# MULTIDIMENSIONAL AGGREGATIONS IN PARALLEL DATABASE SYSTEMS

---

A Dissertation Presented to

the Faculty of the Department of Computer Science

University of Houston

---

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

---

By

Yiqun Zhang

December 2017

# MULTIDIMENSIONAL AGGREGATIONS IN PARALLEL DATABASE SYSTEMS

---

Yiqun Zhang

APPROVED:

---

Carlos Ordonez, Ph.D., Chairman
Dept. of Computer Science

---

Omprakash Gnawali, Ph.D.
Dept. of Computer Science

---

Xin Fu, Ph.D.
Dept. of Electrical & Computer Engineering

---

Lennart Johnsson, Ph.D.
Dept. of Computer Science

---

Stephen Huang, Ph.D.
Dept. of Computer Science

---

Dean, College of Natural Sciences and Mathematics

# Acknowledgements

I would like to express my sincere thanks to my advisor Dr. Carlos Ordonez for his patience, guidance, and support during my study at the University of Houston. His knowledge, insights, and generous help gave me high confidence in my efforts to undertake the challenge of the research projects that this dissertation presents.

I am very grateful to Dr. Omprakash Gnawali, Dr. Xin Fu, Dr. Stephen Huang, and Dr. Lennart Johnsson for taking the time to serve on my dissertation committee and sharing with me their knowledge and experience. It is my privilege to work with my fellow researchers, Dr. Wellington Cabrera and Dr. David Matusevich. I want to thank them for the friendship and teamwork we had together during the past five years.

I want to thank all my colleagues at VoltDB Inc. especially Bruce Reading, John Piekos, and Ning Shi who offered me the most solid and unwavering support when I had to play the role of both a Ph.D. student and a software engineer during the last year of my Ph.D. study.

I also want to thank my parents Jingli and Desheng, and other family members especially my cousin Yuanfeng Wen and my sister-in-law Wensi Song. Their unconditional love has always been the strongest emotional support that I am so blessed to have through my graduate study here, more than 7,000 miles away from home, in the United States.

Last but not least, my sincere thanks go to my dear friends Yifei Wan, Alex Zhou, Fangkai Yang, Zihao Zhou, Lei Qin, Le Chen, and many, many others. It has been such an amazing journey with your company and support.

# MULTIDIMENSIONAL AGGREGATIONS IN PARALLEL DATABASE SYSTEMS

---

An Abstract of a Dissertation Presented to

the Faculty of the Department of Computer Science

University of Houston

---

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

---

By

Yiqun Zhang

December 2017

# Abstract

Aggregations help computing summaries of a data set, which are ubiquitous in various big data analytics problems. In this dissertation, we provide two major technical contributions which work on parallel database systems that significantly extend the capabilities of aggregations, studying two complementary multidimensional mathematical structures: cubes and matrices. Cubes present a combinatorial problem in a set of discrete dimensions, widely studied in database systems. On the other hand, matrices are widely used in machine learning models, requiring iterative numerical methods taking input as multidimensional vectors. Both problems are difficult to solve on large data sets residing on secondary storage, and their algorithms are difficult to optimize in a parallel cluster. First, we extend cubes to intuitively show the relationship between measures aggregated at different grouping levels by introducing the percentage cube, a generalized database cube that takes percentages as its basic measure instead of simple sums. We show that the percentage cube is significantly harder to compute than the standard cube due to a higher exponential complexity. We propose SQL syntax and introduce novel query optimizations to materialize the percentage cube without any memory limitations. We compare our optimized queries with existing SQL functions, evaluating time, speed-up, and effectiveness of lattice pruning methods. Besides, we show columnar storage provides significant acceleration over row storage, the standard storage mechanism. Second, we study parallel aggregation on large matrices stored as tables and study how to compute a comprehensive data summarization matrix, called Gamma. Gamma generally fits in main memory, and it is shown to enable the efficient derivation of many

machine learning models. Specifically, we show our Gamma summarization matrix benefits linear machine learning models, including PCA, linear regression, classification, and variable selection. We analytically show our summarization matrix captures essential statistical properties of the data set and we experimentally show Gamma allows iterative algorithms to iterate faster in main memory. Also, we show Gamma is further accelerated with array and columnar storage. We experimentally prove our parallel aggregations allow faster computation than existing machine learning libraries for model computations in R and Spark, two popular analytic platforms.

# Contents


**1 Introduction** **1**

**2 Related Work** **6**

2.1 Online Analytical Processing . . . . . 6

2.2 Faster Analytic Algorithms . . . . . 8

2.3 Data Summarization . . . . . 9

2.4 Scalable Matrix Multiplication . . . . . 11

**3 Percentage Cube** **13**

3.1 Definition . . . . . 13

3.1.1 Percentage Aggregation . . . . . 13

3.1.2 Standard Cube . . . . . 14

3.1.3 Percentage Cube . . . . . 14

3.1.4 Example . . . . . 15

3.2 SQL Syntax . . . . . 18

3.2.1 Percentage Aggregation . . . . . 18

3.2.2 Percentage Cube . . . . . 21

3.3 Evaluation Method . . . . . 22

3.3.1 Evaluating Percentage Aggregation . . . . . 22

3.3.2 Evaluating Percentage Cube . . . . . 28

3.4 Incremental Computation . . . . . 31

3.5 Pruning the Percentage Cube . . . . . 33

3.5.1 Row Count Threshold . . . . . 34

3.5.2 Percentage Threshold . . . . . 39

3.5.3 Top-$k$ percentages . . . . . 40

3.6 Experiments . . . . . 41

3.6.1 Experimental Setup . . . . . 41
3.6.2 Comparing Percentage Aggregation Methods . . . . . 44
3.6.3 Performance of the incremental computation . . . . . 46
3.6.4 Comparing Percentage Cube Materialization Methods . . . . . 48
3.6.5 Comparing Cube Pruning Mechanisms . . . . . 48

**4 Matrix Aggregation: The Gamma operator 52**

4.1 Definition . . . . . 52
4.2 Gamma Summarization Matrix . . . . . 53
4.3 Two-Phase Analytic Algorithm . . . . . 56
4.4 Models Exploiting the Gamma Matrix . . . . . 58
4.4.1 Principal Component Analysis (PCA) . . . . . 59
4.4.2 Linear Regression (LR) . . . . . 61
4.4.3 Variable Selection (VS) . . . . . 62
4.5 Time Complexity and Parallel Speedup . . . . . 65
4.6 Data Summarization in Parallel Database Systems . . . . . 66
4.6.1 Data summarization in SciDB . . . . . 67
4.6.2 Programming Mechanisms in SciDB . . . . . 67
4.6.3 The Gamma Array Operator to Summarize a Matrix . . . . . 68
4.6.4 Additional Considerations . . . . . 73
4.6.5 Accelerating Summarization with a GPU . . . . . 76
4.7 Data summarization in Columnar DBMS . . . . . 78
4.8 Experiments . . . . . 81
4.8.1 Setup . . . . . 82
4.8.2 Comparing Analytic Systems (N = 1 node) . . . . . 85
4.8.3 Parallel Processing (N = 1, 10, 100 nodes) . . . . . 92

4.8.4 Fast Data Summarization with a GPU . . . . . 95

**5 Conclusions** **98**

**Bibliography** **102**

# List of Figures


3.1 Drill-down percentage cube visualization . . . . . 18

3.2 Expansion from data cube to percentage cube. . . . . 29

3.3 Dimensions as binary codes. . . . . 39

4.1 The workflow for a system using $\Gamma$ for model computation . . . . . 57

4.2 How the Gamma operator works in SciDB . . . . . 75

4.3 The horizontal and vertical layout for $X$ in the columnar DBMS. . . 79

4.4 Comparing systems to compute $\Gamma$ on a local server: Array DBMS, R, and SQL. . . . . 89

4.5 $\Gamma$ computation time: CPU vs. GPU. . . . . 95

# List of Tables


3.1 An example fact table, $F$, with two dimensions. . . . . 15

3.2 An OLAP cube built on top of the fact table $F$ . . . . . 16

3.3 A percentage cube built on top of the fact table $F$. . . . . 17

3.4 $pct(salesAmt)$ on table $F$. . . . . 21

3.5 Summary of grouping columns for individual percentage queries transactionLine ($N$=6M). . . . . 42

3.6 Candidate cube dimensions' cardinalities. . . . . 42

3.7 Selected cube dimensions at various $d$. . . . . 43

3.8 Percentage Aggregation: GROUP-BY vs. OLAP. Scale factor=4,N=24M (times in secs). . . . . 45

3.9 Cube Generation: GROUP-BY vs. OLAP. Scale factor=1, N=6M (times in secs). . . . . 45

3.10 Direct pruning vs. Cascaded pruning (times in secs). . . . . 46

3.11 Time to evaluate the percentage cube with nulls in the measure. n = 20M, d = 4 (times in secs). . . . . 46

3.12 Incremental computation, ONE query ($n$ =20M, $d = 4$, $\delta = 1\%$, times in secs). . . . . 47

3.13 Incremental computation ($n$ =20M, $\delta = 1\%$, times in secs). . . . . 47

3.14 Top $k$ percentages (times in secs). . . . . 50

3.15 Pruning method: row count threshold vs. top $k$, n = 20M (times in secs). . . . . 51

4.1 Summary of matrices: summarization and models. . . . . 57

4.2 Data sets. . . . . 82

4.3 Hardware and operating system. . . . . 83

4.4 Comparing computation of model $\Theta$ using R and DBMS+R; dense matrix operator; data set KDDnet; local server; times in secs. . . . . 86

4.5 Bayesian variable selection (VS) in gene data set ($d = 12506, n = 248$): Comparing R+SciDB with R at 1000 iterations; times in secs. . . . . 87

4.6 Comparing matrix summarization algorithms; data set KDDnet; dense ($dn$ entries) and sparse (zeroes deleted); Local 1/2; times in secs. . . . 88

4.7 Comparing the dense operator (working on disk+RAM), the sparse operator (working on disk+RAM), and the dense matrix multiplication in MKL BLAS (parallel LAPACK, working only on RAM) to compute $\Gamma$; local server 4GB RAM; synthetic data sets; times in secs. 90

4.8 Parallel speedup: $\Gamma$ dense matrix summarization operator with multi-threaded processing varying $M$=# SciDB instances ($N$=1 node, 4-core CPU); times in secs. . . . . 92

4.9 Parallel processing: Comparing R+SciDB and Spark varing $N$ (# of nodes); data set KDDnet $d = 38$; dense matrix; times in secs. . . . . 93

4.10 Comparing computation of model $\Theta$ using R, DBMS+R, and DBMS+R+GPU; dense matrix operator; N=1 node (CPU=8 cores); times in secs. . . . 97

# Chapter 1

# Introduction

Business and scientific activities are becoming more and more data-driven. Most organizations are now collecting data at the most detailed level possible, and the volume of data is growing very fast at ever-accelerating rates. As a result, the increasing need for analyzing and understanding collected data is drawing more and more people's attention to the great challenges of big data analytics.

Big data analytics is a new data mining trend [20, 1] with higher data volume, varied data types, and parallel processing. Data aggregation is one of the most commonly used approaches to help understand the collected data and provide data findings in a summarized, meaningful format that is useful for the end user. Classic aggregate functions include $sum(), avg(), count(), max()$, and $min()$. However, we believe that an essential aggregation that is missing from the SQL list is the percentage.

Percentages are essential in big data analytics. They can express the proportionate relationship between two amounts summarized at different levels. Sometimes, percentages are less deceiving and more intuitive than absolute values. Therefore, they are suitable for comparisons. Furthermore, percentages can also be used as an intermediate step in some applications for complex analytics. Unlike classic aggregation functions that mostly require only one scan pass over the data set and one

single aggregation level, percentage requires aggregations at two different detail levels (one for the total amount and one for the individual amount). Along with the computation complexity is the lack of support in SQL. Traditional SQL aggregate functions are cumbersome and inefficient to compute percentages given the amount of SQL code that needs to be written and the inability of the query optimizer to efficiently evaluate such aggregations. We explore the problem of percentage aggregation and tackle the challenge by proposing new intuitive, easy-to-use SQL syntax and a series of query optimizations to facilitate the efficient evaluation of percentage aggregations.

