1, 4, -1, 3 }. The view for this aggregated tuple is view BC. An integer array in { -1, -1, -1, -1 } is for view ALL.

4.3.2. DataLoader

This class is responsible for loading the input data into Memory. Since we don't need to load all dimension values into Memory in bucExternal, it is designed to be able to discard the first numbers of dimension values of each tuple before writing the partial tuple to Memory in order to reduce the number of I/Os.

4.3.3. RecordWriter

Given OutputRec, RecordWriter writes the aggregated tuple to the file of the corresponding view. For better performance, RecordWriter keeps an OutputStream of every opened file in memory and does not close it until the life cycle of RecordWriter reaches the end. It also provides a recursive method, writeAncestors, to write the aggregated tuple to the ancestors of the corresponding view for optimization. Figure 20 shows the algorithm of writeAncestors.

```
Procedure writeAncestors(OUTPUTREC, MEMORY, ROW)
Input:
     OUTPUTREC: the aggregated tuple
     MEMORY: the memory table
     ROW: the row index of the aggregated tuple in MEMORY
Begin
     Write OUTPUTREC
     dimensions[] = OUPUTREC.dimenions
     For each j from dimensions.length – 1 to 0
          If dimensions[j] = ALL Then
               Exit
          End If
          Copy = OUTPUTREC
          Copy[j[ = MEMORY[row][j]
          writeAncestors(Copy, MEMORY, ROW)
     End For
End
```

Figure 20. Algorithm of writeAncestors

4.3.4. DataPartitioner

DataPartitioner is a class that partitions the input data into multiple groups by columns. Partitioning by column means that tuples, which have the same dimensional value on the specified column, will go to the same partition. When DataPartitioner receives a tuple, it checks the dimension value of the specified column and sends it to the corresponding partition. Tuples of each partition will be written in a temporary file, which will be loaded by DataLoader later on.

4.4. BUC2

BUC2 is the first version of our implementation on multi-dimensional partitioning in BUC. In the previous design of BUC1, if we cannot keep the entire input data in Memory, we will partition the data by dimension and save each

partition into a file. We then load each partition back to Memory and execute bucInternal on the partition. The limitation of this algorithm is that it cannot handle a partition, which size is larger than Memory. BUC algorithm will not be able to generate a data cube correctly if only a portion of a partition is loaded in Memory. The approach of solving this problem is described in section 3.

Our design of this algorithm will generate the optimal set of processing paths when single-dimensional partitioned data requires two-dimensional partitioning. We believe that generating the optimal set of processing paths for every single-dimensional partition we encounter will allow our algorithm to reach the best performance because the data distribution of each single-dimensional partition is different. A new set of processing path is required for a single-dimensional partition in order to reduce the number of I/Os.

We have made an optimization in BUC2 regarding the way to write tuples in a partition file. In BUC1, tuples with the same dimension value on the partitioning column go to the same partition file. Writing the entire tuple to the file, we waste space on saving the same dimension value and also waste time on writing and reading the same dimension value. In BUC2, we skip writing the common dimension value into a partition file but write the value on the file name. Our evaluation shows that this optimization improves the performance of BUC2.

We restructure our design from BUC1 and the basic components of BUC2 are: Main, Memory, InMemoryBUC, PartitionBUC, ExtendedPartitionBUC, and PipeSort. Main is the main class that contains the setting of BUC2 and it is the place where the execution begins. Memory is same component from BUC1. InMemoryBUC is a class that computes a data cube from tuples in memory. It is a modification of the bucInternal procedure in BUC1. PartitionBUC is a class that computes a data cube with single-dimensional partitioning. This class is a modification of the bucExternal procedure in BUC2. If PartitionBUC needs to the second partitioning on a partitioned data, PartitionBUC will pass the execution to ExtendedPartitionBUC, which is responsible for performing two-dimensional partitioning. For simplicity, our implementation handles up to two-dimensional partitioning as we believe that the computation logic beyond two-dimensional partitioning is similar. PipeSort is used by ExtendedPartitionBUC to compute the missing views. Figure 21 shows the general structure of BUC2.

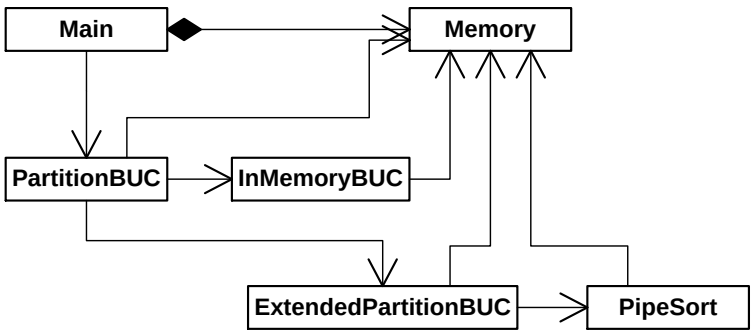

Figure 21. General class diagram of BUC2

The global execution logic of BUC2 is described as follows. After the initialization in Main, PartitionBUC will load the input data into Memory. If the entire input data can fit in Memory, PartitionBUC will call InMemoryBUC to compute a data cube. Otherwise, PartitionBUC will partition the data into numbers of partitions. It then loads each partition into Memory and tries to compute a data cube based on the partition by calling InMemoryBUC. If Memory cannot contain the entire partition, PartitionBUC will pass the execution to ExtendedPartitionBUC for two-dimensional partitioning. Figure 22 shows the activity diagram of BUC2.

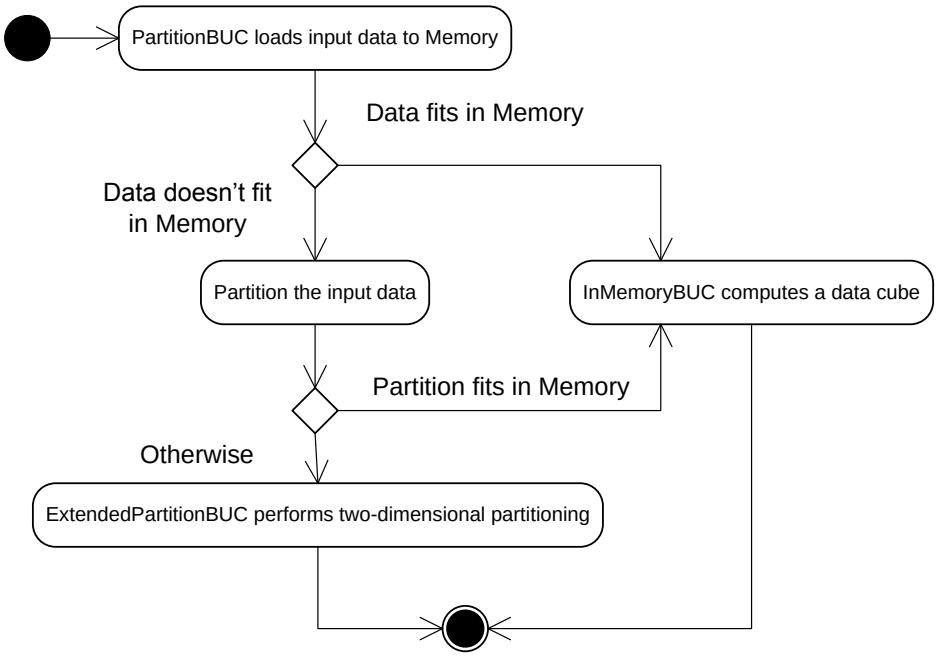

Figure 22. Activity diagram of BUC2

4.4.1. InMemoryBUC

InMemoryBUC is a class that implements the in-memory version of BUC. Unlike BUC1 that has only the bucInternal procedure to compute a data cube, InMemoryBUC provides two methods: InMemoryBUC.run and InMemoryBUC.runWithoutAll. InMemoryBUC.run generates the entire data cube; on the other hand, InMemoryBUC.runWithoutAll generates all views of a data cube except the view All. Figure 23 and 24 shows the algorithms of InMemoryBUC.

```
Procedure InMemoryBUC.run(LEFT, RIGHT, DIMENSION)
Inputs:
     LEFT: the position of the most-left tuple in Memory
     RIGHT: the position of the most-right tuple in Memory
     DIMENSION: the current dimension that we are working
on
Global:
     OUTPUTREC: the aggregated tuple
     NUM_OF_DIMENSIONS: the total number of dimensions
Begin
     Aggregate(LEFT, RIGHT)
     If RIGHT = LEFT
          Write OUTPUTREC to ancestors
          Exit
     End If
     Write OUTPUTREC
     InMemory.runWithoutAll(LEFT, RIGHT, DIMENSION)
End
```

Figure 23. Algorithm of InMemoryBUC

```
Procedure InMemoryBUC.runWithoutAll(LEFT, RIGHT, DIMENSION)
Inputs:
     LEFT: the position of the most-left tuple in Memory
     RIGHT: the position of the most-right tuple in Memory
     DIMENSION: the current dimension that we are working
on
Global:
     OUTPUTREC: the aggregated tuple
     NUM_OF_DIMENSIONS: the total number of dimensions
Begin
     For each d from DIMENSION to NUM_OF_DIMENSIONS
          C = Cardinality(d)
          dataCount[] = Partition(LEFT, RIGHT, d)
          k = LEFT
          For each i from 0 to C - 1
               C = dataCount[i]
               If c >= 1
                    OUTPUTREC[d] = k
                    InMemoryBUC.run(k, k+c-1,d+1)
                    k = k + c
               End If
          End For
          OUTPUTREC[d] = ALL
     End For
End

Note: The implementation of Procedure Aggregate and
Partition is the same as we described in BUC1.
```

Figure 24. Procedure that is used in InMemoryBUC

4.4.2. PartitionBUC

PartitionBUC is a class that governs the entire execution logic of BUC2. This class consists of the following basic components: DataLoader, DataPartitioner, InMemoryBUC, and ExtendedPartitionBUC. DataLoader is responsible for loading the input data or partitioned data into Memory. The implementation of DataLoader is the same as we described in BUC1.

DataPartitioner is responsible for performing partition by dimension. The implementation of this class is a little different to the implementation in BUC1 because of the optimization we implement. InMemoryBUC is the in-memory version of BUC and we have described it already in the previous section. ExtendedPartitionBUC is a class that contains our implementation on multi-dimensional partitioning. Figure 25 shows the class diagram of PartitionBUC.

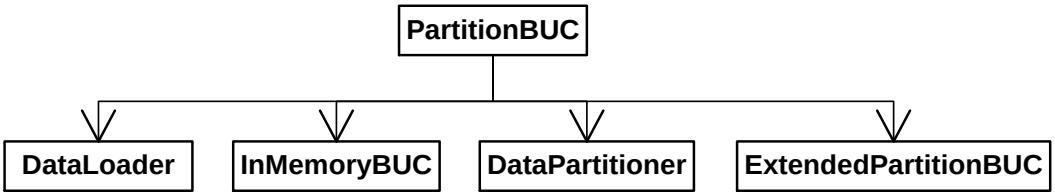

Figure 25. Class diagram of PartitionBUC

The execution logic of PartitionBUC is described as below. Let D be the total number of dimensions of input. For each dimension d, PartitionBUC initializes Memory to have (D – d + 1) columns and load the input into Memory. Any dimension value, which order is less than d, will be skipped during the loading process. If DataLoader can load all data into Memory, PartitionBUC will run InMemoryBUC.run to generate a data cube. If DataLoader can load all data into Memory and view All has already been created, PartitionBUC will run InMemoryBUC.runWithoutAll instead. PartitionBUC will move onto the next dimension if PartitionBUC can load the entire input into Memory. Otherwise, PartitionBUC will ask DataPartitioner to partition the data until the entire input is partitioned. After that, PartitionBUC will generate view All if the view hasn't been generated yet. PartitionBUC then loads every partition into Memory. For every

partition it loads, PartitionBUC will update outputRec[d] with the common dimension value of the partition. If a partition can be fit into Memory, PartitionBUC will invoke InMemoryBUC.run(); otherwise, it will invoke ExtendedPartitionBUC to handle additional partitioning. Before PartitionBUC moves onto the next dimension, it will reset outputRec[d] back to ALL. Figure 26 shows the activity diagram of PartitionBUC.