Traditional aggregate functions, supported by the GROUP-BY operator, can produce zero-dimensional or one-dimensional summaries of the data set. Data cube, first introduced in [19], generalized the standard "GROUP BY" operator to provide multi-dimensional summaries of the dataset, computing aggregations for every combination of the grouping columns. Building data cubes has been well recognized as one of the most important and essential operations in OLAP. However, exploring percentages we just mentioned in a full data cube is a new problem, which is computationally harder. In this dissertation, we introduce a specialized form of data cube taking percentages as the aggregated measure which we call the percentage cube. A percentage cube shows the fractional relationship in every cuboid between each aggregated measure and its further summed-up measures aggregated by less detailed grouping columns. Unfortunately, existing SQL aggregate functions, OLAP window functions, and Multidimensional Expressions (MDX) are insufficient to compute percentage cubes. The computation is rather complicated to express even with the percentage aggregation syntax we proposed. The exponential number of grouping

column pairs adds further complexity. In this dissertation, we also further explore percentage aggregation in a cube, proposing percentage cube syntax as well as important techniques and optimizations to efficiently evaluate them. We justify that the percentage cube subsumes iceberg queries (based on a decreasing row count threshold) and it represents a harder problem because there are exponentially more groups and it is feasible to find large percentages both at high levels and deep levels in the dimension lattice. Moreover, it is generally necessary to explore percentages interactively. Such challenges make percentage cube materialization (precomputation) mandatory.

Analyzing big data for interesting pattern or trends can go way more complicated than just aggregating relational data in DBMSs. Analysts develop machine learning models from the data set. Beyond simple aggregations on the data, people build much more complicated models from which they can not only understand the data but also make predictions and decisions proactively. The data, in this case, is much better represented as matrices.

From a system perspective, row DBMSs [45] remain the best technology for transaction processing [44] and column DBMSs are becoming a competitor in query processing [41] in large databases [42]. However, both kinds of DBMS are considered slow and difficult to use for mathematical computations on large datasets due to the abundance of matrices. Currently, Hadoop and MapReduce systems [23, 42] (e.g., Spark [50, 47], HadoopDB, Cassandra, Hive, MongoDB, and the Mahout project) are the most popular for big data analytics. From a statistical perspective, R is the most popular platform, but despite many efforts it remains slow and not scalable for big data analytics. Array DBMSs (SciDB [43], Rasdaman [6]) represent alternative

analytic systems, closer to relational DBMSs, that enable computation with large matrices in R, which share many features with old DBMSs like efficient I/O, indexing, concurrency control, and parallel processing, but have significantly different storage and query processing mechanisms. On the algorithm side, there has been extensive research efforts on optimizing machine learning algorithms on large-scale data sets and in parallel. Many algorithms are adapted to work iteratively or using a randomized approach. However, those work still require scanning the original data set multiple times. As the data sets grow explosively, those algorithms can suffer degraded performance due to the I/O bottleneck. In contrast, optimizing machine learning algorithms with data summarization techniques has received scant attention. We focus on the optimization of algorithms to compute three fundamental and complementary models. Specifically, we study the computation of principal component analysis (PCA) [22] with the correlation matrix[48], linear regression (LR) [22] and variable selection (VS) on large datasets. However, these three models require different numerical and statistical methods for their solutions: PCA is generally computed with a Singular Value Decomposition (SVD) on the correlation [48, 22] (covariance) matrix of the data set; LR is solved with least squares via a matrix factorization; VS requires visiting an exponential search space of variable combinations, whose most promising solution is currently found by MCMC methods, used in Bayesian statistics [17]. A costly matrix multiplication plays a very important role in the computation of those models. The root causes are a matrix transposition and then a slow matching between the $i$th row from the input matrix with the $j$th column from its transpose. With that motivation in mind, we introduce a summarization matrix, called Gamma, that captures essential statistical properties of the data set,

to compute the first and the second moment (mean, variance) of multivariate probability distributions. The most important characteristics of the Gamma matrix is that it is much smaller than the data set and it can be used to avoid recomputing costly matrix multiplications. We show the Gamma summarization matrix can be efficiently computed via a novel form of matrix multiplication, optimized for large data sets, in which Gamma can be incrementally updated in main memory. To that end, we introduce two parallel incremental algorithms for dense and sparse large input matrices (i.e. the data set), respectively, which scan the data set once, in parallel. Exploiting Gamma, we introduce a "meta" algorithm that divides the computation of our models into two phases: (1) parallel incremental summarization to get Gamma, (2) model computation in main memory exploiting the Gamma matrix, including the iterative behavior.

Experimental evaluation shows our summarization operator removes main memory limitations from R and it is much faster than R (even when the data set fits in RAM) and orders of magnitude faster than a column DBMS (one of the fastest DBMSs, evaluating summarization with SQL queries). There are efficient functions in R, LAPACK [5, 14], SciDB [43], and Spark [50] to compute matrix multiplication, but we show they are inefficient to summarize the data set. More importantly, we show R and LAPACK fail when there is insufficient RAM for large input matrices. From a parallel perspective, the array DBMS is orders of magnitude faster and has better speedup than Spark, when running in a large cluster in the cloud.

# Chapter 2

# Related Work

## 2.1 Online Analytical Processing

As previously mentioned, there exist OLAP extensions proposed in the ANSI SQL-OLAP [25], an amendment that allows computing percentages in a single, but inefficient, query. These extensions involve windowing and partitioning with the OVER and PARTITION clauses. OLAP extensions are available in Oracle, IBM DB2, HP Vertica, and Teradata. Microsoft SQL Server provides a loosely related SQL extension to get the top or bottom percent of rows according to some numeric expression. These extensions are different from our proposal in several aspects. Their usage, syntax, and optimization are not as simple as ours since they are based on a window of rows. They are more general, but not particularly suitable to compute percentages which we argue it is a very common aggregation. These extensions require specifying another aggregate function as an argument whereas ours only requires calling the $pct()$ function.

Some SQL extensions to help data mining tasks are proposed in [12]. These include a primitive to compute samples and another one to transpose the columns of a table. SQL extensions to perform spreadsheet-like operations with array capabilities are introduced in [46]. Unfortunately, those spreadsheet extensions are not adequate to compute percentage aggregations because their goal is avoiding joins to express cells formulas, but they are not optimized to handle two-level aggregations or perform

transposition. Our optimizations and proposed query generation can be combined with this approach. UDFs represent a programming mechanism to materialize and query the cube in RAM [10], while maintaining a tight integration with the DBMS. This approach allows processing the input table directly, using the cube in RAM as a proxy of the fact table and then evaluating SQL cuboid queries on the cube. Another closely related approach is a horizontal aggregation [36], which presents multi-row results in pivoted form. This approach enables more intuitive understanding of all percentages within a cuboid.

Extending data cubes to support more types of aggregations has been explored in [39, 40], mainly focusing on aggregating textual data. Percentage aggregation queries were introduced in [31]. We make several significant contributions beyond this paper. Specifically, we refine the definition of left keys and right keys in the function calls with the introduction of "BREAKDOWN BY" and "TOTAL BY" clauses. We believe our new syntax is more intuitive. We also improve the OLAP evaluation method with the $row\_number()$ approach, being twice faster than the previous "DISTINCT" approach because we avoid an external sort operation. Nevertheless, the GROUP BY method remains the winner. We revisit query processing in a columnar DBMS, which presents new challenges. More importantly, we generalized percentage aggregation queries to the percentage cube, which is a significantly harder problem and even harder than standard cubes. To the best of our knowledge, percentage cubes had never been explored before. Moreover, traditional pruning techniques need to be adapted. Finally, preliminary research on percentage aggregation appeared in [53]. Further research on percentage cube appeared in [54]. The main differences are the following. We introduce an incremental algorithm. We consider nulls, which require

different semantics from traditional SQL. We introduce two alternative methods to prune the cube: a minimum percentage threshold and getting the top $k$ largest percentages. We showed top $k$ percentages is a harder problem, resulting in a holistic [19] aggregation.

## 2.2 Faster Analytic Algorithms

Research has developed fast algorithms based mostly on sampling, data summarization, and gradient descent [18, 24], but generally working in a sequential manner (data mining) or outside a parallel data analysis system (Hadoop or DBMS). Sample-based algorithms [15] require mechanisms to control approximation error, and they are hard to program in parallel for data sets with skewed distributions. In our case, no sampling is required, although it can further accelerate our summarization algorithms. Stochastic (incremental) gradient descent (SGD) [23] is another popular approach, useful when there is a convex function to optimize (like least squares in LR). As for drawbacks, SGD is naturally sequential (difficult to process in parallel), it obtains an approximate solution, and it is difficult to adapt to non-convex functions (e.g., clustering). On the other hand, we believe we are the first to study parallel data summarization as matrix multiplication. Given their relevance to our work, data summarization and matrix multiplication are discussed in more detail below.

Despite the prominence of statistical and machine learning models, there is not much work studying how to accelerate their computation inside a DBMS. Accelerating SVD has received attention [29], but producing approximate solutions and

without considering parallelism. Linear classification with Support Vector Machines (SVMs) is not a problem we considered, but it is straightforward to define a summarization operator for a diagonal matrix $\Gamma$ to get a reasonably accurate Naive Bayes classifier. The importance of aggregate UDFs to accelerate the computation of statistical models in relational DBMSs is identified in [35], making a big step forward compared to a pure query-based approach [32]. Following the same ideas, [23] introduces the MADlib library to compute statistical models with SQL mechanisms, combining queries and UDFs. Nevertheless, neither [35] nor [23] had envisioned matrix multiplication as a vector-based outer product computation which can be pushed into an aggregation operator with arrays both as input and output. Moreover, the integration with R was not considered.

## 2.3 Data Summarization

Our summarization matrix was first proposed in [37] to solve PCA, linear regression, and variable selection. Our recent research [38] explored the optimizations for sparse input data sets and showed that our SciDB implementation defeated Spark in performance on a large cluster on the cloud. In [52], we migrated our algorithm to a relational columnar DBMS and its performance is very close to that of SciDB. A similar, but less general data summarization to ours was pioneered in [51] to accelerate the computation of distance-based clustering: the sums of values and the sums of squares. Later, [8] exploited such summaries as multidimensional sufficient statistics for the K-means and EM clustering algorithms. The main differences with

[51] and [8] are: data summaries were useful only for one model (clustering). Compared to our proposed matrix, their summaries represent a (constrained) diagonal version of $\Gamma$ because dimension independence is assumed (i.e. cross-products, covariances, correlations, are ignored) and there is a separate vector to capture $L$. From a computational perspective, our summarization algorithm boils down to one matrix multiplication, whereas those algorithms work are aggregations. Another major difference is that in our models, one summarization matrix is sufficient, whereas those clustering models need more than one matrix. In short, our summarization is more general and it can help computing more complex models like PCA, LR, and VS that could not be solved with older summaries. From a statistical perspective, we have identified the product of the extended matrix $Z$ with itself as a fundamental summarization for a data set covering zeroth, first, and second moments of its probabilistic distribution. A more general data summarization capturing up to the fourth moment was proposed in [16], but it relies on binning (i.e. building histograms) which are incompatible with most statistical methods. Parallel processing for data summarization has received moderate attention. Reference [27] highlights the following techniques: sampling, incremental aggregation, matrix factorization, and similarity joins. Our proposal is a combination of incremental aggregation and scalable matrix multiplication that enables fast matrix factorization in main memory. A closely related work that identified the linear sum of points $L$ and the quadratic sum of points $Q$ with cross-products is [35], but it did not recognize the importance of expressing the computation as a single matrix product, did not study parallel processing, did not consider sparse matrices, and did not synthesize the 2-step algorithm, which has potential for new models. From a "systems" angle, array DBMSs supporting UDFs did not exist and it was not envisioned the summarization step could be solved entirely

by the DBMS and the iterative and mathematical computations being evaluated in R.

## 2.4 Scalable Matrix Multiplication

Matrix multiplication has been extensively studied with dense and sparse matrices, as well as one processor (sequentially) or $N$ processors (in parallel) [5, 13]: parallel sparse matrix multiplication is the hardest combination. The specific assumptions about matrix shape, density, and parallel computation model vary widely. Most research has proposed algorithms partitioning and storing the input matrices by block in distributed memory [13, 14]. Dense matrix multiplication in main memory in one computer is considered as a solved problem for the most part [13]. Even though multiplication of large matrices exceeding RAM limits (out-of-core) has been studied before [13], nobody has looked at the case that the input matrix (data set) is large, but the result matrix is much smaller and dense. In other words, most research has focused on dense-dense multiplication producing a dense matrix or sparse-sparse multiplication producing a sparse matrix. From a parallel perspective, most algorithms work in a multi-core CPU or GPU. In consequence, parallel matrix multiplication algorithms for matrices residing on secondary storage have not made their way into ScaLAPACK because of the complexity of combining parallel computation with MPI and efficient I/O on disk. MPI is not an efficient interface for DBMS technology because it has an underlying shared-memory model that requires transmitting data across nodes.

Sparse and parallel matrix multiplication are harder problems [4, 9, 26, 49], where

the parallel computation model is shared memory via MPI. That is, such approaches are generally incompatible with shared-nothing architectures like Hadoop and parallel DBMSs. However, there is some work from the database systems angle to multiply sparse matrices for graph analytics [34]. As additional differences, most algorithms assume matrices of arbitrary shape and size, as long as they are compatible with matrix multiplication and they reduce the problem to a matrix-vector multiplication. In contrast, our matrix summarization operator is significantly different (but useful in big data analytics), because it is optimized for a single rectangular matrix as input (in general the shape of a large data set), the input matrix is block-partitioned only along the largest dimension ($n$), parallelization is based on vector-vector outer products and there is no communication overhead during parallel processing. Last but not least, an array DBMS provides not only fast mechanisms to manipulate large matrices but also an elegant and well-defined programming mechanism to create new matrix operators, something that was not feasible with SQL-based DBMSs.

# Chapter 3

# Percentage Cube

## 3.1 Definition

Let $F$ be a relational table having a primary key represented by a row identifier $i$, $d$ discrete attributes (dimensions) and one (or more) numerical attribute (measures): $F(i, D_1, \ldots, D_d, A)$. Discrete attributes (dimensions) are used to group rows to aggregate the numerical attribute (measure). In general, $F$ can be a temporary table resulting from some queries or a view.

### 3.1.1 Percentage Aggregation

Consider a typical percentage problem, for example, how much is the Q1 sales amount in California accounted for in the total Q1 sales amount. Percentage computations like this involve two levels of aggregations: the individual level that will appear as the numerator in the percentage computation (Q1 sales amount in California), and the total level that will show as the denominator (total Q1 sales amount). In the DBMS, we name the result table of the individual level aggregation "$F_{indv}$" and the total level aggregation "$F_{total}$". Both levels of aggregations aggregate attribute $A$ by different sets of grouping columns.

### 3.1.2 Standard Cube

We consider the standard cube having a set of discrete dimensions, where some dimensions may be hierarchical like location (continent, country, city, state) or time (year, quarter, month, day). In order to simplify exposition and better understand query processing, we compute the cube on a denormalized table $F$ defined below, where all dimensions, including all dimension levels, are available on the fact table $F$. That is, $F$ represents a star schema. Such fact table $F$ enables roll-up/drill-down, slice/dice cube operations using any dimension at any level without join computation. Including joins to explore dimension levels which would significantly complicate studying query processing. We will use $F$ to generate a percentage cube with all the $d$ dimensions and one measure [19].

### 3.1.3 Percentage Cube

Percentage computation in a percentage cube happens in the unit of a cuboid. When talking about a cuboid, we use $G$ to represent its grouping column set, that is, $G$ contains all the dimensions in that cuboid which are not "ALL"s. Also, we let $g = |G|$ to represent the number of the grouping columns in a cuboid. To answer the sales amount question, for example, we need to look at the cuboid $G = \{state, quarter\}$, where no dimension should be "ALL", $g = |G| = 2$. In each cuboid, we use $L = \{L_1, \ldots, L_j\}$ to represent the grouping columns used in the total level aggregation ("total by"). When computing percentages, measures aggregated by $L$ will serve as the total amount (denominator). The total amounts then can be further broken down to individual amounts using some additional grouping columns

$R = \{R_1, \ldots, R_k\}, L \cap R = \emptyset$. Columns in $R$ are called "break down by" columns. Overall, the individual level aggregation will use $L \cup R$ as its grouping columns. In our sales amount example, to get the total level amount (total Q1 sales amount), we need to aggregate the attribute by $L = \{quarter\}$. To add more granularity to the per state level, the aggregation result needs to be further broken down by adding the grouping columns in $R = \{state\}$. Note that set $L$ can be empty, in that case, the percentages are computed with respect to the total sum of $A$ for all rows. The total level and individual level has to differ, therefore $R \neq \emptyset$. In each cuboid where the two levels of aggregation happen, $L \cup R = G$. The percentage will be the quotient of each aggregated measure from the individual level and its corresponding value from the total level. All the individual percentage values derived from the same total level group will add up to 100%.

### 3.1.4 Example

Here we give an example of a percentage cube. Assume, we have a fact table, $F$, storing the sales amounts of a company in the first two quarters of 2017 in some US states as shown in Table 3.1.

Table 3.1: An example fact table, $F$, with two dimensions.

| $i$ | state | quarter | salesAmt (million dollars) |
|---|---|---|---|
| 1 | CA | Q1 | 73 |
| 2 | CA | Q2 | 63 |
| 3 | TX | Q1 | 55 |
| 4 | TX | Q2 | 35 |

The fact table $F$ has two dimensions $D_1$=state and $D_2$=quarter (taken from the

cube time dimension) and only one measure $A = salesAmt$. In order to explore sales, we build the multi-dimensional cube shown in Table 3.2.

Table 3.2: An OLAP cube built on top of the fact table $F$

| state | quarter | salesAmt (million dollars) |
|---|---|---|
| $D_1$ | $D_2$ | $A$ |
| CA | Q1 | 73 |
| CA | Q2 | 63 |
| CA | $ALL$ | 136 |
| TX | Q1 | 55 |
| TX | Q2 | 35 |
| TX | $ALL$ | 90 |
| $ALL$ | Q1 | 128 |
| $ALL$ | Q2 | 98 |
| $ALL$ | $ALL$ | 226 |

From Table 3.2 it is easy to find out the sum of the sales amount grouped by any combination of the dimensions in the fact table. However, the user may be interested in a "pie-chart" style quotient, such as how much Q1 sales amount in California contributed to the total Q1 sales amount. That is when the user wants a percentage. With the standard OLAP cube, we first evaluate the query to get the total Q1 sales amount (128M), then a second query to get the Q1 sales amount in California (73M), and finally, perform the division to get the answer (57%). This process may not look complicated when looking at a single question, but data analysts usually explore the cube with a lot of cube exploration operations (roll-up/drill-down, slice/dice). Therefore, the effort of identifying the individual/total group and evaluating additional queries every time to get percentages becomes a burden in the analysis. Instead, Table 3.3 shows a percentage cube built on top of the fact table $F$.

Table 3.3: A percentage cube built on top of the fact table $F$.

| total by | break down by | state | quarter | salesAmt% |
|---|---|---|---|---|
| $L_1$ | $\{R_1\}$, $\{R_1, R_2\}$ | | | $A$ |
| state | quarter | CA | Q1 | 54% |
| state | quarter | CA | Q2 | 46% |
| state | quarter | TX | Q1 | 61% |
| state | quarter | TX | Q2 | 39% |
| **quarter** | **state** | **CA** | **Q1** | **57%** |
| quarter | state | TX | Q1 | 43% |
| quarter | state | CA | Q2 | 64% |
| quarter | state | TX | Q2 | 36% |
| *ALL* | state | CA | *ALL* | 60% |
| *ALL* | state | TX | *ALL* | 40% |
| *ALL* | quarter | *ALL* | Q1 | 57% |
| *ALL* | quarter | *ALL* | Q2 | 43% |
| *ALL* | state,quarter | CA | Q1 | 32% |
| *ALL* | state,quarter | CA | Q2 | 28% |
| *ALL* | state,quarter | TX | Q1 | 24% |
| *ALL* | state,quarter | TX | Q2 | 16% |

Using this percentage cube table, we can easily answer the question "how much did California contribute to Q1 sales?" with a glance at one row. But this flexibility has a price. Compared to the standard cube table (Table 3.2), each cuboid in the percentage cube is significantly exploded. For instance, for the cuboid $\{state, quarter\}$, we only have four rows of data showing the sales amount in every {state,quarter} combination. On the other hand, in the percentage cube, given all potential dimension combinations for left keys ("total by") and right keys ("break down by"), we have 12 rows of data showing the percentages dividing individual amounts (numerator) by total amounts (denominator).

In a real environment, analysts may not even need to look at this percentage cube

table. Pie charts are considered as a natural visualization of percentages. Percentage cubes are, in this sense, a collection of hierarchical pie charts which users can easily navigate by rolling up or drilling down to choose cube dimensions. Once the computation of a percentage cube is complete, it can be interactively visualized traversing the dimension lattice up and down, without further query evaluations. Figure 3.1 shows a pie chart example.

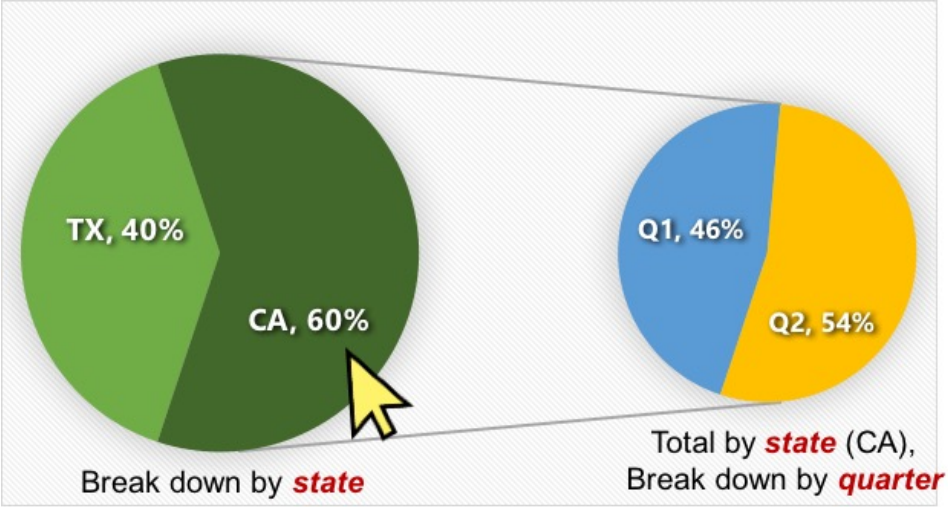



Figure 3.1: Drill-down percentage cube visualization

## 3.2 SQL Syntax

### 3.2.1 Percentage Aggregation

By far, there is no syntax in the standard SQL for percentage aggregations. In this section, we will first propose our *pct*() function to compute them.

$$pct(A \textbf{ TOTAL BY } L_1, \ldots, L_j$$
$$\textbf{BREAKDOWN BY } R_1, \ldots, R_k).$$

The first argument is the expression to aggregate represented by $A$. The next two arguments represent the list of grouping columns used in the total level aggregation and the additional grouping columns to break the total amounts down to the individual amounts. The following SQL statement shows one typical $pct()$ call:

SELECT $L_1, \ldots, L_j, R_1, \ldots, R_k$,
    $pct(A$ TOTAL BY $L_1, \ldots, L_j$
        BREAKDOWN BY $R_1, \ldots, R_k)$
FROM $F$
GROUP BY $L_1, \ldots, L_j, R_1, \ldots, R_k$;

When using the $pct()$ aggregate function, several rules shall be enforced:

1. The "GROUP BY" clause is required because we need to perform a two-level aggregation.

2. Since set $L$ can be empty, the "TOTAL BY" clause inside the function call is optional, but the "BREAKDOWN BY" clause is required because $R \neq \emptyset$. Any columns appeared in either of those two clauses must be listed in the "GROUP BY" clause. In particular, the "TOTAL BY" clause can have as many as $d-1$ columns.

3. Percentage aggregations can be applied to any queries along with other aggregations based on the same GROUP BY clause in the same statement. But

for the simplification and exposition purposes, we do not apply percentage aggregations on queries having joins.

4. When having more than one $pct()$ calls in one single query, each of them can be used with different sub-grouping columns, but still, all of their columns have to be present in the "GROUP BY" clause.

The $pct()$ function computes one percentage per row and has a similar behavior to the standard aggregate functions $sum(), avg(), count(), max()$, and $min()$ that have only one argument. The order of rows in the result table does not have any impact on the correctness, but usually, we return the rows in the order given by the "GROUP BY" clause because rows belong to the same group (i.e. rows making up 100%) are better displayed together. The $pct()$ function returns a real number in the range of [0,1] or NULL when dividing by zero or doing operations with null values. If there are null values, the $sum()$ aggregate function determines the sums to be used. That is, $pct()$ preserves the semantics of $sum()$, which skips null values.

**Example**

We still use our fact table shown in Table 3.1. The following SQL statement shows one specific example that computes the percentage of the sales amount of each state out of every quarter's total.

SELECT $quarter, state$,

$pct(salesAmt$ TOTAL BY $quarter$

BREAKDOWN BY $state$)

FROM $F$

GROUP BY *quarter*, *state*;

In this example, at the total level we first group the total sums by *quarter*, then we further break each group down to individual level by *state*. The result table is shown in Table 3.4.