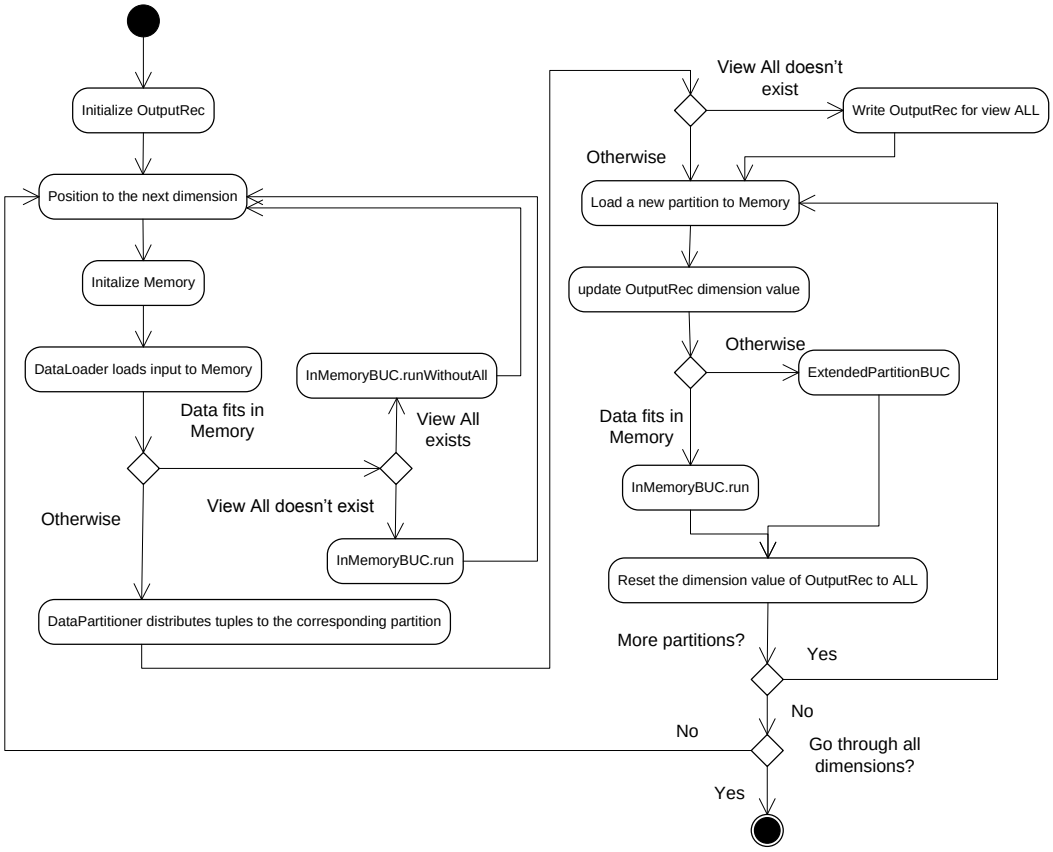

Figure 26. Activity diagram of partitionBUC

4.4.3. DataPartitioner

DataPartitioner helps partitioning data and write tuples to the corresponding partition files. Its implementation is similar to the implementation in BUC1 except that we will skip writing the dimension value of the partitioning column to the partition file. The dimension value will be indicated in the file name.

4.4.4. ExtendedPartitionBUC

ExtendedPartitionBUC contains our implementation of two-dimensional partitioning logic. It consists of the following basic components: DataLoader, DataPartitioner, InMemoryBUC, ProcessingPath, ProcessingTreeManager, PlanGenerator and PipeSort. ProcessingPath is a data structure that represents a processing path. The duty of ProcessingTreeManager is to create a processing tree for the current two-dimensional partitioning. ExtendedPartitionBUC then summits the tree to PlanGenerator to generate the optimal set of processing paths for the partition. PipeSort generates the missing tuples, according to the processing paths. Figure 27 shows the class diagram of ExtendedPartitonBUC.

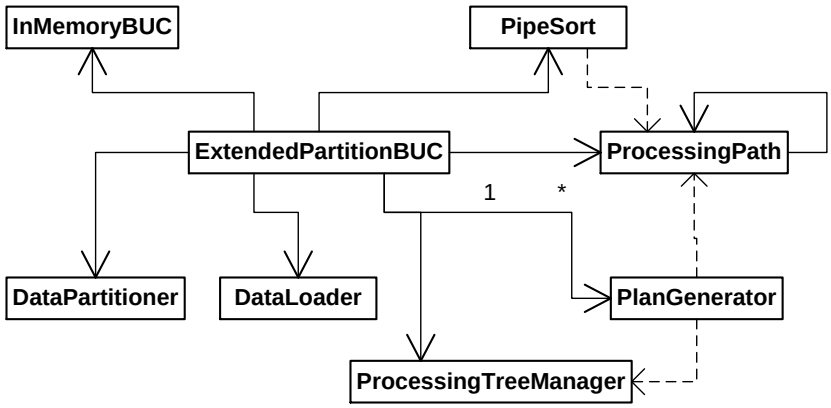

Figure 27. Class diagram of ExtendedPartitionBUC

The execution logic of ExtendedPartitionBUC is described as below. ProcessingTreeManager constructs a processing tree for the current single-dimensional partition. On each node of the processing tree, ExtendedPartitionBUC saves the current file position of the corresponding view for marking the left boundary of the upcoming aggregated tuples. DataLoader then loads the partition into Memory and DataPartitioner performs two-dimensional partitioning. ExtendedPartitionBUC executes InMemoryBUC.run to compute a data cube for each of the partitions. After it goes through all the partitions, it updates the current file position of every view on the processing tree for marking the right boundary of the aggregated tuples. If the left boundary points to the same position as the right boundary, the corresponding view has no aggregated tuple and we call it as a missing view. ExtendedPartitionBUC then sends the updated processing tree to PlanGenerator, which generates the optimal set of processing paths. PipeSort generates the missing tuples,

according to the processing paths. Figure 28 shows the activity diagram of ExtendedPartitionBUC.

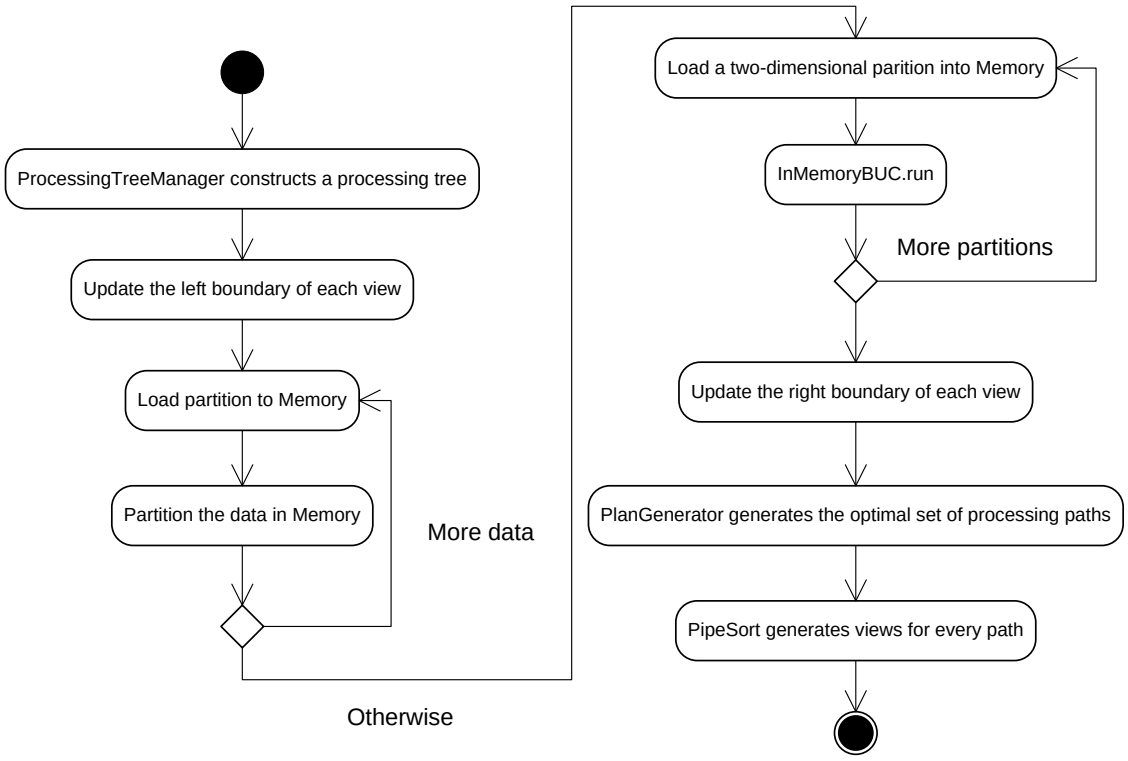

Figure 28. Activity diagram of ExtendedPartitionBUC

4.4.5. Processing Tree

A processing tree is a data structure that is generated by ProcessingTreeManager. It is composited by numbers of TreeNode, which is a simple class that represents a view. A TreeNode contains a view ID, a reference to its parent, a Boolean flag indicating whether sorting is required to generate this view from its parent, a list of reference to its children, a file position for the left boundary of the aggregated tuples from the current partition, and a file position of

the right boundary of the same tuples. A view ID is a string consists of 0 and 1. If the i-th character of a view ID is 1, the corresponding view is projecting the i-th dimension. For example, given input with four dimensions (A, B, C, and D), the ID for view ACD will be 1011.

4.4.6. ProcessingPath

ProcessingPath is a data class that represents a processing path in BUC2. It consists of a list of view IDs, each of which represents the view that this path travels to. The first view ID represents the source view that we will read from in order to generate the rest of the views on the list.

4.4.7. ProcessingTreeManager

ProcessingTreeManager is a class that constructs a processing tree for PlanGenerator to generate the optimal set of processing paths, which PipeSort will use to generate the missing tuples. The algorithm to construct a processing tree is a recursive function called buildTree, which takes TreeNode as the only parameter. The algorithm of buildTree is shown in Figure 29.

```
Procedure buildTree(TreeNode)
Input:
     TreeNode: a tree node
Global:
     d: the partitioning column
     D: the total number of dimension
Begin
     For each dimension i from d+1 to D
          If TreeNode.viewId[i] = 0
               Continue
          End If
          CopyID = TreeNode.viewID
          CopyID[i] = 0
          ChildNode.viewID = CopyID
          If CopyID shares the same prefix as TreeNode
               ChildNode.sorted = true
          End If
          Add ChildNode under TreeNode
          ChildNode.parent = TreeNode
          buildTree(ChildNode)
     End For
End
```

Figure 29. Algorithm of buildTree

4.4.8. PlanGenerator

Given a processing tree, PlanGenerator constructs the optimal set of processing paths by going through TreeNodes at the same level from bottom to top. For each level, PlanGenerator gathers every distinct TreeNode that is qualified as a missing view into a set. It then creates a cost table. Each row of the cost table represents a parent of one of the TreeNodes. Each column of the cost table represents one of the TreeNodes. The value of row x and column y on a cost table means the cost to generate the y-th view from the x-th view. PlanGenerator then submits the cost table to the Hungarian algorithm to find the

minimum cost matching between the parent views and the child views. This process is repeated until PlanGenerator goes through every level and reaches the top level. Finally, PlanGenerator will construct the optimal set of processing paths based on the matching that the Hungarian algorithm returns. Figure 30 shows the activity diagram of PlanGenerator.