Table 3.4: $pct(salesAmt)$ on table $F$.

| quarter | state | salesAmt% |
|---|---|---|
| Q1 | CA | 57% |
| Q1 | TX | 43% |
| Q2 | CA | 64% |
| Q2 | TX | 36% |

### 3.2.2 Percentage Cube

Below we propose our SQL syntax to create a percentage cube on the fact table we showed in Table 3.1. When creating a percentage cube, the *pct*() function call will no longer require a "TOTAL BY" or "BREAKDOWN BY" clause.

SELECT *quarter*, *state*, $pct(salesAmt)$ FROM $F$
GROUP BY *quarter*, *state*
**WITH PERCENTAGE CUBE;**

Comparing Table 3.4 and Table 3.3 we will find that a percentage cube is no more than a collection of percentage aggregation results.

## 3.3 Evaluation Method

### 3.3.1 Evaluating Percentage Aggregation

The *pct*() function call can be unfolded and evaluated in standard SQL. The general idea can be described as the following two steps:

1. Evaluate the two levels of aggregations, respectively.

2. Compute the quotient of the aggregated measures as the individual percentage value from $F_{indv}$ and $F_{total}$ where both of them match in their $L$ (total-by) columns.

In practice, how do we compute the two levels of aggregations is the key factor to distinguish the evaluation methods. In this section, we introduce two methods: the first one is the OLAP window method exploiting window functions, and the second one is the GROUP-BY method using standard aggregations.

#### The OLAP Window Method

We first consider SQL built-in functions. Queries with OLAP functions can apply aggregations on window partitions specified by the "OVER" clauses. There can exist several window partitions with different grouping columns in each OLAP query. That makes this method the only way we can get $F_{indv}$ and $F_{total}$ from the fact table within only one single query. The issue with the OLAP window function is that although the aggregate function is computed with respect to all the rows in each partition,

the result is applied to each row. Therefore, in our case, the result table may have duplicated rows with the same percentage values. The following example shows the SQL query to compute the percentage of the sales amount for each state per quarter, using the raw OLAP window function method:

SELECT $quarter, state$, (CASE WHEN $Y <> 0$ THEN $X/Y$
ELSE NULL END) AS pct
FROM
(SELECT $quarter, state$,
$sum(salesAmt)$ OVER (PARTITION BY $quarter, state$) AS X,
$sum(salesAmt)$ OVER (PARTITION BY $quarter$) AS Y FROM $F$) foo;

To get the results correct, we can get rid of the duplicates with the following two methods:

1. Use the "DISTINCT" keyword.

   SELECT $L_1, \ldots, L_j, R_1, \ldots, R_k$,
   (CASE WHEN $Y <> 0$ THEN $X/Y$
   ELSE NULL END) AS pct
   FROM
   (SELECT **DISTINCT** $L_1, \ldots, L_j, R_1, \ldots, R_k$,
   $sum(A)$ OVER (PARTITION BY $L_1, \ldots, L_j, R_1, \ldots, R_k$) AS X,
   $sum(A)$ OVER (PARTITION BY $L_1, \ldots, L_j$) AS Y FROM $F$) foo;

   The disadvantage with this method is that the use of "DISTINCT" keyword will introduce external sorting in the query execution plan. Such sorting can be expensive if no auxiliary data structures (indexes, projections) are exploited.

2. Use "row_number()"

   *row_number*() is another OLAP function that can assign a sequential number to each row within a window partition (starting at 1 for the first row). When using this method, we assign row identifiers in each partition defined by $L \cup R$, then we just need to select one of such tuples per group to eliminate the duplicates.

   SELECT $L_1, \ldots, L_j, R_1, \ldots, R_k$,

   (CASE WHEN $Y <> 0$ THEN $X/Y$ ELSE NULL END) AS pct

   FROM

   (SELECT $L_1, \ldots, L_j, R_1, \ldots, R_k$,

   $sum(A)$ OVER (PARTITION BY $L_1, \ldots, L_j, R_1, \ldots, R_k$) AS X,

   $sum(A)$ OVER (PARTITION BY $L_1, \ldots, L_j$) AS Y,

   *row_number*() OVER (PARTITION BY $L_1, \ldots, L_j, R_1, \ldots, R_k$)

   AS rnumber FROM $F$) foo WHERE rnumber = 1;

**The GROUP-BY Method**

This method is based on standard aggregations. The two levels of aggregations are pre-computed and stored in temporary tables $F_{total}$ and $F_{indv}$ respectively. The percentage value is evaluated in the last step by joining $F_{indv}$ and $F_{total}$ on the $L$ columns and computing $F_{indv}.A/F_{total}.A$. It is always important to check before computing that $F_{total}.A$ cannot be zero as the denominator.

We explain the evaluation of $F_{total}$ and $F_{indv}$. It is evident that $F_{indv}$ can only be computed from the fact table $F$:

SELECT $L_1$, ... ,$L_j$,$R_1$, ... ,$R_k$, $sum$(A) INTO $F_{indv}$ FROM $F$
GROUP BY $L_1$, ... ,$L_j$,$R_1$, ... ,$R_k$;

$F_{total}$, however, as a more brief summary of the fact table with fewer grouping columns (only columns in $L$) than $F_{indv}$, can be derived either also from the fact table $F$, or directly from $F_{indv}$. Evaluating aggregate functions requires a full scan of the input table. Therefore, the size of the input table has a major impact on the performance. When the size of $F$ is much larger than $F_{indv}$, in which case the cardinality of the grouping columns is relatively small, getting $F_{total}$ from $F_{indv}$ will be much faster than computing it from $F$ because fewer rows are scanned:

SELECT $L_1$,$L_2$, ... ,$L_j$,$sum$(A) INTO $F_{total}$
FROM $F_{indv}$ or $F$
GROUP BY $L_1$,$L_2$, ... ,$L_j$;

The final result can be either inserted into another temporary table or in-place by updating on $F_{indv}$ itself. The in-place update avoids creating the temporary table with the same size as $F_{indv}$. That is helpful for saving disk space.

INSERT INTO $F_{pct}$
SELECT $F_{indv}.L_1$, ... ,$F_{indv}.L_j$,$F_{indv}.R_1$, ... ,$F_{indv}.R_k$,
  (CASE WHEN $F_{total}.A \neq 0$ THEN $F_{indv}.A/F_{total}.A$
  ELSE NULL END) AS pct
FROM $F_{total}$ JOIN $F_{indv}$
ON $F_{total}.L_1 = F_{indv}.L1$, ... ,$F_{total}.L_j = F_{indv}.L_j$;

Compared to the two methods we introduced just now, we argue that our syntax

for percentage is a lot simpler and do not require join semantics.

### Handling Abnormal Data

In order to have a well-defined and robust aggregation for diverse analyses, it is necessary to consider abnormal and missing values. Specifically, it is necessary to define rules to handle nulls in the dimensions, nulls in the measure attribute, zeroes, and negative values in the measure attribute. We introduce the following rules, indicating where our proposed aggregation deviates from standard SQL. These rule can be considered to integrate percentage aggregations into a DBMS or BI tool.

1. Nulls in the values of the dimensions are treated as a single missing value. Therefore, the behavior is same as a GROUP BY query. That is, SQL automatically handles nulls in the dimensions.

2. A null in at least one measure value in some BREAKDOWN subgroup will result in all percentages for all subgroups being null. In other words, we do not treat a missing measure value as a zero like standard SQL; such percentages would be misleading. In other words, it is better to make all percentages in the corresponding cuboid null.

3. In general, percentages adding up to 100% come from positive values. Therefore, negative percentages are allowed with a warning.

4. A total sum equal to zero will result in all percentages for individual subgroups being null (i.e. undefined).

### Classification of Percentage Aggregations

In this section, we answer the questions: are percentage aggregation harder? how hard is it to compute the percentage cube? what is their theoretical connection to existing aggregations? Let us recall the taxonomy of cube aggregations proposed by Gray [19].

- Distributive: sum(), count()
- Algebraic: pct()
- Holistic: top-$k$ percentages

Since percentages are expressed as an equation dividing one sum() by another sum() it is an algebraic aggregation. Notice count(*) can be treated as a sum(1). Recall holistic aggregations are the most challenging ones since they require sorting rows by the aggregated value. In our case, percentages are ranked, and we select the $k$ percentages with the highest ranks in descending order. On the other hand, filtering out low percentages does not change the time complexity of the percentage cube. Each aggregation class results in a different cube generalization, going from easiest to hardest:

- Distributive: standard cube
- Algebraic: percentage cube
- Holistic: pruned percentage cube with top $k$ percentages

In our experimental evaluation, we will quantify the extra effort getting each of these cubes.

### 3.3.2 Evaluating Percentage Cube

Recall that percentage cubes extend standard data cubes. Even though they share a similarity that can give us insight on data in a hierarchical manner, they are quite different. A data cube has a multidimensional structure that summarizes the measures over cube dimensions grouped at all different levels of details. A data cube whose dimensionality is $d$ will have $2^d$ different cuboids. While a percentage cube, in addition to summarizing the measure in cuboids as a data cube does, it categorizes the dimensions in each cuboid into set $L$ and $R$ in all possible ways. Then vertical percentage aggregation is evaluated based on each $L$ and $R$ key sets. The computational complexity of the percentage cube can be summarized in the following properties:

*Property 1:* The number of different grouping column combinations in a cuboid with $g$ grouping columns is $2^g - 1$.

*Property 2:* The total number of all different grouping column combinations in a percentage cube with $d$ dimensions is $\sum_{i=1}^{d} \binom{d}{i}(2^i - 1) = O(2^{2d})$.

So a percentage cube can be much larger than an ordinary data cube in size, and it is a lot more difficult to evaluate. Figure 3.2 shows a specific example when $d = 2$, the data cube will have 4 cuboids while the percentage cube will have in total 5 different grouping column combinations (The last cuboid $\{*, *\}$ will not be included

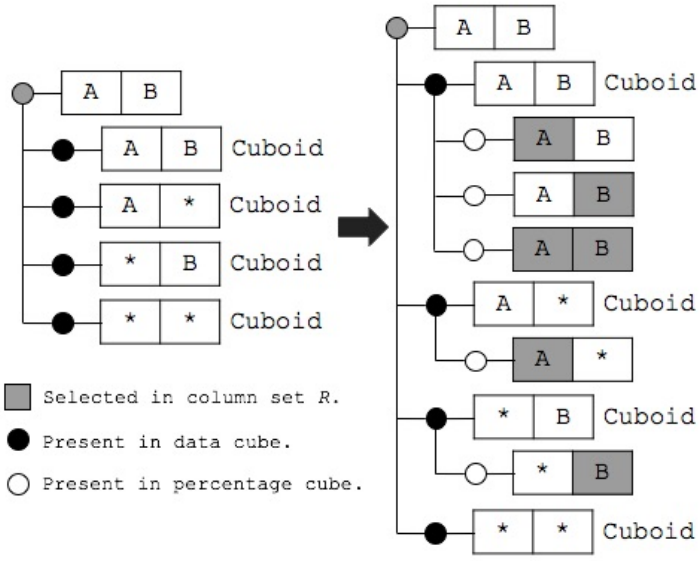



Figure 3.2: Expansion from data cube to percentage cube.

in the percentage cube because the set $R$ cannot be empty). The gap is not so big because the $d$ we show is small due to space limit, since both the number of cuboids in the cube and the number of possible grouping column combinations in one cuboid grow exponentially as $d$ increases, this gap will become surprisingly large when $d$ gets large.

Due to the similarity of the representation of percentage cubes and percentage aggregations, it is not surprising that the problem of building a percentage cube can be broken down to evaluating multiple percentage aggregations and they can share similar SQL syntax.

We describe the algorithm to evaluate a percentage cube using percentage queries in Algorithm 1. The outer loop in Algorithm 1 iterates over each cuboid. Recall that we use $G$ to represent a cuboid's grouping column set (dimensions in that cuboid

that are not "ALL"s). For each cuboid, we will exhaust all the possible ways in the inner loop to allocate the columns in set $G$ to set $L$ and $R$ (columns in set $L$ are the grouping columns for the total level aggregation, and columns in set $R$ are the additional grouping columns to break down the total amounts). For each $L$ and $R$ allocation, we evaluate a percentage aggregation and union all the aggregation results together to be the final percentage cube table.