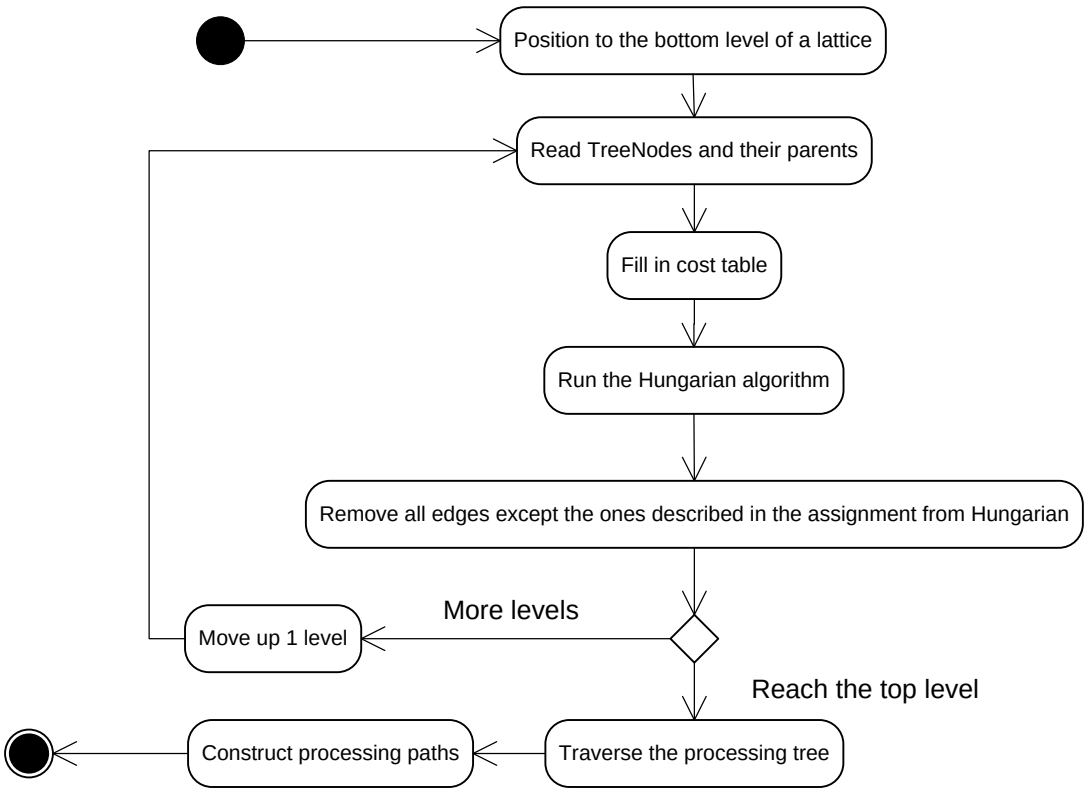

Figure 30. Activity diagram of PlanGenerator

4.4.9. PipeSort

This is the simplified version of PipeSort comparing to the algorithm we described in section 4.2 because the construction of processing paths is removed. The execution logic of this version of PipeSort is described as below.

Given a processing path, PipeSort sets the first view to be the source view and creates an instance of PipeSortHelper for each of the rest of the views. PipeSortHelper is the same class as ViewAggregator in section 4.2.4 except that the name is changed for clarity. PipeSort performs external sorting on the tuples within the boundaries of the source view by MergeSort and reads the sorted tuples in order. For every tuple it reads, PipeSort passes the tuple to every PipeSortHelper to generate the aggregated tuple for the corresponding view.

4.5. BUC3

BUC3 is the advanced version of BUC2. The difference between the two algorithms is that BUC3 uses the same set of processing paths for all two-dimensional partitions that are partitioned by the same pair of dimensions. The only change we need to make is in ExtendedPartitionBUC, which now keeps a set of processing paths as its attribute. The activity diagram of the modified version ExtendedPartitionBUC is shown in Figure 31.

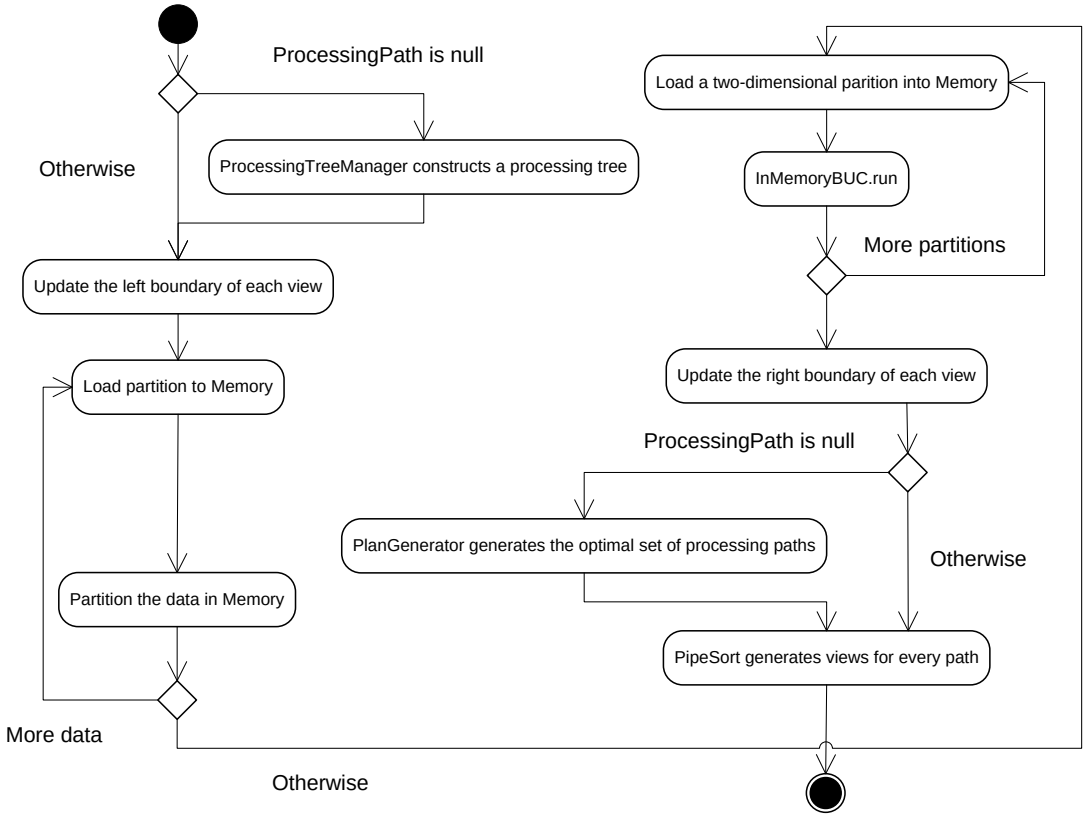

Figure 31. Activity diagram of the modified ExtendedPartitionBUC

5. Evaluation

We implemented PipeSort, BUC1, BUC2, and BUC3 in JAVA and compared them in terms of processing time and the number of read and write (R/W). BUC1 was our first implementation on Bottom-Up Computation that only supported single-dimensional partitioning without optimization on memory usage and I/O reduction. BUC2 was our second implementation on Bottom-Up Computation that supported up to two-dimensional partitioning with optimization on memory usage and I/O reduction. However, this algorithm generated processing paths for every partition it encountered. BUC3 was similar to BUC2

except that BUC3 generated processing paths once for a two-dimensional partitioned partition (e.g. AB_1) under a single-dimensional partitioned partition (e.g. A) and used the same set of processing paths for the rest of two-dimensional partitioned partitions (e.g. AB_i) that were under the same single-dimensional partitioned partition (e.g. A). In the rest of this paper, if the behavior or the outcome is very similar among BUC1, BUC2, and BUC3, we will refer all three algorithms as BUC for simplicity; otherwise, we will refer the algorithms using their original names. We generated 14 sets of sample data, which will be described in section 5.2, and fed them into the algorithms with different memory conditions: 10 MB, 1 MB, 500kB, and 250kB in order to trigger partitioning. We analyzed the results from the perspective of the number of dimensions, sparsity and size of input.

5.1. System Configuration

We executed our algorithms on a computer, which was equipped with Intel® Core™ 2 Duo CPU at 2.50 GHz and 3 GB of memory. Its operating system was Windows Vista™ Home Edition (32-bit) with Service Pack 1. The algorithms were run in Java™ SE Runtime Environment in version 1.6.0. During the execution, the computer was running in the minimum number of applications in order to reduce the noise of the results as much as possible.

5.2. Sample Generation

Since we are interested in the performance of our algorithms when the properties of input data are changed, we have introduced three variables to our sample generation: the number of dimensions (D), sparsity (S), and the number of tuples (T). We define sparsity (S) as the number of tuples (T) of input data divided by the multiplication of the size of the domain of all dimensions of the data. Let A be the size of the domain of a dimension of a set of sample data. In addition, suppose that A is the same for all dimensions in the sample data. We then have the following equation:

$$A^D * S = T$$

For a set of sample data with predefined D, S, and T, we obtain A through the above equation. We then multiply A with a random number from 0 to 1 for each value in the sample data. As the results, a sample data file can be treated as a T by D table filled with numbers.

5.3. Measurement Methods

We evaluated the processing time and the number of read and write of the algorithms by placing timestamps and counters in them. To measure the processing time of each algorithm, we constructed a JAVA object (java.util.Date) at the beginning and at the end of the program separately. When the execution was done, we subtracted the objects and found out the duration of the algorithm in microseconds.

The number of read and write (R/W) refers to the number of times for an algorithm to read/write a value from/to a storage device. To measure R/W, we constructed a global counter in each algorithm, which incremented the counter when a value was transferred between memory and disk. Such transfer was occurred in loading input data or generated views from files, external sorting, and writing views to files.

5.4. Number of Dimensions

We selected five sets of sample data to be our candidates for testing our algorithms. Since we were interested in the relationship between the performance of our algorithms and the number of dimensions only, we chose the sets of data to have the same sparsity at 0.5 and the same data size at 400,000 numbers of values. The benchmark can be found in Appendix.

For the tests under 10 MB of memory, we found that BUC was faster than PipeSort in processing in every test case. The purpose of running the algorithms in such memory condition was to allow all four algorithms to perform in-memory cubing without any external sorting. As the number of dimensions increased, the processing time of all algorithms increased exponentially. However, the rate of the exponential increase in PipeSort was larger than that in BUC. In addition, BUC generally ran two times faster than PipeSort and the number of R/W of PipeSort was nearly double to that of BUC in every test case. Figure 32 shows

the graph of the processing time versus the number of dimensions for this testing.