**Data**: fact table $F$, measure $A$, cube dimension list $M = \{D_1, \ldots, D_d\}$
**Result**: $d$-dimension percentage cube

Result table $RT = \emptyset$ ;
**for** *each* $G \subseteq M, G \neq \emptyset$ **do**
  **for** *each* $L \subset G$ **do**
    $R = G \setminus L$
    $RT_{temp} = pct(A$ TOTAL BY $L$
      BREAKDOWN BY $R)$;
    $RT = RT \cup RT_{temp}$ ;
  **end**
**end**
return $RT$ ;

**Algorithm 1:** Algorithm to evaluate percentage cube.

There is one small difference in the output schema between an individual percentage aggregation and a percentage cube. In a percentage cube, we add two more columns called "total by" and "break down by" to keep track of the total and the individual level setting (See Table 3.3). This is because unlike individual percentage queries having only one total and the individual level setting in the output, the percentage cube explores all the potential combinations. An entry having column $\{A, B\}$ may be "total by" $A$ and "break down by" $B$ or the opposite, or even "break down by" both $A$ and $B$.

We also need to point out that for each cuboid, no matter how the grouping column setting $L$ and $R$ change, the individual level aggregation $F_{indv}$ will stay the same. This is because the $F_{indv}$ is grouped by $L$ and $R$ meanwhile $L \cup R = G$ which will always stay the same in one cuboid. Based on this observation, unlike in vertical percentage aggregations that we compute $F_{indv}$ from $F$ in each $pct()$ call, here we only compute $F_{indv}$ once for every cuboid, then the result will be materialized for the rest of the $L$ and $R$ combinations in the same cuboid to avoid duplicated computations of $F_{indv}$.

## 3.4 Incremental Computation

In a real environment, the data warehouse is periodically refreshed with batches of new records [30], significantly growing in size. So it is highly desirable to reuse previous cube computations to refresh the percentage cube. Assume the data warehouse has a large table $F$ and a smaller table $F_\delta$ with newly inserted records. We assume $|F_\delta| \ll |F|$, typically $\leq 1\%$.

Let $F_{total} = F \cup F_\delta$. A straightforward algorithm is to recompute the percentage cube on $F \cup F_\delta$, which we call a *full recomputation*. On the other hand, we can obtain an equivalent relational algebra equation,

$$\pi_{D_j, sum(A)}(F_{total}) = \pi_{D_j, sum(A)}(F) \cup \pi_{D_j, sum(A)}(F_\delta)$$

.

Since percentage aggregations are algebraic, we can materialize the standard cube

with sum(). The sum() aggregation is distributive which means it can be incrementally computed. Based on these facts, we can state two important properties:

*Property 4:* A Percentage Aggregation can be incrementally computed.

Property 4 enables developing incremental algorithms by materializing total and individual aggregation queries on $F$ and defining materialization aggregations on $F_\delta$.

*Property 5:* The Percentage Cube can be incrementally computed.

Property 5 is a generalization of Property 4. This property allows developing incremental algorithms by materializing the standard cube on $F$, also materializing the standard cube on $F_\delta$ and finally recomputing the percentage cube. However, this may not be optimal when some cuboids do not change. This is precisely the case when some combinations of dimension values do not have newly inserted records. Tracking which percentage cells change is a much harder problem, a research issue for future work.

The experimental section will compare the full recomputation and the incremental computation. Computing one percentage aggregation query incrementally is challenging since a join computation cannot be avoided. Computing the percentage cube incrementally is harder since we cannot avoid the combinatorial explosion of the cube. Experiments will pay attention to time complexity as $d$ grows.

## 3.5 Pruning the Percentage Cube

Not all percentages are interesting or provide valuable information. Based on common analytic goals, we propose three mechanisms to identify interesting percentages:

1. Row count threshold: filtering out groups below a row count threshold, similar to frequent itemsets [33].

2. Percentage threshold: filtering out percentages below a minimum percentage threshold.

3. Top $k$ percentages: Getting the top-k highest (or lowest) percentages.

The percentage computation can exploit a row count (SQL count(*)) threshold, like iceberg queries [21], to significantly reduce the computation effort and avoid getting percentages on tiny groups with very few records behind. Since a high $d$ is sparse, the row count threshold is essential when computing percentages on many dimensions or dimensions where percentages are very small (e.g., percentage of sales by product id, for all products).

The last two mechanisms are alternative filtering mechanisms: The user should use either filter, but not both. The rationale behind such constraint is that output would be incomplete and it would be difficult to understand an overall picture.

### 3.5.1 Row Count Threshold

We revisit the classical optimization to prune the search space of cubes. Since the data cube with $d$ dimensions has $2^d$ cuboids as well as numerous group rows within each cuboid, it is a computationally hard problem. Moreover, as explained in Section 3.3.2, a percentage cube is much larger than a standard data cube because, in addition to the $2^d$ cuboids, there are a lot more potential total group columns combinations in each cuboid. Therefore, computing percentage cubes is significantly more demanding than getting cubes.

Taking a closer look, not all percentage groups (i.e. groups formed by the total level aggregation) are valuable. Although in some groups the user can discover entries with remarkable percentage values, the group itself may be small in row count. Discoveries based on such groups do not have enough "statistical evidence", like support in frequent itemsets [2]. It is expected there will be many of such small groups in the percentage cube, especially when $d$ is large. If we can avoid computing those groups, the overall evaluation time can be correspondingly reduced and output size reduced.

On the data cube side, a similar problem of eliminating GROUP-BY partitions with an aggregate value (e.g., count) below some support thresholds can be solved by computing Iceberg cubes [21]. For Iceberg queries, it is justified that a frequency threshold is required, and it is even more necessary in a percentage cube. In an analog manner, we introduce a threshold to prune groups under a specified size. We call this threshold *group threshold*, represented by $\phi$. In percentage cubes, all the groups are generated by the total level aggregation ($F_{total}$). Therefore, unlike in

Iceberg cubes, we prune the partitions, in percentage cubes we prune groups under a specified size $\phi$, that is, to filter the aggregated $count()$ of groups formed by all the possible $L$ sets through this frequency threshold.

Previous studies have developed two major approaches to compute Iceberg cubes, top-down [55] and bottom-up [7]. The most important difference between those two methods is that the bottom-up algorithm can take advantage of Apriori pruning [3]. Such pruning strategy can also apply on percentage cubes. In this section, we introduce two pruning strategies: direct pruning and bottom-up cascaded pruning.

**Direct pruning based on row count**

Direct pruning further results in Algorithm 1. This algorithm validates the threshold on all possible grouping column combinations directly without sharing pruning results between computations at difference grouping levels. In order to let the computation of $F_{total}$ continue to reuse the result of $F_{indv}$ table that comes from the coarser level of details, we also put $count(1)$ in $F_{indv}$ results. When computing $F_{total}$ from $F$, the group frequency is evaluated by $count()$. However, when utilizing $F_{indv}$ to get $F_{total}$, the group frequency is evaluated by summing up the counts in $F_{indv}$. The threshold is enforced in $F_{total}$ query by specifying the threshold in the "HAVING" clause.

SELECT $L_1$,$L_2$, ... ,$L_j$,$sum$(A),$count$(1) AS count
INTO $F_{total}$
FROM $F$
GROUP BY $L_1$,$L_2$, ... ,$L_j$

HAVING count(1)> $\phi$;

SELECT $L_1$,$L_2$, ... ,$L_j$,$sum$(A),$sum$(count) AS count

INTO $F_{total}$

FROM $F_{indv}$

GROUP BY $L_1$,$L_2$, ... ,$L_j$

HAVING $sum$(count)> $\phi$;

**Cascaded pruning**

In order to take advantage of previous pruning results, we propose a new algorithm that iterates over all cube groups going from coarser aggregation levels (few grouping columns) to finer (more grouping columns) levels. We show the cascaded pruning algorithm to compute the percentage cube with a frequency threshold $\phi$ in Algorithm 2. If the *count*() of any group fails to meet the minimum threshold, the group can be pruned and we avoid going deeper down checking other groups that have more dimensions included in the path along the dimension lattice. On the other hand, qualified groups can be materialized in temporary tables with their grouping column values, the *count*(), and the aggregated measures so that this materialized table can later be used for percentage computations, or the prune the lattice search space with more detailed grouping columns.

We contrast our cascaded algorithm, shown in Algorithm 2, with the percentage cube algorithm shown in Algorithm 1. We first determine the cuboid to get $G$, the cuboid's dimension list in the outermost loop. Then we get each $L$ and $R$ sets from $G$ in the inner loop. Keep in mind that the $F_{indv}$ is always grouped by $G$.

**Data**: Fact table $F$, measure $A$, cube dimension list $G = \{D_1, \ldots, D_p\}$, group threshold $\phi$

**Result**: $d$-dimensional percentage cube

Result table $RT = \emptyset$ ;
$F_{total_0} = \sigma_{count>\phi}(\pi_{count(1),sum(A)}(F))$ ;
**for** *each* $L \subset G, L \neq \emptyset$ **do**
  $i = getRepresentationCode(L)$ ;
  $p = getParentCode(i)$ ;
  **if** $p = 0$ **then**
    $F_{total_i} = \sigma_{count>\phi}(\pi_{L,count(1),sum(A)}(F))$ ;
  **else**
    **if** $F_{total_p}$ *does not exist* **then**
      continue next $L$;
    **else**
      **if** $F_{indv_i} exists$ **then**
        $F_{total_i} = \sigma_{sum(count)>\phi}(\pi_{L,sum(count),sum(A)}($
          $F_{total_p} \ltimes_{F_{total_p}.L=F_{indv_i}.L} F_{indv_i}))$ ;
      **else**
        $F_{total_i} = \sigma_{count(1)>\phi}(\pi_{L,count(1),sum(A)}($
          $F_{total_p} \ltimes_{F_{total_p}.L=F.L} F))$ ;
      **end**
    **end**
  **end**
  **if** $|F_{total_i}| \neq 0$ **then**
    Materialize $F_{total_i}$;
  **end**
  **for** *each* $R \subseteq (G \setminus L)$ **do**
    $S = L \cup R$; $sCode = getRepresentationCode(S)$ ;
    **if** $F_{indv_{sCode}}$ *does not exist* **then**
      Materialize $F_{indv_{sCode}} = \pi_{S,sum(A)}(F)$ ;
    **end**
    $RT_{temp} = \pi_{S,F_{indv_{sCode}}.A/F_{total_i}.A}($
      $F_{total_i} \bowtie_{F_{total_i}.L=F_{indv_{sCode}}.L} F_{indv_{sCode}})$ ;
    $RT = RT \cup RT_{temp}$ ;
  **end**
**end**
return $RT$ ;

**Algorithm 2:** Cascaded bottom up algorithm to get the percentage cube.

Therefore, in this computational order, every $F_{indv}$ is computed, intensively utilized and discarded. The cascaded pruning algorithm, however, generates the aggregation result level by level. The complication here is that a grouping column set on a certain level may appear in multiple cuboids. For example, for a cube with dimensions $\{A, B, C, D\}$, a grouping column set $\{A, B\}$ is used in cuboid $\{A, B, C\}$, $\{A, B, D\}$, and $\{A, B, C, D\}$. Therefore, the materialized result table for each grouping column set can be reused in multiple cuboids in a more scattered manner. It is important we keep the results for future usage after a group is first computed. A side effect is that when computing some $F_{total}$, it is not always true that $F_{total}$ can be computed from $F_{indv}$ because the $F_{indv}$ it needs may not have been computed yet.

In order to label the $L$ set on each materialized table, we assign each cube dimension with an integer identifier according to the position it is standing in the dimension list. The identifier for the $i$-th dimension will be $2^{i-1}$. With the dimension identifier, any $L$ set or $R$ set or cuboid dimension list can be represented by a representation code that comes from the bitwise $OR$ operation on all the dimensions' identifier in the set, and the code for its parent set can be evaluated by eliminating the highest non-zero bit. All materialized $F_{indv}$ and filtered $F_{total}$ will be named as $F_{indv_i}$ and $F_{total_i}$ where $i$ stands for the dimension representation code. Figure 3.3 shows the relation between the representation code and the dimension set it represents. Also by eliminating the highest non-zero bit, it can be linked with its parent dimension set.