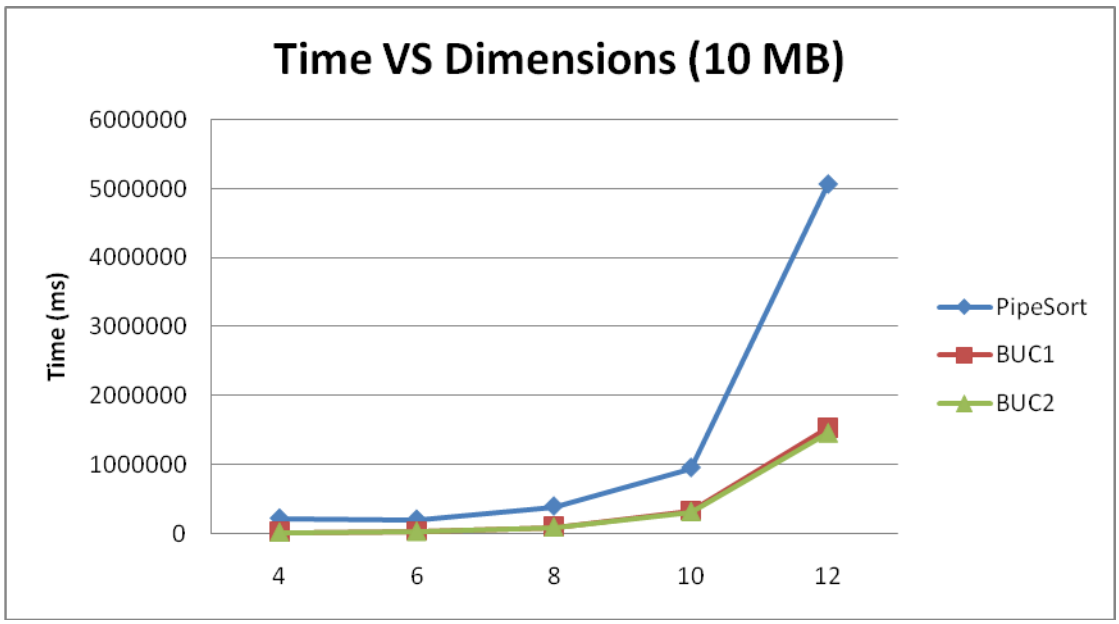

Figure 32. Processing time versus number of dimensions (10 MB)

We did not collect the benchmark from BUC3 since the cubing logic with sufficient memory was the same as the logic of BUC2. Hence, the benchmark from BUC3 would be the same as the benchmark from BUC2.

For the tests under 1 MB of memory, we found that BUC was faster than PipeSort in processing in every test case. The purpose of running the algorithms under such memory condition was to trigger external sorting in PipeSort and single-partitioning in BUC. Similar to the benchmark in 10 MB of memory, the processing time of all algorithms were exponentially proportional to the number of dimensions and the processing time of PipeSort was still nearly double to the

processing time of BUC. We also found that the processing time of BUC2 was less than that of BUC1. Comparing to BUC1, The number of R/W of BUC2 was reduced at least 5% and the number of single dimensional partitioning in BUC2 was reduced by at least 25%. Figure 33 shows the graph of the processing time versus the number of dimensions for this testing.

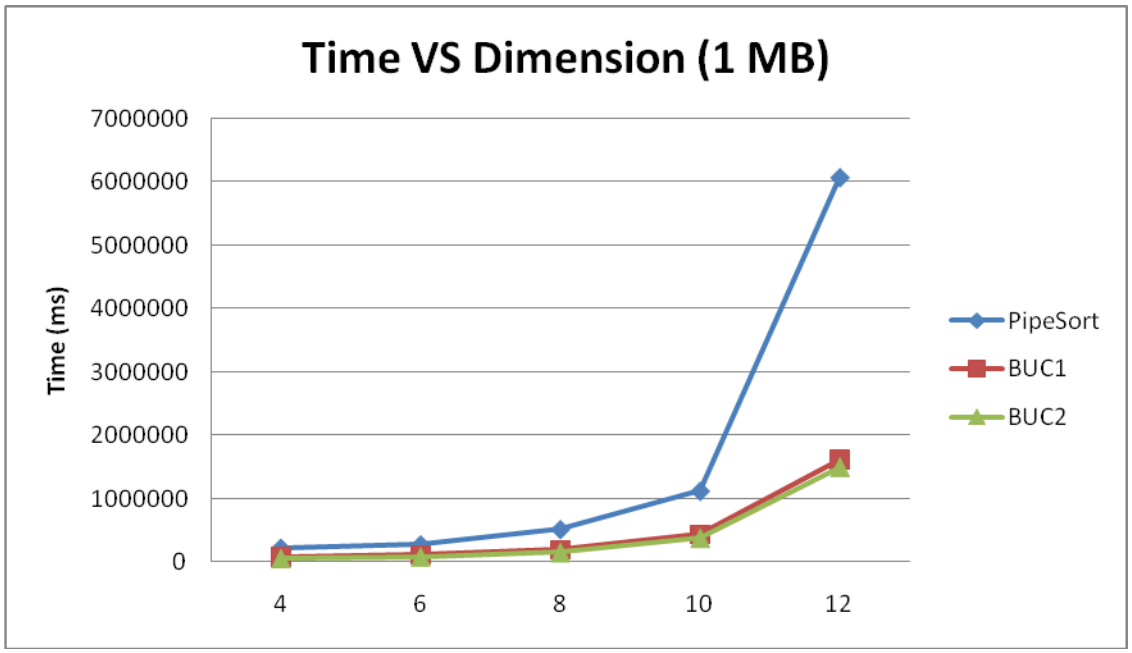

Figure 33. Processing time versus number of dimensions (1 MB)

We didn't evaluate BUC3 under 1MB of memory because the cubing logic was the same as the logic of BUC2 and two-dimensional partitioning didn't happen in this memory condition. The benchmark from BUC3 would be the same as that from BUC2.

For the tests under 500 kB of memory, the performance of BUC was better than that of PipeSort although we didn't collect enough data. First of all,

the size of the memory wasn't small enough for BUC3 to trigger two-dimensional partitioning. Second of all, BUC3 failed on processing data in 10 dimensions and 12 dimensions. It required higher dimensional partitioning to handle these data. Regardless to these situations, BUC2 still performed at least 50% better than PipeSort. Figure 34 shows the graph of the processing time versus the number of dimensions for this testing.

We didn't evaluate BUC1 because we understood that BUC2 performed better than BUC1 in the previous testing. We also didn't include the benchmark from BUC3 since no two-dimensional partitioning occurred in processing data in lower number of dimensions.

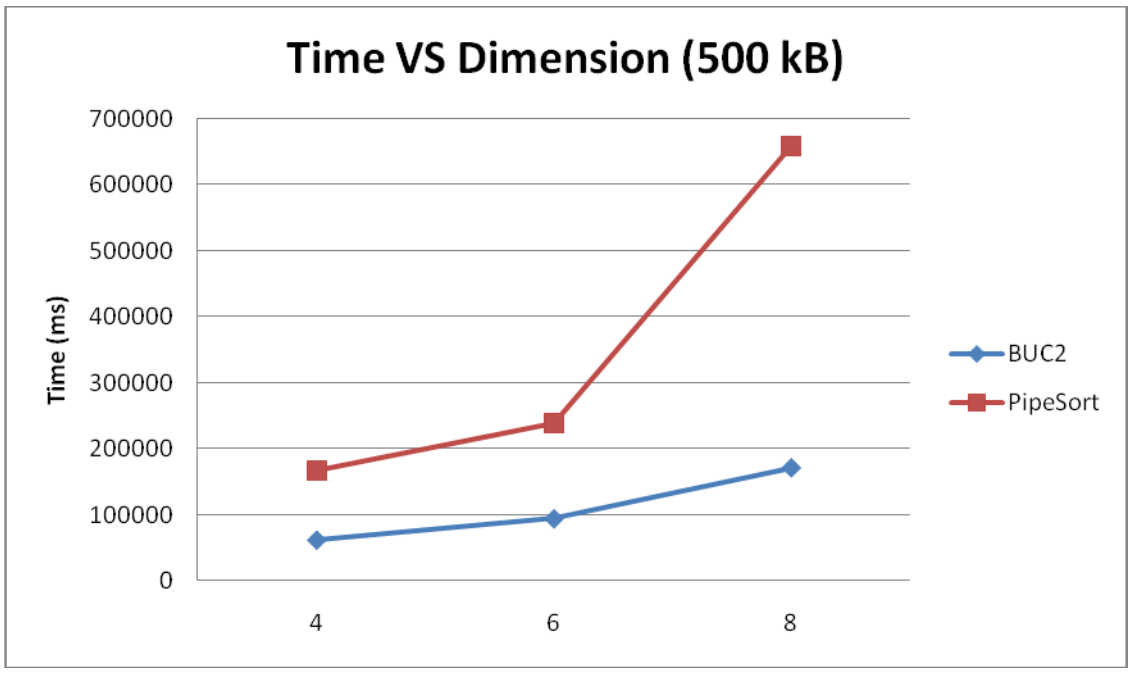

Figure 34: Processing time versus number of dimensions (500 kB)

For the tests under 250 kB of memory, we collected mixed results. Similar to the previous testing, we couldn't collect the benchmark for input data in 10 Dimensions and 12 Dimensions. Fortunately, two-dimensional partitioning happened in processing data in 8 dimensions. In processing data in 4 dimensions and 6 dimensions, only single-dimensional partitioning occurred in BUC2; thus, the processing time of BUC was faster than the processing time of PipeSort. However, the processing time of BUC2 was larger than the processing time of PipeSort in processing data in 8 dimensions. Figure 35 shows the graph of the processing time versus the number of dimensions for this testing.

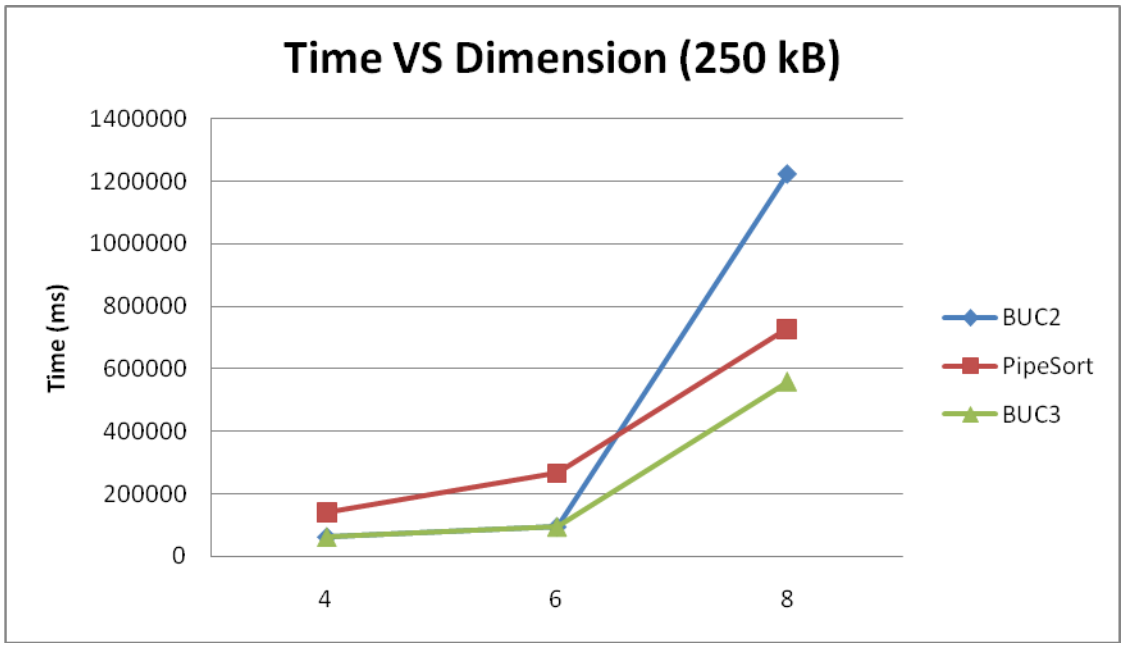

Figure 35. Processing time versus number of dimensions (250 KB)

We didn't evaluate BUC1 and the reasons were the same as we mentioned in the tests under 500 kB of memory.