To close this section, we emphasize that by applying pruning mechanisms time complexity is significantly reduced from $O(2^{2d})$. The specific time $O()$ bound will depend on dimensions probabilistic properties.

| Position | 1 | 2 | 3 | p |
|---|---|---|---|---|
| Identifier | 1 | 2 | 4 | $2^{p-1}$ |
| Dimension | $D_1$ | $D_2$ | $D_3$ | $D_p$ |

| Code | Binary | Dimensions | | | Parent |
|---|---|---|---|---|---|
| 0 | $(000)_2$ | $\boldsymbol{D_1}$ | $\boldsymbol{D_2}$ | $\boldsymbol{D_3}$ | None |
| 1 | $(001)_2$ | $\boldsymbol{D_1}$ | $\boldsymbol{D_2}$ | $\boldsymbol{D_3}$ | $(000)_2$ |
| 2 | $(010)_2$ | $\boldsymbol{D_1}$ | $\boldsymbol{D_2}$ | $\boldsymbol{D_3}$ | $(000)_2$ |
| 3 | $(011)_2$ | $\boldsymbol{D_1}$ | $\boldsymbol{D_2}$ | $\boldsymbol{D_3}$ | $(001)_2$ |
| 4 | $(100)_2$ | $\boldsymbol{D_1}$ | $\boldsymbol{D_2}$ | $\boldsymbol{D_3}$ | $(000)_2$ |
| 5 | $(101)_2$ | $\boldsymbol{D_1}$ | $\boldsymbol{D_2}$ | $\boldsymbol{D_3}$ | $(001)_2$ |
| 6 | $(110)_2$ | $\boldsymbol{D_1}$ | $\boldsymbol{D_2}$ | $\boldsymbol{D_3}$ | $(010)_2$ |
| 7 | $(111)_2$ | $\boldsymbol{D_1}$ | $\boldsymbol{D_2}$ | $\boldsymbol{D_3}$ | $(011)_2$ |

Means being selected in the code.

Figure 3.3: Dimensions as binary codes.

### 3.5.2 Percentage Threshold

Getting rid of almost empty cells in the cube helps a lot, but it is not enough in a practical scenario. The user may want to further filter out percentages below a certain percentage threshold. The main challenge is that it is not possible to prune the dimension lattice like classical cubes, because large percentages may be "hidden" behind groups with small percentages. Therefore, it is impossible to use traditional lattice pruning strategies like those used in frequent itemsets or iceberg queries [20]. That is, percentages are not anti-monotonic. In short, a percentage cube represents a significantly harder problem than standard cubes. We summarize this challenge as the following result:

*Property 3:* A percentage aggregation on a set of cube dimensions is not anti-monotonic. Therefore, it is impossible to develop bottom-up minimum percentage discovery algorithms based on percentages.

Time complexity $O(2^{2d})$ does not change from building the percentage cube since filtering out small percentages can be done with a sequential algorithm on each cuboid after the percentage cube is materialized.

### 3.5.3 Top-$k$ percentages

Filtering out low percentages may be difficult if the dimensions have high cardinality. Instead, the user may decide to look at the highest percentages. The main idea is to rank percentages from highest to lowest and then select the $k$ highest ones, where $k \geq 1$. This filtering procedure needs to be done on each cuboid. Query evaluation will require sorting percentages within each cuboid, which is an expensive computation in a big cube. An important observation is that selecting the top $k$ percentages should be done after the cube is materialized because $k$ may increase. That is, it would be a bad idea to materialize a pruned percentage cube.

Time complexity is significantly higher than computing a percentage cube and filtering out low percentages. For the percentage cube, this additional computation will result in $O(2^{2d})$ sorts, where each sort has time complexity $O(mlog_2(m))$ assuming an average $m$ rows per cuboid. That is, time becomes $O(2^{2d}mlog_2(m))$. Notice it is difficult to derive worst-case bounds because the number of rows in a cuboid depends on dimension cardinality, which can vary widely.

## 3.6 Experiments

### 3.6.1 Experimental Setup

#### Hardware and Software

We conducted our experiments on a 2-node cluster of Intel dual-core workstations running at a 2.13 GHz clock rate. Each node has 2GB main memory and 160GB disk storage with SATA II interface. The nodes are running Linux CentOS release 5.10 and are connected by a 1 Gbps switched Ethernet network.

The HP Vertica DBMS was installed under a parallel 2-node configuration for obtaining the query execution times shown in this section's tables. Our default configuration was 2 nodes in a local cluster, but some experiments used 1 node on the Amazon cloud.

The reported times in the table are rounded down to integers and are averaged on seven identical runs after removing the best and worst total elapsed time.

#### Data Set

The data set we used to evaluate the aggregation queries with different optimization strategies is the synthetic data sets generated by the TPC-H data generator. In most cases, we use the fact table transactionLine as input and the column "quantity" as the measure. Table 3.5 shows the specific columns from the TPC-H fact table that we used as left key and right key to evaluate percentage queries. $|L_1|$ and $|R_1|$ are

the cardinalities of $L_1$ and $R_1$ respectively. Table 3.6 shows the candidate dimensions and their cardinalities we used to evaluate percentage cubes.

Table 3.5: Summary of grouping columns for individual percentage queries transactionLine ($N$=6M).

| $L_1$ | $R_1$ | $\|L_1\|$ | $\|R_1\|$ |
|---|---|---|---|
| brand | quarter | 25 | 4 |
| brand | dweek | 25 | 7 |
| brand | month | 25 | 12 |
| clerkKey | dweek | 1K | 7 |
| clerkKey | month | 1K | 12 |
| clerkKey | brand | 1K | 25 |
| custKey | dweek | 200K | 7 |
| custKey | month | 200K | 12 |
| custKey | brand | 200K | 25 |

Table 3.7 shows the dimensions we chose to generate the Percentage Cube at various cube dimensionality. In this dissertation, we vary the dimensionality of the cube $d$ from 2 to 6. Very large $d$ does not make much sense for our computation because the groups will be too small.

Table 3.6: Candidate cube dimensions' cardinalities.

| Dimension | Cardinality |
|---|---|
| manufacturer | 5 |
| year | 7 |
| ship mode | 7 |
| month | 12 |
| nation | 25 |
| brand | 25 |

Table 3.7: Selected cube dimensions at various $d$.

| $d$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ |
|---|---|---|---|---|---|---|
| 2 | nation | brand | | | | |
| 3 | nation | brand | year | | | |
| 4 | nation | brand | year | month | | |
| 5 | nation | brand | year | month | ship mode | |
| 6 | nation | brand | year | month | ship mode | manufacturer |

**System Programming**

All the algorithms we presented in this paper can be implemented by assembling queries that are already supported in DBMSs. That is, our solution is portable, not tied to any platform or specific database system. To support the percentage aggregation and the percentage cube SQL syntax that we proposed, we developed a Java program that parses the percentage cube queries and converts them into a sequence of SQL queries for each method (i.e. OLAP window function or GROUP-BY queries). The Java program connects to the DBMS via JDBC, sends SQL queries, finally downloading the final result cube table. There are two reasons in favor of JDBC: it is a standard protocol, it works on Java guaranteeing portability across diverse computers and operating systems. The main cons with JDBC are the slow speed to export large tables and the overhead to submit multiple SQL statements. We avoid the first limitation by computing the cube inside the DBMS (i.e. we export one small table at the end). Regarding the overhead to submit multiple SQL statements, we minimize it, by sending several statements together. At the end, the percentage cube can be exported to external programs, like Excel or R, to visualize (as explained in Section 3.1) or further explore results. An important acceleration

could be obtained with high $d$ cubes by integrating our algorithms inside the DBMS, but such approach requires availability of source code and it represents a significant programming effort. Therefore, we believe that Java/JDBC are a reasonable compromise.

### 3.6.2 Comparing Percentage Aggregation Methods

As explained in Section 3.3.2. the computation of the percentage cube is based on assembling multiple percentage aggregation queries, in a lattice traversal algorithm. With this motivation in mind, we first compare the two methods to implement $pct()$ as discussed in Section 3.3.1, i.e. GROUP-BY and OLAP. We performed this comparison on a replicated fact table $F$ whose scale factor = 4. In this comparison, we include optimization options for both methods. For the GROUP-BY method, we evaluate $F_{total}$ from $F$ and $F_{indv}$ separately, while for the OLAP window function method we eliminate duplicates by using "DISTINCT" and $row\_number()$ separately.

We also compared the time to evaluate the percentage cube when certain portion of the measure data is null. In Table 3.11, we showed the comparison result. The comparison does not show an obvious difference in evaluation performance when we vary the percentage of null measure values in the fact table.

Table 3.8 shows the result of the comparison. For each grouping column combination in each row, we highlight the fastest configuration with bold font. Generally speaking, evaluation by the GROUP-BY method was much faster than by the OLAP window function method. For the OLAP method, using $row\_number()$ instead of

Table 3.8: Percentage Aggregation: GROUP-BY vs. OLAP. Scale factor=4,N=24M (times in secs).

| | | GROUP-BY | | OLAP | |
|---|---|---|---|---|---|
| $\|L_1\|$ | $\|R_1\|$ | $F_{total}$ from $F$ | $F_{total}$ from $F_{indv}$ | DISTINCT | row_number() |
| 25 | 4 | 8 | **5** | 52 | 29 |
| | 7 | 7 | **5** | 50 | 28 |
| | 12 | 7 | **5** | 50 | 28 |
| 1K | 7 | 10 | **6** | 48 | 22 |
| | 12 | 10 | **7** | 49 | 22 |
| | 25 | 19 | **16** | 63 | 30 |
| 200K | 7 | 35 | **10** | 43 | 23 |
| | 12 | 13 | **11** | 44 | 27 |
| | 25 | 56 | **51** | 62 | 54 |

"DISTINCT" keyword may accelerate the evaluation speed by about 2 times. But it is still obviously slower than the GROUP-BY method. For the GROUP-BY method, we can see the cardinality of the right key $|R1|$, has a direct impact on the performance of two strategies to generate $F_{total}$. When $|R1|$ is relatively small, say $|R1| = 7$, for all different $|L1|$ we have tested, generating $F_{total}$ from $F_{indv}$ is about 2-3 times faster than from $F$. However as $|R1|$ gets larger, say $|R1| = 25$, it matters less where $F_{total}$ is generated from.

Table 3.9: Cube Generation: GROUP-BY vs. OLAP. Scale factor=1, N=6M (times in secs).

| $d$ | GROUP-BY | OLAP |
|---|---|---|
| 2 | 4 | 41 |
| 3 | 10 | 155 |
| 4 | 30 | 545 |
| 5 | 148 | 2402 |
| 6 | 1147 | 9174 |

Table 3.10: Direct pruning vs. Cascaded pruning (times in secs).

| | threshold | Direct Pruning | | Cascaded Pruning | |
|---|---|---|---|---|---|
| $d$ | (% of N ) | SF=1 | SF=8 | SF=1 | SF=8 |
| 5 | 10% | 91 | 690 | 70 | 674 |
| | 8% | 91 | 693 | 72 | 675 |
| | 6% | 90 | 692 | 74 | 676 |
| | 0 | 124 | 728 | 130 | 729 |
| 6 | 10% | 268 | 1767 | 177 | 1680 |
| | 8% | 269 | 1769 | 182 | 1687 |
| | 6% | 272 | 1774 | 186 | 1687 |
| | 0 | 437 | 1981 | 436 | 1930 |

Table 3.11: Time to evaluate the percentage cube with nulls in the measure. n = 20M, d = 4 (times in secs).

| null% | time |
|---|---|
| 0 | 3.72 |
| 5 | 3.58 |
| 10 | 3.55 |
| 15 | 3.73 |
| 20 | 3.62 |

### 3.6.3 Performance of the incremental computation

We start by studying incremental computation for a single percentage query, shown in Table 3.12. We vary the cardinality of the first dimension of the fact table, which is the number of unique groups the first dimension generates. As we can see from the table, the incremental computation is always faster than recomputing the full percentage cube. Also, the performance gain from the incremental computation grows as the cardinality increases (more groups are generated).

Table 3.12: Incremental computation, ONE query ($n$ =20M, $d = 4$, $\delta = 1\%$, times in secs).

| Cardinality | Original | Full Recomp. | Incremental | Fraction incr/full |
|---|---|---|---|---|
| 1 | 0.63 | 0.63 | 0.26 | 41% |
| 10 | 0.67 | 0.70 | 0.27 | 39% |
| 100 | 26.91 | 25.65 | 5.97 | 23% |

Table 3.13: Incremental computation ($n$ =20M, $\delta = 1\%$, times in secs).

| $d$ | Original | Full Recomp. | Incremental | Fraction incr/full |
|---|---|---|---|---|
| 1 | 0.3 | 0.4 | 0.1 | 25% |
| 2 | 0.5 | 0.6 | 0.2 | 33% |
| 3 | 0.9 | 1.0 | 0.6 | 60% |
| 4 | 2.6 | 2.7 | 2.6 | 96% |
| 5 | 26.4 | 30.5 | 29.8 | 98% |

Table 3.13 compares the incremental percentage cube computation with a full recomputation when inserting 1% records (200k). The goal is to understand if there are time savings and if time can be bounded by $|F_\delta|$. To our surprise, the incremental computation is almost as slow as a full recomputation as $d$ grows. That is, extra time is not proportional to $|F_\delta|$. But it does work at low $d$. The fraction trend of time between incremental and full recomputation indicates that an incremental computation time approaches a full recomputation time. In fact, at $d = 5$, the times are almost the same. After profiling the query plan for bottlenecks, analyzing each query and trying several $n$ sizes (smaller $n$ sizes omitted because the trend was fuzzy) we concluded that $d$ is a much more important performance factor than $n$ because two percentage cubes are computed: one on $F$ and one on $F_\delta$. Coming up with a more efficient incremental algorithm, compatible with SQL, is an item for future

work. As can be seen, at low $d$ there is some gain, but at high $d$ time almost doubles despite the fact that top $k$ percentages are discovered on the materialized percentage cube.