5.5. Sparsity

We selected five sets of sample data to be our candidates for testing our algorithms. Since we were interested in the relationship between the performance of our algorithms and the sparsity only, we chose the sets of data that were in the same number of dimensions (8) and the same number of tuples (6000). The benchmark can be found in Appendix.

For the tests under 10 MB, BUC processed the sample data at least 4 times faster than PipeSort in every test case. Under such memory condition, all algorithms could process the data in memory without any partitioning or external sorting. When sample data tended to be dense, all algorithms took less time to process the data entirely. Particularly, the processing time of PipeSort was cut approximately 72% when the sparsity of the data increased to 0.9. Figure 36 shows the graph of the processing time versus the sparsity for this testing.

We didn't evaluate BUC3 in this testing because the computation logic of cubing with sufficient memory of BUC3 was the same as that of BUC2. Therefore, the benchmark from BUC3 should be similar to the benchmark of BUC2.

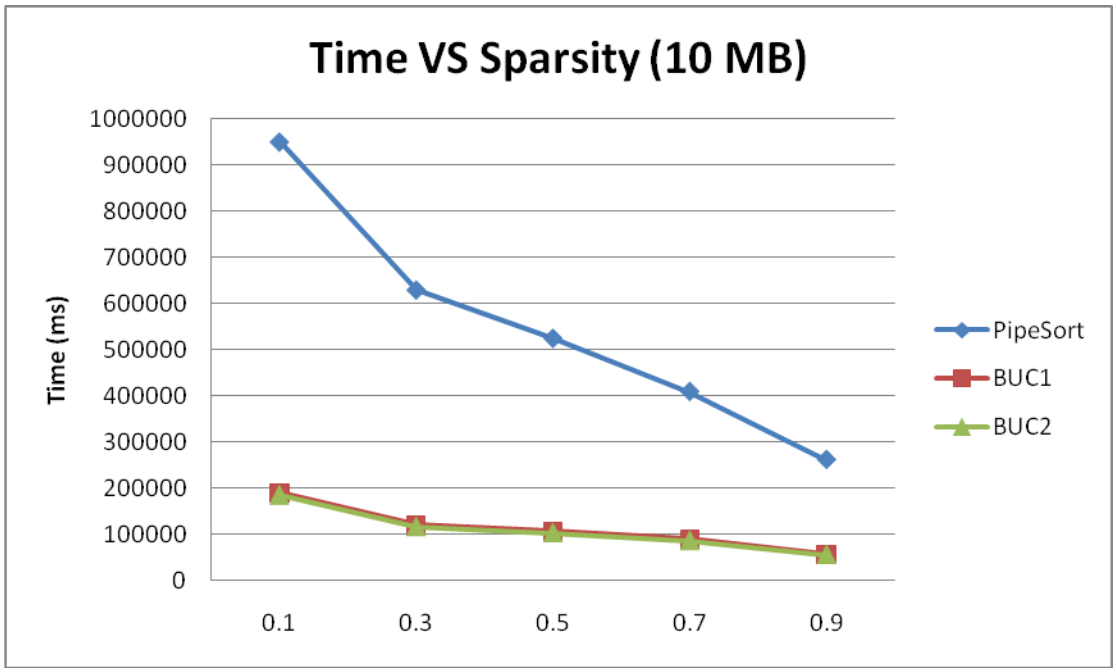

Figure 36. Processing time versus sparsity (10 MB)

For the tests under 1 MB, BUC performed better than PipeSort in terms of processing time. Under such memory condition, PipeSort performed external sorting and BUC performed single-dimensional partitioning. We found that the processing time of all algorithms decreased as the sparsity of the data increased. Similar to the previous testing, the processing time of PipeSort was cut down significantly when the data tended to dense. Figure 37 shows the graph of the processing time versus the sparsity of data for this testing.

We didn't evaluate BUC3 in this testing because two-dimensional partitioning didn't happen under such memory condition. The benchmark from BUC3 would be very similar to that of BUC2.

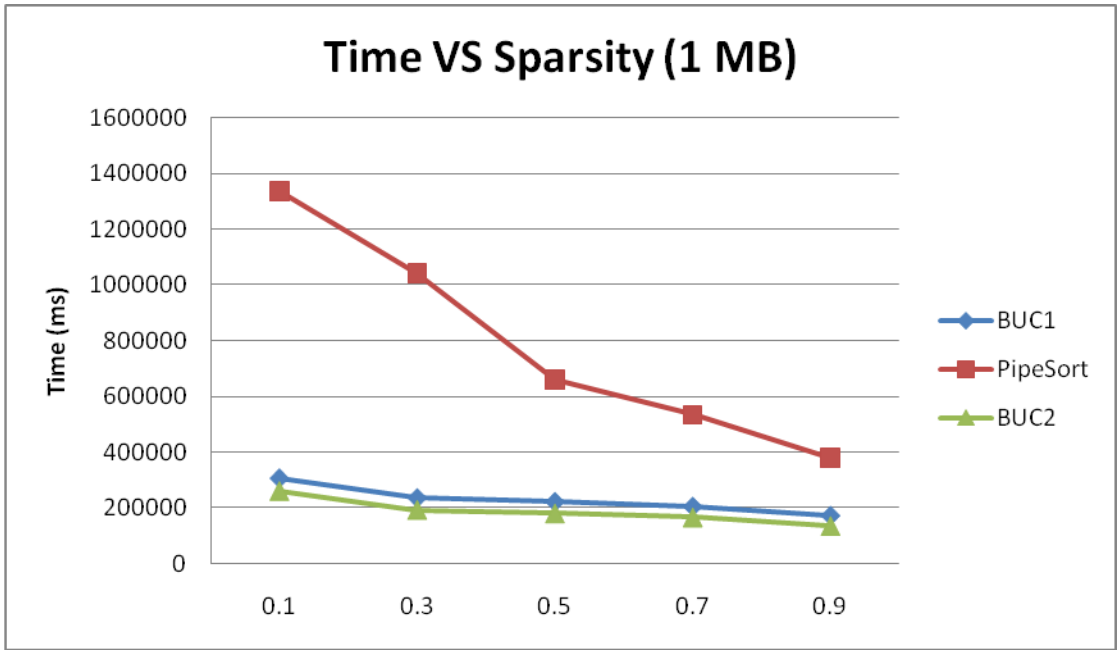

Figure 37. Processing time versus sparsity (1 MB)

For the tests under 500 kB, BUC still performed better than PipeSort in terms of processing time. Under such memory condition, PipeSort performed external sorting and BUC performed single-dimensional partitioning only. All algorithms performed better when the sparsity of the data increased and this change was significant in PipeSort. Figure 38 shows the graph of the processing time versus the sparisty of data for this testing.

We didn't evaluate BUC1 and BUC3 in this testing because we understood that BUC2 performed better than BUC1 in the previous testing and the logic of BUC3 on handling single-dimensional partitioning was the same as that of BUC2. The benchmark from BUC3 would be very similar to the benchmark from BUC2.

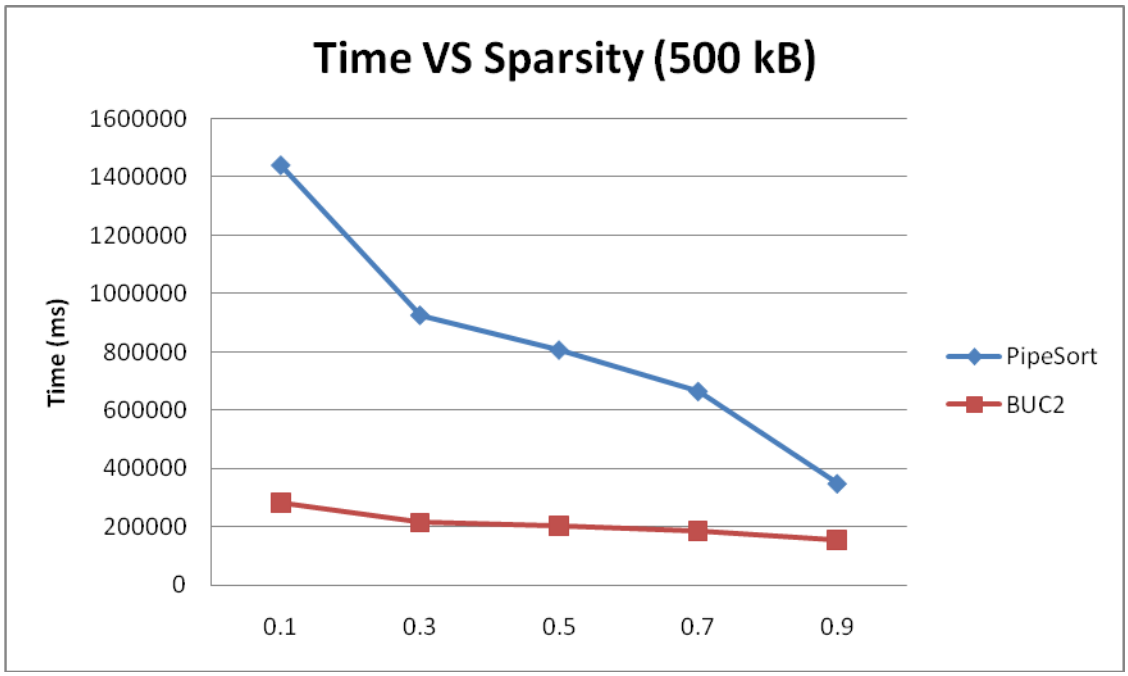

Figure 38. Processing time versus sparsity (500 kB)

For the tests under 250 kB of memory, we collected mixed results. Under such memory condition, external sorting occurred in PipeSort while two-dimensional partitioning happened in BUC. First of all, the processing time of all algorithms was reduced when the sparsity of the sample data increased and this change was significant in PipeSort. Second of all, although BUC3 generally performed better than PipeSort in terms of processing time, BUC2 performed poorer than PipeSort by 20% to 60%. Figure 39 shows the graph of the processing time versus the sparisty of data for this testing.

Again, we didn't evaluate BUC1 and the reasons were the same as we mentioned in the tests under 500 kB of memory.