### 3.6.4 Comparing Percentage Cube Materialization Methods

We now assemble the $pct()$ queries together using the Algorithm 1 to materialize the entire percentage cube. Since evaluating a percentage cube is much more demanding than evaluating individual percentage queries, when comparing those two methods in cube generation, we choose the best method based on the experimental results presented in Section 3.6.2. That is, for GROUP-BY method, we generate $F_{total}$ from $F_{indv}$, and for OLAP window function method, we use $row\_number()$ OLAP function to eliminate duplicated rows. Table 3.9 shows the result of this comparison. The result shows that our GROUP-BY method is about 10 times faster than the OLAP window function method for all $d$'s. When $d$ gets larger, it will take the OLAP window function method hours to finish. The two methods show an enormous difference in their performances because our GROUP-BY method takes advantage of the materialized $F_{indv}$. So we can not only avoid duplicated computation of $F_{indv}$ for each group in the same cuboid but also benefit from the generation of $F_{total}$ for different $L$ set.

### 3.6.5 Comparing Cube Pruning Mechanisms

Cube cells with very few rows do not provide reliable knowledge because an interesting finding may be a coincidence, without enough evidence. Therefore, in

general, the user will apply a minimum row count threshold in order to avoid almost empty cells. Since the percentage cube is big, it is necessary to apply a further filter on the percentages. We propose to either: get the top $k$ percentages, where $k \geq 1$ or get all percentages above a minimum threshold, $\phi$. Applying both percentage pruning filters would result in incomplete and confusing output.

#### 3.6.5.1 Row count threshold

Even though the GROUP-BY method exhibits acceptable performance when evaluating the percentage cube, it is still not sufficient especially for fact tables with large $d$ and $N$. As discussed in Section 3.5.1, we want to further optimize this evaluation process by introducing *group frequency threshold* to prune the groups with low *count*(). In this section, we compare the cube generation with various group thresholds using direct pruning on Algorithm 1 and cascaded pruning in Algorithm 2. We choose the values of the threshold as certain percentages of total row number of the fact table $N$. Here we choose the *count*() threshold as 10% of $N$, 8% of $N$, and 6% of $N$ as well as no threshold applied. The result is shown in Table 3.10. We first look at evaluation times for each algorithm separately. It shows although the time increases when we decrease the threshold, the difference is not so big. This is because the data distribution of the data set we used is almost uniform. A more skewed distribution of values will result in a more obvious increase in evaluation times for the decreasing thresholds. But on the other hand, we can see from the result that the evaluation time increases greatly for runs with no threshold. Therefore it proves to us that it is necessary to prune the groups with small sizes because not only users have little interest in them but also by pruning them, the evaluation

time can be shortened. Then we compare the evaluation time between algorithms. When $d$ continues to grow, bottom-up algorithm shows a much better performance for cases when thresholds are applied. But two algorithms shows almost no difference when threshold $\phi = 0$. Direct pruning can run without the support of the Java program. However, for cascaded pruning, a Java program is needed to participate in the processing throughout the evaluation. So, the cost of communication via JDBC and the running the Java program can compromise the true performance of the algorithm. This cost can be diminished by integrating the algorithm into the DBMS as a built-in functionality.

#### 3.6.5.2 Getting Top $k$ percentages

Table 3.14: Top $k$ percentages (times in secs).

| $d$ | pct cube | top $k$ | Total | top $k$ |
|---|---|---|---|---|
| 1 | 0.3 | 0.1 | 0.4 | 25% |
| 2 | 0.5 | 0.1 | 0.6 | 17% |
| 3 | 0.9 | 0.4 | 1.3 | 31% |
| 4 | 2.6 | 1.9 | 4.5 | 42% |
| 5 | 26.4 | 62.9 | 89.3 | 76% |

As explained before, identifying the highest percentages is a demanding computation. Table 3.14 analyzes evaluation times as $d$ grows on a large table (relative to the system). The fraction of the time taken by the top $k$ computation grows as $d$ grows and the trend indicates it approaches 1. These times highlight the combinatorial time complexity and the high cost of one sort per cuboid.

#### 3.6.5.3 Selecting percentages above a minimum percentage threshold

In Table 3.15, we compare time to prune percentages on fact tables with varying $d$. We generated the percentage cube on fact tables that have 20M rows and increasing number of dimensions. From the result, we can see that as $d$ grows, the time it takes to compute the top $k$ percentages become more and more significant and its growth rate is so much larger than the time for computing percentages with row count thresholds given the exponential number of sort operations, one per cuboid.

Table 3.15: Pruning method: row count threshold vs. top $k$, n = 20M (times in secs).

| $d$ | threshold | top-k=2 |
|---|---|---|
| 1 | 0.42 | 0.44 |
| 2 | 0.58 | 0.69 |
| 3 | 1.23 | 1.65 |
| 4 | 4.84 | 7.89 |
| 5 | 42.94 | 126.24 |

# Chapter 4

# Matrix Aggregation:

# The Gamma operator

## 4.1 Definition

We start by defining the input matrix, stressing it is a set of $n$ *column* vectors. All models take a $d \times n$ matrix $X$ as input. Let $X = \{x_1, ..., x_n\}$ be the input data set with $n$ points, where each point $x_i$ is a vector in $\mathbf{R}^d$. Intuitively, $X$ is a wide rectangular matrix. In the case of LR and VS, $X$ is augmented with a $(d+1)$th dimension containing an output variable $Y$, making $X$ a $(d+1) \times n$ matrix. We use $i = 1 \ldots n$ and $j = 1 \ldots d$ as matrix subscripts. We emphasize that for convenience in mathematical notation we use *column* vectors and *column*-oriented matrices.

We use $\Theta$, a set of matrices and associated statistics, to refer to a statistical model in a generic manner. Thus $\Theta$ represents the PCA, LR, and VS models defined in Chapter 4.4. Moreover, $\Theta$ could represent a clustering or classification model. PCA represents an unsupervised (descriptive) model to reduce dimensionality. Linear regression is a fundamental supervised model, whose solution helps understanding and building other linear models. Variable selection (VS), a supervised model, is among the computationally most difficult problems in statistics and machine learning, in which Markov Chain Monte Carlo (MCMC) methods [22, 17] are a promising solution.

## 4.2 Gamma Summarization Matrix

In this section we introduce the $\Gamma$ summarization matrix. This matrix contains several vectors and submatrices that represent important sums derived from the data set. In Chapter 4.4, we show such sums help derive fundamental statistical properties of the data set, for each dimension (variable) and for each variable pair.

We first review sufficient statistics matrices [22, 35], which are integrated and generalized into a single matrix:

$$n = |X| \tag{4.1}$$

$$L = \sum_{i=1}^{n} x_i \tag{4.2}$$

$$Q = XX^T = \sum_{i=1}^{n} x_i \cdot x_i^T \tag{4.3}$$

Intuitively, $n$ counts points, $L$ is a linear sum of $x_i$ and $Q$ is a quadratic sum of $x_i$, where $x_i$ is multiplied by itself (i.e. squared) with a vector outer product. As explained in Section 4.4.2 , the linear regression model uses an *augmented* matrix, represented by $\mathbf{X}$. We introduce a more general augmented matrix $\mathbf{Z}$, by appending an additional $(d+1)^{th}$ row to $\mathbf{X}$, which contains the vector Y. Since $X$ is $d \times n$, $\mathbf{Z}$ has $(d+2)$ rows and $n$ columns, where row $[0]$ are 1s and row $[d+1]$ is $Y$.

Matrix $\Gamma$ contains a comprehensive, accurate, and sufficient summary of $X$ to efficiently compute all models previously defined. Below, we show $\Gamma$ in two equivalent forms: (1) as vector-matrix and matrix-matrix multiplications; (2) as sums of vector outer products. Notice $\mathbf{1}$ is a column-vector of $n$ 1s, which allows expressing a

sum as a matrix product. Such equivalence has important performance implications depending on how the matrix is processed.

$$\Gamma = \begin{bmatrix} n & L^T & \mathbf{1}^T \cdot Y^T \\ L & Q & XY^T \\ Y \cdot \mathbf{1} & YX^T & YY^T \end{bmatrix} = \begin{bmatrix} n & \sum x_i^T & \sum y_i \\ \sum x_i & \sum x_i x_i^T & \sum x_i y_i \\ \sum y_i & \sum y_i x_i^T & \sum y_i^2 \end{bmatrix}$$

The fundamental property of $\Gamma$ is that it can be computed by a single matrix multiplication using $\mathbf{Z}$. Therefore, we study how to compute the matrix product below, related to, but not the same as the Gram matrix $\mathbf{Z}^T\mathbf{Z}$ [13]:

$$\Gamma = \mathbf{Z}\mathbf{Z}^T \tag{4.4}$$

Matrix $\Gamma$ is fundamentally the result of "squaring" matrix $\mathbf{Z}$. $\Gamma$ is comparatively much smaller than $X$ for large $n$, symmetric and computable via vector outer products. Such facts are summarized in the following properties:

*Property 1 (matrix product versus vector outer product):* $\Gamma$ can be equivalently computed as follows:

$$\Gamma = \mathbf{Z}\mathbf{Z}^T = \sum_{i=1}^{n} z_i \cdot z_i^T. \tag{4.5}$$

*Property 2 (small size):* Given a large data set $X$ where $d \ll n$, matrix $\Gamma$ size is $O(d^2) \ll O(dn)$ as $n \to \infty$.

*Property 3 (symmetry):* $\Gamma = \Gamma^T$, which can be condensed into a triangular matrix with $d + (d+2)^2/2 \approx d^2/2$ entries, saving 1/2 of CPU operations.

*Property 4 (first and second moment):* Matrix $\Gamma$ summarizes $X$ to compute the first and second moment of several multivariate probabilistic distributions.

Property 1 is fundamental to derive a highly efficient summarization operator. Property 1 gives two equivalent expressions to obtain $\Gamma$: one as a matrix multiplication and the second one as a sum of vector *outer products*. We make the following fundamental observation. Even though it is tempting to evaluate $\mathbf{Z}\mathbf{Z}^T$ as a matrix multiplication it is a bad idea: we need to compute and materialize $\mathbf{Z}^T$ first, a costly transposition, storing it, and then perform the actual matrix multiplication. Instead, we defend the idea of evaluating the summation $\sum_{i=1}^{n} z_i \cdot z_i^T$. We will experimentally prove this hypothesis. Assuming $z_i$ fits in main memory, then it is reasonable to assume the vector outer product also fits in main memory. Therefore, it makes more sense to evaluate such sum in parallel. Given the prominent importance of $\Gamma$, we believe our operator should be available on any big data analytic system. We call our operator $\Gamma(X)$ (called Gamma($X$) in a C++ function).

Property 2 means that if $\Gamma$ can fit in main memory, it is feasible to maintain a summary of $X$ in main memory. However, we emphasize that we cannot assume $X$ can fit in main memory. Therefore, the challenge is to efficiently compute $\Gamma$, minimizing the number of times $X$ must be read from secondary storage.

Property 3 states it is possible to save one half of the CPU work. Notice storage space in triangular form is still $O(d^2)$, but it requires a nested indexing mechanism. In

order to allocate a square array, which enables faster address computation and block-based access, we prefer not to use compact storage. Notice Property 2 implies $\mathbf{Z}\mathbf{Z}^T = (\mathbf{Z}\mathbf{Z}^T)^T = (\mathbf{Z}^T)^T\mathbf{Z}^T = \Gamma$, highlighting the matrix product is non-commutative. In other words, $\Gamma \neq \mathbf{Z}^T\mathbf{Z}$. Table 4.1 provides a summary of all matrices, including the input data set, the output variable, model $\Theta$, and $\Gamma$.