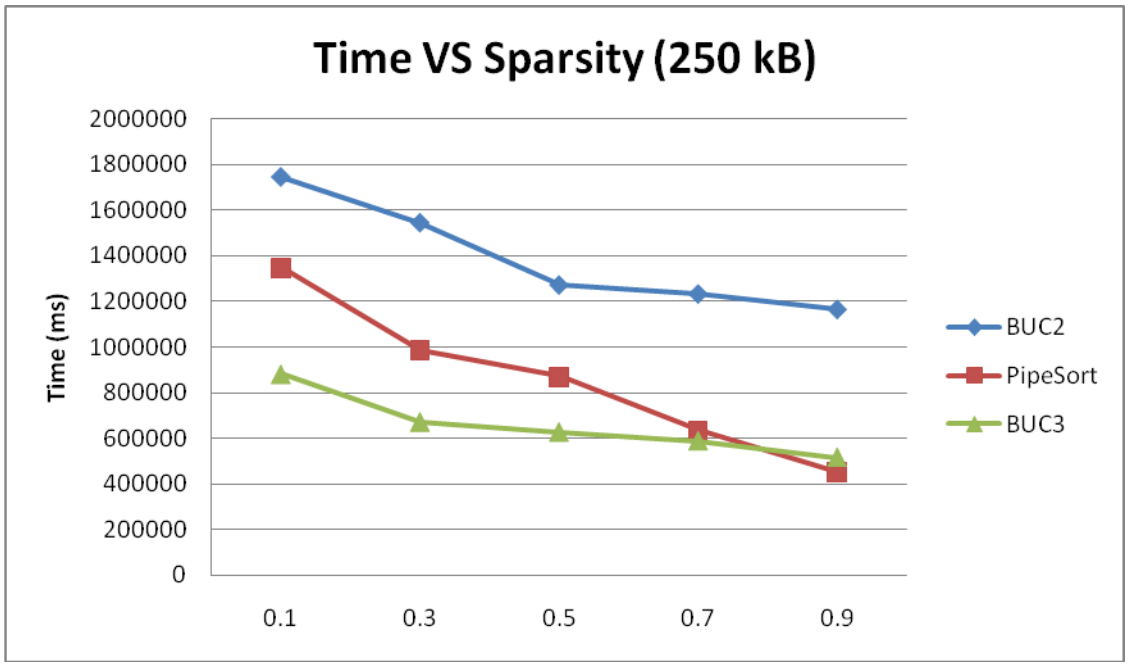

Figure 39. Processing time versus sparsity (250 kB)

5.6. Number of Tuples

We selected five sets of sample data to be our candidates for testing our algorithms. Since we were interested in the relationship between the performance of our algorithms and the size of input only, we chose the sets of data to have the same sparsity at 0.5 and the same number of dimensions (8). The benchmark can be found in Appendix.

For the tests under 10 MB of memory, the performance of BUC was better than the performance of PipeSort. The purpose of running the algorithms under this memory condition was to compare them on computing a data cube in memory without any external sorting or partitioning. In terms of processing time, BUC was at least 3.8 times faster than PipeSort. When processing the data in

120,000 tuples, the processing time of BUC was about 8.8 times faster than PipeSort. Although BUC and PipeSort were scalable in terms of the size of input, the increasing rate on processing time of PipeSort was way higher than that of BUC. Figure 40 shows the graph of processing time versus the number of tuples for this testing.

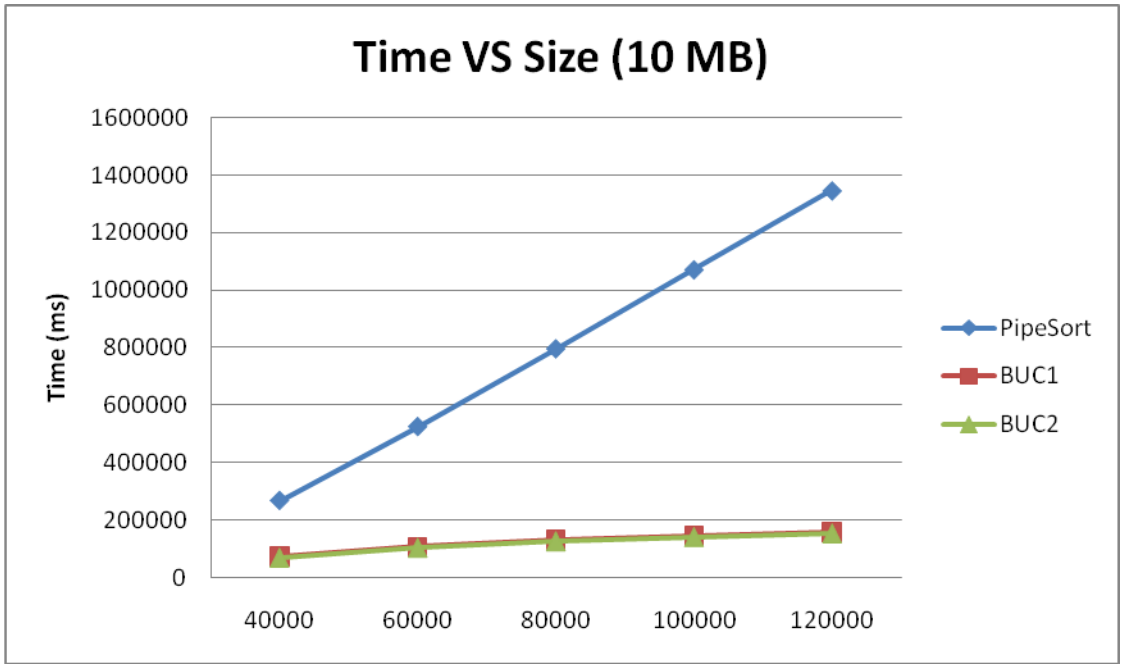

Figure 40. Processing time versus size (10 MB)

We evaluated BUC1 and BUC2, but not BUC3 in this testing. We found that the processing time of BUC2 was between 2.2% and 3.7% less than the processing time of BUC1. Since this testing didn't trigger any partitioning and the logic of BUC3 on computing a data cube in memory was the same as the logic of BUC2, the benchmark of BUC3 would be the same as the benchmark of BUC2. Therefore, we didn't spend time on evaluating BUC3 in this testing.

For the tests under 1 MB of memory, the performance of BUC was better than the performance of PipeSort. The purpose of this testing was to trigger external sorting on PipeSort and single-partitioning on BUC. Under such memory condition, the processing time of BUC was between 3.6 and 4.8 times faster than the processing time of PipeSort. Although both algorithms were scalable in terms of the size of input, the increasing rate on the processing time of PipeSort was higher than that of BUC. Figure 41 shows the graph of the processing time versus the number of tuples for this testing.

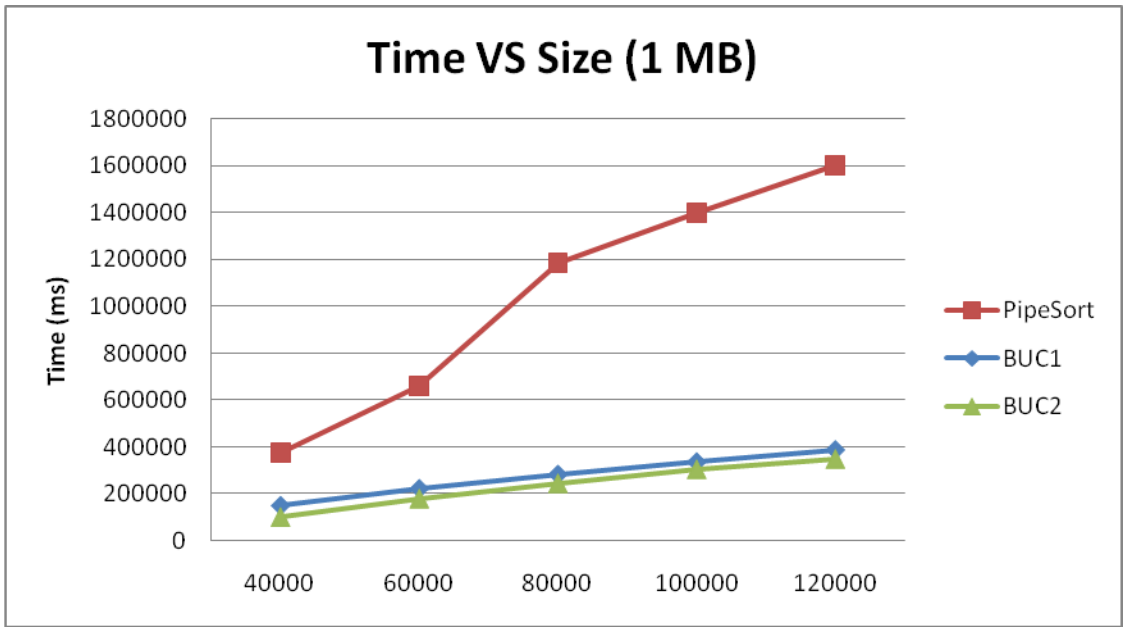

Figure 41. Processing time versus size (1 MB)

We evaluated BUC1 and BUC2, but not BUC3 in this testing. We found that the processing time of BUC2 was between 9.3% and 31.9% less than the processing time of BUC1. Since this testing didn't trigger two-dimensional

partitioning and the logic of BUC3 on computing a data cube with single-dimensional partitioning was the same as the logic of BUC2, the benchmark of BUC3 would be the same as the benchmark of BUC2. Therefore, we didn't spend time on evaluating BUC3 in this testing.

For the tests under 500 kB of memory, we collected mixed results on the performance of the algorithms. The purpose of this testing was to trigger two-dimensional partitioning in BUC and this happened when we processed input data at 80,000 tuples and beyond. For input data at 40,000 tuples and 60,000 tuples, the processing time of BUC was 3.8 times faster than the processing time of PipeSort. However, for input data at 80,000 and beyond, the processing time of BUC3 was only 1.5 times faster than the processing time of PipeSort; the processing time of BUC2 was even 1.13 times slower than the processing time of PipeSort. Figure 42 shows the graph of the processing time versus the number of tuples for this testing.

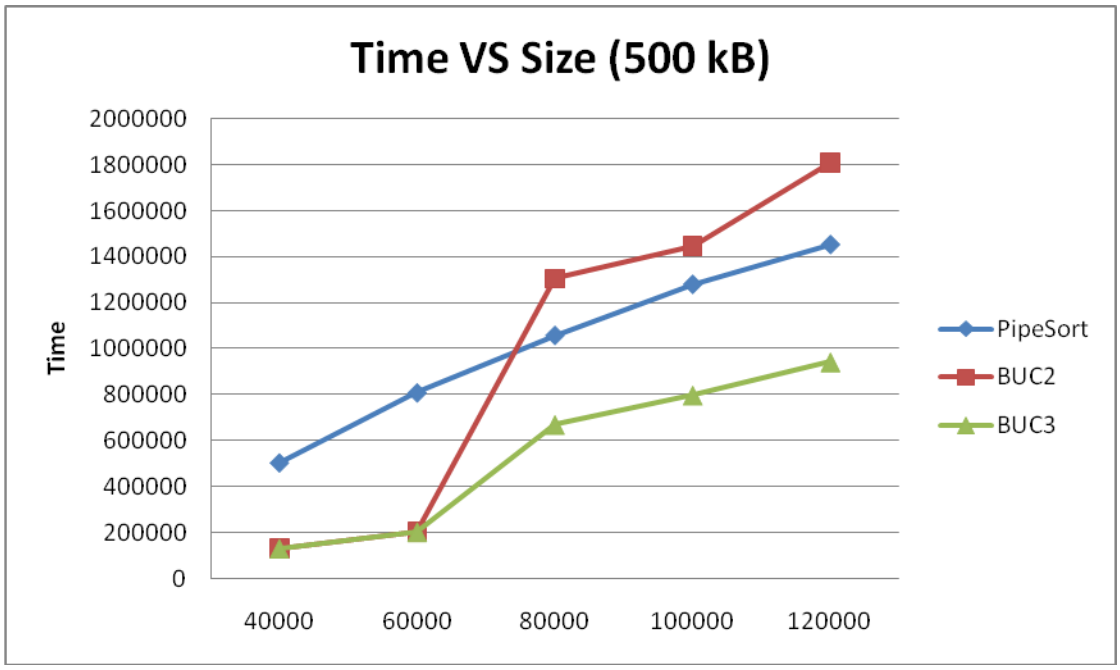

Figure 42. Processing time versus size (500 kB)

For the tests under 250 kB of memory, BUC3 had a better performance than PipeSort but BUC2 didn't do well comparing to PipeSort. Under this memory condition, two-dimensional partitioning happened in processing every set of input data. We found that the processing time of BUC3 was about 1.5 times faster than the processing time of PipeSort; on the other hand, BUC2 was in average of 1.54 times slower than the processing time of PipeSort. Figure 43 shows the graph of the processing time versus the number of tuples for this testing.

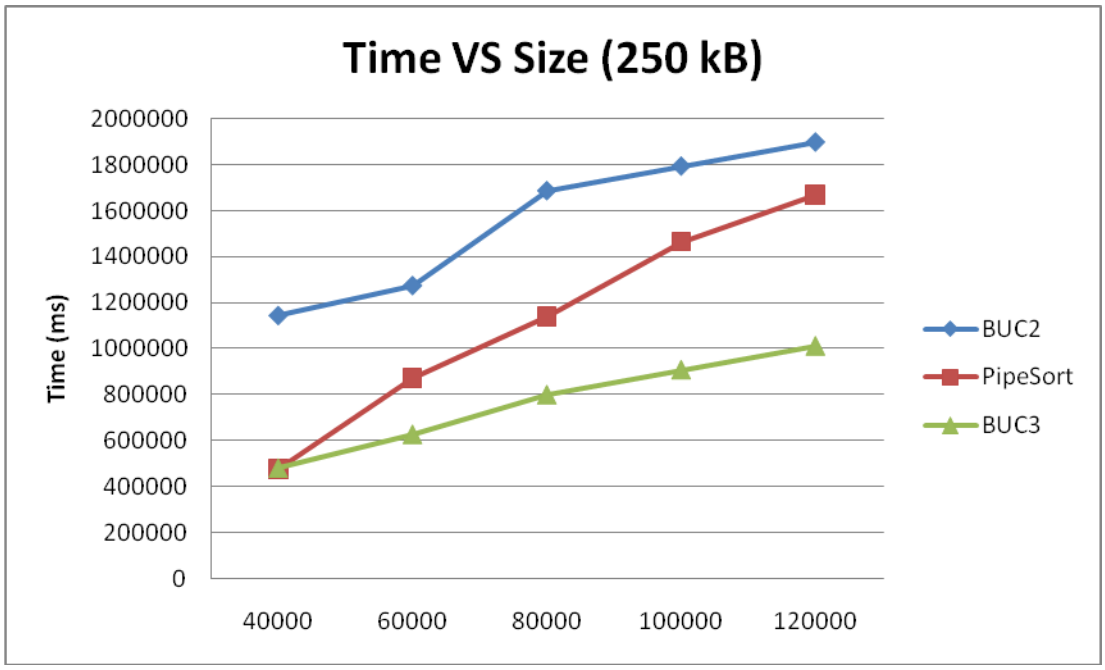

Figure 43. Processing time versus size (250 kB)

6. Discussion

Our evaluation helps us achieve the goals of our research. The main goal of our research is to provide a heuristic implementation on multi-dimensional partitioning in BUC. Based on our results, we found that our implementation was promising and considerable. The optimization we implemented also improved the performance of BUC. We were also able to compare PipeSort and BUC as well as to identify the characteristics of the algorithms, which agreed with other researchers in this area.

Our results confirmed that our implementation for multi-dimensional partitioning in BUC was promising. In most of the tests that involved BUC3, the performance of BUC3 was generally better by a margin than the performance of

PipeSort. BUC is an algorithm that strongly demanded to have sufficient memory. Without sufficient memory, BUC not only slowed down in terms of performance, but also stopped functioning. Our heuristic approach enabled BUC to continue operating in such extreme memory condition; therefore, we can still benefit the high performance of BUC by using multi-dimensional partitioning.

Generating a new set of processing paths for every two-dimensional partitioning that we encountered was a bad idea. We thought that it was a good idea at the beginning since the data distribution was different from one partition to another. For example, in an incident of two-dimensional partitioning, it is wise to generate a view from View A because the number of tuples in View A for this partitioning is the smallest among other views. Generating child views from View A will require the minimum number of R/W. However, in another incident of two-dimensional partitioning, the number of tuples in View A for this partitioning is no longer the less among the others. It will need to generate child views from another view to reach the minimum cost of R/W. Unfortunately, generating a new set of processing paths for every two-dimensional partitioning involved executing the Hungarian algorithm, which complexity is $O(n^3)$. The generation became a costly operation and slowed down the performance of BUC. As we found out in our evaluation, the processing time of BUC2 in every test was nearly double to the processing time of BUC3. In addition, BUC2 ran slower than PipeSort by from 29% to 158%. Therefore, it is not wise to generate a new set of processing paths for every two-dimensional partitioning. Instead, we generate a

set of processing paths for a two-dimensional partitioned data and reuse the same set of paths for other two-dimensional partitioned data that are under the same single-dimensional partitioned data. Assuming that the data distribution of an input data is even and no skewed data is involved, we could rely on the only set of processing paths, which won't be far from the optimal paths for every other partition.

According to our evaluation, the optimization that we applied on BUC2 improved the performance of BUC1 by an average of 15% in terms of processing time. The optimization was designed to reduce the number of R/W and maximize the usage on the memory by not reading and writing duplicated data in tuples. We found that the optimization cut down the number of R/W by an average of 14.7% and the number of partitioning by an average of 35%. We recommended on implementing this optimization for any BUC algorithms that required partitioning.

Our evaluation drew the similar results as other researchers regarding the characteristics of PipeSort and BUC. First of all, our results showed that the time complexity and space complexity of PipeSort and BUC was exponentially proportional to the number of dimensions [2] [8]. Although the complexity of both algorithms was exponentially proportional to the number of dimensions, the increasing rate of PipeSort was much higher than the increasing rate of BUC. One of the possible reasons was that the number of R/W of BUC was only about

42% of the number of R/W of PipeSort in our evaluation. This told us that how effective BUC was on utilizing the memory and reducing I/O costs. Second of all, PipeSort didn't scale well on processing sparse data [2] [6]. In our evaluation, the processing time of PipeSort on processing sparse data (sparsity at 0.1) was increased by 357% of the processing time of PipeSort on processing dense data (sparsity at 0.9). Likewise, the processing time of BUC was increased by only 218%, not to mention that the processing time of BUC was much less than that of PipeSort. These results showed that PipeSort was good at processing dense data while BUC was good at processing sparse data.

7. Future of Work

One of the possible directions of this research is to implement the complete solution of multi-dimensional partitioning and perform evaluation. We believed that, in the extreme memory condition, the logic of the algorithm will be very similar to PipeSort and the performance of this algorithm will not be able to catch up with the performance of PipeSort. If this is the situation, we should investigate this turning point on when we should switch using PipeSort instead.

Another possible direction of this research is to analyze the performance on processing skew data. As we have discussed, our assumption of using one set of processing paths for all the two-dimensional partitioning under the same single-dimensional partitioning was based on the fact that the input data is evenly distributed. If the input data is skew, selecting the optimal set of processing

paths will become critical. This is because the processing paths are highly unlikely to be the optimal processing paths for most of the two-dimensional partitioning.

The last, but not least, possible direction of this research is to estimate the size of views. Having better estimation on the size of views will allow us to generate a better set of processing paths. In fact, size estimation has always been one of the active research areas. Since we don't need to generate the processing paths until multi-dimensional partitioning is required, we will be able to collect more information and/or statistics of the input data to estimate the size of views. We imagined that this problem will not be as difficult as a generic size estimation problem since our situation allows us to read or analyze parts of the input data during the processing time.

8. Conclusion

Our main goal in this research is to provide a practical implementation of multi-dimensional partitioning in BUC. Our implementation extends from single-dimensional partitioning, in which the new partition cannot fit in memory. We partition the data by an additional dimension and run BUC on those new partitions. Since this computation allows us to generate the aggregated tuples for partial views, we have to apply our modified version of PipeSort to generate the missing tuples. The modified PipeSort makes use of the generated tuples and generates the optimal set of processing paths, which are used to compute

the missing tuples. The results are promising and its performance is better than the performance of PipeSort in many perspectives in our evaluation. We believe our approach lowers the memory requirement of BUC data cubing algorithm.

The second goal of our research is to compare the performance of a top-down data cubing algorithm and a bottom-up data cubing algorithm on processing input data with different kinds of properties. We selected PipeSort and BUC as the candidates for our evaluation because both of them are the best representatives of their own types of data cubing algorithms. We tested the algorithm with data in different numbers of dimensions, different levels of sparsity, and different size of memory. Our results show agreements with the findings from existing literatures. First of all, the performance of a data cubing algorithm is exponentially proportional to the number of dimensions. Secondly, PipeSort performs better on dense data and BUC performs better on sparse data. Finally, the performance of PipeSort does not scale well as the number of dimensions increases because of the increase of the number of I/Os and external sorting. Comparing PipeSort and BUC side-by-side shows us the pros and cons of both types of data cubing algorithms.

REFERENCES

[1] Agarwal, S., Agrawal, R., Deshpande, P., Gupta, A., Naughton, J. F., Ramakrishnan, R., & Sarawagi, S. (1996). On the computation of multidimensional aggregates. *Proceedings of Very Large Data Bases*, 506-521.

[2] Beyer, K. S., & Ramakrishnan, R. (1996). Bottom-up computation of sparse and iceberg cubes. *Proceedings of Very Large Data Bases*, 506-521.

[3] Chaudhuri, S., & Dayal, U. (1997). *An overview of data warehousing and OLAP technology. SIGMOD Record, 26, 1*, 65-74.

[4] Gray, J., Bosworth, A., Layman, A., & Pirahesh, H. (1996). Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-total. *Proceedings of ACM Special Interest Group on Management of Data*, 463-474.

[5] Morfonios, K., & Ioannidis, Y. (2006). CURE for cubes: cubing using a ROLAP engine. *Proceedings of Very Large Data Bases*, 379-360.

[6] Morfonios K., Konakas, S., Ioannidis, Y., & Kotsis N. (2007). ROLAP implementations of the data cube. *ACM Computing Surveys, 39, 4*, 12.

[7] Papadimitriou, C. H., & Steiglitz, K. (1998). *Combinatorial optimization: algorithms and complexity*. New York: Mineola.

[8] Ross, K. A., & Srivastava, D. (1997). Fast computation of sparse datacubes. *Proceedings of the International Conference on Very Large Databases*, 116-125.

[9] Runapongsa, K., Nadeau, T. P., & Teorey T. J. (1999). Storage estimation for multidimensional aggregates in OLAP. *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Resarch*, 10.

[10] Sarawagi, S., Agrawal, R., & Gupta, A. (1996). On computing the data cube. *Technical Report RJ10026*. San José, CA: IBM Almaden Research Center.

[11] Wang, W., Feng, J., Lu, H., & Yu, J.X. (2002). Condense cube: an efficient approach to reducing data cube size. *Proceedings of International Conference on Data Engineering*, 155-165.

APPENDIX

Table 1

Benchmark of BUC1 on data with different number of dimensions in 10 MB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W
4	100000	0.5	15242	1069182
6	66666	0.5	29297	1949961
8	50000	0.5	92570	5888334
10	40000	0.5	327226	20441152
12	33333	0.5	1529455	95392351

Table 2

Benchmark of BUC2 on data with different number of dimensions in 10 MB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W
4	100000	0.5	14679	1069182
6	66666	0.5	28470	1949961
8	50000	0.5	89950	5888334
10	40000	0.5	314356	20441152
12	33333	0.5	1460971	95392351

APPENDIX (CONT'D)

Table 3

Benchmark of PipeSort on data with different number of dimensions in 10 MB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W
4	100000	0.5	216949	2337397
6	66666	0.5	202363	5088524
8	50000	0.5	394196	15231206
10	40000	0.5	954158	46981580
12	33333	0.5	5068487	208436485

Table 4

Benchmark of BUC1 on data with different number of dimensions in 1 MB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W	Number of 1-D Partitioning
4	100000	0.5	72275	5369182	88
6	66666	0.5	105783	7883235	48
8	50000	0.5	188136	13438334	40
10	40000	0.5	436020	29601152	40
12	33333	0.5	1624537	106158910	36

APPENDIX (CONT'D)

Table 5

Benchmark of BUC2 on data with different number of dimensions in 1 MB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W	Number of 1-D Partitioning
4	100000	0.5	57018	4369182	66
6	66666	0.5	82821	6216585	32
8	50000	0.5	152692	11138334	25
10	40000	0.5	375476	25841152	20
12	33333	0.5	1500922	101792287	18

Table 6

Benchmark of PipeSort on data with different number of dimensions in 1 MB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W
4	100000	0.5	217183	8131365
6	66666	0.5	269365	13410452
8	50000	0.5	511322	26483110
10	40000	0.5	1116944	60797620
12	33333	0.5	6073283	292890373

APPENDIX (CONT'D)

Table 7

Benchmark of BUC2 on data with different number of dimensions in 500 kB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W	Number of 1-D Partitioning
4	100000	0.5	61261	4569182	88
6	66666	0.5	93444	7083243	48
8	50000	0.5	170477	12538334	35

Table 8

Benchmark of PipeSort on data with different number of dimensions in 500 kB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W
4	100000	0.5	167185	8131365
6	66666	0.5	238992	13410452
8	50000	0.5	659006	41647294

APPENDIX (CONT'D)

Table 9

Benchmark of BUC2 on data with different number of dimensions in 250 kB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W	Number of 1-D Partitioning	Number of 2-D Partitioning
4	100000	0.5	61261	4569182	88	0
6	66666	0.5	93943	7083243	48	0
8	50000	0.5	1224662	22746185	40	60

Table 10

Benchmark of BUC3 on data with different number of dimensions in 250 kB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W	Number of 1-D Partitioning	Number of 2-D Partitioning
4	100000	0.5	61261	4569182	88	0
6	66666	0.5	93943	7083243	48	0
8	50000	0.5	558683	22746185	40	60

APPENDIX (CONT'D)

Table 11

Benchmark of PipeSort on data with different number of dimensions in 250 kB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W
4	100000	0.5	139714	8131365
6	66666	0.5	266666	17274912
8	50000	0.5	729207	48879022

Table 12

Benchmark of BUC1 on data with different sparsity in 10 MB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W
8	60000	0.1	190570	12340629
8	60000	0.3	119933	7728959
8	60000	0.5	106361	6855519
8	60000	0.7	89653	5723761
8	60000	0.9	57377	3640457

APPENDIX (CONT'D)

Table 13

Benchmark of BUC2 on data with different sparsity in 10 MB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W
8	60000	0.1	185437	12340629
8	60000	0.3	117062	7728959
8	60000	0.5	104005	6855519
8	60000	0.7	87033	5723761
8	60000	0.9	56597	3640457

Table 14

Benchmark of PipeSort on data with different sparsity in 10 MB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W
8	60000	0.1	950961	29727401
8	60000	0.3	629663	20624566
8	60000	0.5	524784	18105547
8	60000	0.7	409547	14993546
8	60000	0.9	261815	9927459

APPENDIX (CONT'D)

Table 15

Benchmark of BUC1 on data with different sparsity in 1 MB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W	Number of 1-D Partitioning
8	60000	0.1	305697	21400629	48
8	60000	0.3	235934	16788959	40
8	60000	0.5	222160	15915519	40
8	60000	0.7	205374	14783761	40
8	60000	0.9	173378	12700457	40

Table 16

Benchmark of BUC2 on data with different sparsity in 1 MB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W	Number of 1-D Partitioning
8	60000	0.1	260769	18640629	30
8	60000	0.3	193035	14028959	25
8	60000	0.5	180492	13155519	25
8	60000	0.7	166296	12023761	25
8	60000	0.9	135517	9940457	25

APPENDIX (CONT'D)

Table 17

Benchmark of PipeSort on data with different sparsity in 1 MB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W
8	60000	0.1	1336686	72585905
8	60000	0.3	1042345	57147426
8	60000	0.5	658195	31725611
8	60000	0.7	532288	27583882
8	60000	0.9	377567	21018115

Table 18

Benchmark of BUC2 on data with different sparsity in 500 kB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W	Number of 1-D Partitioning
8	60000	0.1	281689	20320629	42
8	60000	0.3	213813	15708959	35
8	60000	0.5	201879	14835519	35
8	60000	0.7	184283	13703761	35
8	60000	0.9	153848	11620457	35

APPENDIX (CONT'D)

Table 19

Benchmark of PipeSort on data with different sparsity in 500 kB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W
8	60000	0.1	1439396	90477065
8	60000	0.3	926109	57147426
8	60000	0.5	806988	50337491
8	60000	0.7	665122	42184594
8	60000	0.9	349362	21018115

Table 20

Benchmark of BUC2 on data with different sparsity in 250 kB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W	Number of 1-D Partitioning	Number of 2-D Partitioning
8	60000	0.1	1743503	43740438	48	90
8	60000	0.3	1544463	30356605	40	85
8	60000	0.5	1273630	27190059	40	80
8	60000	0.7	1234178	24759478	40	80
8	60000	0.9	1167020	20095843	40	80

APPENDIX (CONT'D)

Table 21

Benchmark of BUC3 on data with different sparsity in 250 kB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W	Number of 1-D Partitioning	Number of 2-D Partitioning
8	60000	0.1	882071	43740438	48	90
8	60000	0.3	671159	30356605	40	85
8	60000	0.5	626684	27190059	40	80
8	60000	0.7	588245	24759478	40	80
8	60000	0.9	514988	20095843	40	80

Table 22

Benchmark of PipeSort on data with different sparsity in 250 kB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W
8	60000	0.1	1348760	90477065
8	60000	0.3	986902	66986946
8	60000	0.5	871010	58747691
8	60000	0.7	636371	42184594
8	60000	0.9	452228	30456159

APPENDIX (CONT'D)

Table 23

Benchmark of BUC1 on data with different number of tuples in 10 MB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W
8	40000	0.5	72353	4548039
8	60000	0.5	106361	6855519
8	80000	0.5	128825	8299864
8	100000	0.5	144066	9324829
8	120000	0.5	157310	10117853

Table 24

Benchmark of BUC2 on data with different number of tuples in 10 MB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W
8	40000	0.5	69670	4548039
8	60000	0.5	104005	6855519
8	80000	0.5	125783	8299864
8	100000	0.5	140385	9324829
8	120000	0.5	152505	10117853

APPENDIX (CONT'D)

Table 25

Benchmark of PipeSort on data with different number of tuples in 10 MB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W
8	40000	0.5	266760	11582967
8	60000	0.5	524784	18105547
8	80000	0.5	796115	22784125
8	100000	0.5	1071767	26360213
8	120000	0.5	1347247	29265552

Table 26

Benchmark of BUC1 on data with different number of tuples in 1 MB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W	Number of 1-D Partitioning
8	40000	0.5	149339	10588039	40
8	60000	0.5	222160	15915519	40
8	80000	0.5	282766	20379864	40
8	100000	0.5	336399	24424829	40
8	120000	0.5	388252	28237853	40

APPENDIX (CONT'D)

Table 27

Benchmark of BUC2 on data with different number of tuples in 1 MB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W	Number of 1-D Partitioning
8	40000	0.5	101758	7308039	15
8	60000	0.5	180492	13155519	25
8	80000	0.5	245685	17899864	30
8	100000	0.5	305043	22624829	35
8	120000	0.5	348941	26077853	35

Table 28

Benchmark of PipeSort on data with different number of tuples in 1 MB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W
8	40000	0.5	373121	20373527
8	60000	0.5	658195	31725611
8	80000	0.5	1184571	65710229
8	100000	0.5	1396621	78449633
8	120000	0.5	1601091	89367768

APPENDIX (CONT'D)

Table 29

Benchmark of BUC2 on data with different number of tuples in 500 kB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W	Number of 1-D Partitioning	Number of 2-D Partitioning
8	40000	0.5	129387	9348039	30	0
8	60000	0.5	201879	14835519	35	0
8	80000	0.5	1304535	29217059	40	25
8	100000	0.5	1446354	39144149	40	60
8	120000	0.5	1808852	48667438	40	90

Table 30

Benchmark of BUC3 on data with different number of tuples in 500 kB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W	Number of 1-D Partitioning	Number of 2-D Partitioning
8	40000	0.5	129387	9348039	30	0
8	60000	0.5	201879	14835519	35	0
8	80000	0.5	669224	29217059	40	25
8	100000	0.5	797113	39144149	40	60
8	120000	0.5	941912	48667438	40	90

APPENDIX (CONT'D)

Table 31

Benchmark of PipeSort on data with different number of tuples in 500 kB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W
8	40000	0.5	501649	31616395
8	60000	0.5	806988	50337491
8	80000	0.5	1057727	65710229
8	100000	0.5	1280339	78449633
8	120000	0.5	1453952	89367768

Table 32

Benchmark of BUC2 on data with different number of tuples in 250 kB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W	Number of 1-D Partitioning	Number of 2-D Partitioning
8	40000	0.5	1143106	17322192	40	40
8	60000	0.5	1273630	27190059	40	80
8	80000	0.5	1686329	39373239	40	105
8	100000	0.5	1793298	47704529	40	140
8	120000	0.5	1897631	55188566	40	145

APPENDIX (CONT'D)

Table 33

Benchmark of BUC3 on data with different number of tuples in 250 kB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W	Number of 1-D Partitioning	Number of 2-D Partitioning
8	40000	0.5	479326	17322192	40	40
8	60000	0.5	626684	27190059	40	80
8	80000	0.5	799281	39373239	40	105
8	100000	0.5	908512	47704529	40	140
8	120000	0.5	1012580	55188566	40	145

Table 34

Benchmark of PipeSort on data with different number of tuples in 250 kB

Number of Dimensions	Number of Tuples	Sparsity	Time (ms)	R/W
8	40000	0.5	474723	31616395
8	60000	0.5	871010	58747691
8	80000	0.5	1136975	76919381
8	100000	0.5	1464232	99299489
8	120000	0.5	1668607	113355336