We explain Property 4 in more technical detail. $\Gamma$ is a fundamental summarization matrix because it can help computing the first and second expected moments of many probabilistic distributions: the first moment $E[x] = \mu$, to get the global mean (average) and the second moment $E[(x-\mu)(x-\mu)^T] = V$ for the covariance matrix, which measures mean squared error (MSE) per dimension and helps to understand each variable pair behavior. Mean squared error is the most commonly used error measure in statistics and machine learning because it is mathematically easier to optimize [22]. $\Gamma$ can be used to *directly* derive the global mean and the covariance matrix: That is, $\Gamma$ helps to understand each individual variable and each pair of variables. The following models represent a generalization of these basic statistical matrices. Higher order variable interactions can be analyzed adding dimensions to $X$ for each variable pair.

## 4.3 Two-Phase Analytic Algorithm

Recall from Section 4.1, $\Theta$ represents a statistical model. Therefore, based on $\Gamma$ a fast algorithm to compute $\Theta$ in two phases is the following:

1. Compute $\Gamma$.

Table 4.1: Summary of matrices: summarization and models.

| Matrix | Size | Description |
|---|---|---|
| $X$ | $d \times n$ | Data set: dimensions/variables |
| $\mathbf{X}$ | $(d+1) \times n$ | Augmented matrix with 1s |
| $\mathbf{Z}$ | $(d+2) \times n$ | Augmented $X$ matrix with 1s and $Y$ |
| $\Gamma$ | $(d+2) \times (d+2)$ | Summarization matrix |
| $U$ | $d \times d$ | Principal components |
| $V$ | $d \times d$ | Squared eigenvalues; diagonal |
| $Y$ | $1 \times n$ | Dependent variable |
| $\beta$ | $(d+1) \times 1$ | Regression coefficients, $Y$ intercept |
| $\gamma$ | $d \times 1$ | Selected variables; binary vector |

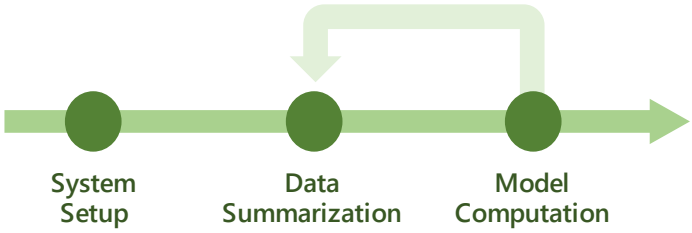



Figure 4.1: The workflow for a system using $\Gamma$ for model computation

2. Iterate exploiting $\Gamma$ in intermediate matrix computations.

These two phases can be repeated as many times as needed for iterative algorithms in a system using this paradigm, the workflow is shown in Figure 4.1. This dissertation focuses on optimizing Phase 1, which is a well-defined problem for any input data set $X$. Phase 2 requires incorporating $\Gamma$ in the steps of numerical and statistical methods. By exploiting $\Gamma$, it becomes possible to reduce the number of times $X$ is read and to reduce CPU computations in iterative methods. Identifying in which models intermediate matrix computations accept $\Gamma$ is a deep research topic. We point out that this work builds on top of previous research and we make clear for which kind of models our optimizations apply. That is, our choice of methods to

solve PCA, LR, and VS has been carefully chosen because they can exploit $\Gamma$ in the most demanding matrix equations.

## 4.4 Models Exploiting the Gamma Matrix

Our summarization matrix benefits a large family of statistical models. In this dissertation, we focus on PCA, LR, and VS, where one summarization matrix is sufficient despite their iterative methods are different. Our summarization matrix can be generalized to compute the Naive Bayes classifier, K-means/EM clustering, logistic regression, and Linear Discriminant Analysis, among others. But these latter models require more than one summarization matrix (e.g., $k$ summarization matrices for K-means or EM). This introduces different assumptions and other iterative methods. Therefore, such models are a research issue for future work.

For the statistical models, we focus on we use a single summarization matrix $\Gamma$ that is non-diagonal. That is, we do not assume dimensions are independent. Therefore, $\Gamma$ is a full matrix with $O(d^2)$ non-zero entries. From an algorithmic perspective, we have managed to use $\Gamma$ as a proxy of $X$ in every step. Therefore, the model $\Theta$ is computed in one (parallel) pass over $X$, as explained in the next subsections.

There are two basic directions to generalize $\Gamma$: (1) computing multiple $\Gamma$ matrices, where each one corresponds to some subset of $X$, obtained by partitioning $X$ based on the values of some attribute $G$ (e.g., class, cluster ID). (2) Assuming dimensions are independent, resulting in a diagonal $\Gamma$ matrix with $O(d)$ non-zero entries. Both generalizations can be combined resulting in multiple data summaries

assuming independence or dependence, covering a wide spectrum of models. Generalization (1) can potentially benefit classification models like Naive Bayes and Linear Discriminant Analysis Generalization (2) is the gateway to compute simple descriptive statistics like the mean and standard deviation on multiple data subsets and also clustering models. In general, both generalizations require iterative methods, reading the data set at every iteration. Therefore, their solution is different from the one-pass solution presented in this work, and it is an issue for future work.

We now present customized algorithms to compute each model $\Theta$ under our unifying 2-phase method.

### 4.4.1 Principal Component Analysis (PCA)

The objective of PCA is to reduce the noise and redundancy of dimensions by re-expressing the data set $X$ on a new orthogonal basis, which is a linear combination of the original dimensions basis. In this case, $X$ has $d$ potentially correlated dimensions but no $Y$. In general, PCA is computed on the covariance or the correlation matrix of the data set [22].

We focus on computing PCA on the covariance matrix $V$ or the correlation matrix $\rho$ [22] of $X$, both symmetrical matrices. The PCA model $\Theta$ consists of $U$, a set of $d$ orthogonal vectors, called the principal components of the data set, ordered in decreasing order by their length (variance) and a diagonal matrix $D^2$, with the squared eigenvalues (lengths). PCA is computed with an eigen decomposition of $V$ or $\rho$, whose solution is the factorization $\rho = UD^2U^T = (UD^2U^T)^T$. We stress that we are not computing the SVD factorization $X = UDV^T$, where $U \neq V$ as that is a more

general numerical analysis problem [13, 22]. The correlation matrix, $\rho$, is generally preferred because it normalizes dimensions. Notice that when $X$ has been normalized (i.e. with a z-score, centering at the origin subtracting $\mu$ and rescaling variables by standard deviation $\sigma_j$) $\rho = XX^T/n$. Therefore, given $\rho$, its eigen decomposition [22] is a symmetric matrix factorization expressed as $\rho = (XX^T)/n = UD^2U^T$. In general, only $k$ principal components ($k < d$) carry the most important information about the variance of the data set and the remaining $d-k$ components can be safely discarded to reduce $d$. The actual reduction of $d$ to some lower dimensionality $k$ (i.e. the $k$ principal components) requires an additional matrix multiplication, which is simpler and faster than computing $\Theta$.

PCA can be computed solving SVD on the correlation matrix. To compute PCA we rewrite the correlation matrix equation based on the sufficient statistics in $\Gamma$:

$$\rho_{ab} = (nQ_{ab} - L_aL_b)/(\sqrt{nQ_{aa} - L_a^2}\sqrt{nQ_{bb} - L_b^2}) \tag{4.6}$$

The PCA algorithm two phases are:

1. Compute $\Gamma$.
2. Compute $\rho$ (or covariance matrix $V$), solve SVD of $\rho$ (or $V$). Optional: select the $k$ principal components (generally whose eigen-value is at least 1).

We omit discussion of the actual dimensionality reduction of $X$ to a $k$-dimensional data set. This computation is straightforward requiring only matrix multiplication between a $d \times k$ matrix $\Gamma$ derived from SVD and $X$.

### 4.4.2 Linear Regression (LR)

Let $X = \{x_1, \ldots, x_n\}$ be a set of $n$ data points with $d$ explanatory variables and $Y = \{y_1, \ldots, y_n\}$ be a set of values such that each $x_i$ is associated to $y_i$. The linear regression model characterizes a linear relationship between the dependent variable and the $d$ explanatory variables.

Using matrix notation, the data set is represented by matrices $Y$ $(1 \times n)$ and $X$ $(d \times n)$, and the standard definition of a linear regression model is:

$$Y = \beta^T \mathbf{X} + \epsilon \tag{4.7}$$

where $\beta = [\beta_0, \ldots, \beta_d]$ is the column-vector of regression coefficients, $\epsilon$ represents the Gaussian error and for mathematical convenience $\mathbf{X}$ represents $X$ augmented with an extra row of $n$ 1s stored on dimension $X_0$. The vector $\beta$ is usually estimated using the ordinary least squares method [28], whose solution is:

$$\hat{\beta} = (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}Y^T \tag{4.8}$$

As introduced above, $\Gamma$ contains those 2 partial matrix products and so $\hat{\beta} = Q^{-1}(\mathbf{X}Y^T)$. Therefore, the LR algorithm becomes:

1. Compute $\Gamma$.
2. Solve $\hat{\beta}$ exploiting $\Gamma$.

### 4.4.3 Variable Selection (VS)

We now focus on one of the hardest analytical problems: variable selection in statistics, feature selection in machine learning. Because we have previously introduced linear regression, we will study variable selection in this model. However, our algorithms have the potential to be used in other models such as probit models [22], Bayesian classification [22], and logistic regression [22]. All of which have similar intermediate matrix computations.

The input is $X$, like LR, but it is necessary to add model parameters. In general, a linear regression model is difficult to interpret when $d$ is large, say more than 20 variables. The search for the best subsets of explanatory variables that are good predictors of $Y$ is called a variable selection. The assumption of this search is that the data set contains redundant variables or they have low predictive accuracy and thus they can be safely excluded from the model. When $d$ is high, an exhaustive search on the $2^d$ subsets of variables is computationally intractable for even a moderately large $d$. Traditional greedy methods, for instance, stepwise selection [22] can only find one subset of variables, likely far from the optimal, but the probabilistic Bayesian approach aims to identify several promising subsets of variables, which in theory can be closer to the optimal. In this work, the set of selected variables will be represented by a $d$-dimensional vector $\gamma \in \{0,1\}^d$, such that $\gamma_j = 1$ if the variable $j$ is selected, and $\gamma_j = 0$ otherwise. We denote by $\Theta_\gamma$ the model that selects $k$ out of the $d$ variables, corresponding to the vector $\gamma$. The dot product $k = \gamma^T \cdot \gamma$ indicates how many variables were selected. Throughout this dissertation, we will use $\gamma$ as an index to project matrices on selected variables such as $\beta_\gamma$ and $\mathbf{X}_\gamma$.

## MCMC Methods: the Gibbs Sampler for VS

We use the Bayesian formulation and Gibbs sampler introduced in [28] based on a Zellner G-prior, which has the outstanding feature of sharing several intermediate matrix computations with the other models defined above. Given a model $\Theta$ with $K$ parameters, one of the parameters is sampled from its prior distribution, while the others $K-1$ parameters remain fixed. The set of variables selected at iteration $i$ of the sequence of observations is denoted by a vector $\gamma^{[i]}$. The Gibbs sampler for variable selection generates the Markov chain: $\gamma^{[0]}, \ldots, \gamma^{[N]}$, in which it is expected the best variable subsets will appear more frequently. At each iteration $I$, the Gibbs sampler generates $\gamma^{[I]}$ based on the previous state of the Markov chain, $\gamma^{[I-1]}$. Since each coordinate $\gamma_j^{[I-1]}$ is a parameter of model $\Theta_\gamma$, the Gibbs sampler visits all $d$ entries of $\gamma^{[I]}$. In more detail, for every variable $\gamma_j^{[i]}$ the normalized probabilities $p(\gamma_j^{[i]} = 0)$ and $p(\gamma_j^{[i]} = 1)$ are calculated. Based on these probabilities either $\gamma_j^{[i]} = 0$ or $\gamma_j^{[i]} = 1$ is chosen by sampling and the $j$th position of vector $\gamma_j^{[i]}$ is updated accordingly. The Markov chain becomes stable after an initial number of iterations, known as the *burn-in* period $B$. Then, the MCMC method iterates for a large number of iterations until the posterior probabilities of $\gamma$ have been sufficiently explored (essentially a CPU bound computation, which requires many trial and error runs).

Recall that we performed the variable selection with an MCMC method, potentially requiring thousands of iterations. We base the computation of $\pi(\gamma|X, Y)$ on the Zellner G-prior due to [28]. Zellner's G-prior relies on a conditional Gaussian prior for $\beta$ and an improper (Jeffrey's) prior for $\sigma^2$ [28], with parameters $c$ and $\widetilde{\beta}$. The most common prior probability for $\gamma$ is the uniform prior $\pi(\gamma|X) = 2^{-d}$ [28].

Under Zellner's G-prior, the parameter $c$ influences the number of variables selected by the model: large $c$ values lead to parsimonious models (few variables), whereas small values of $c$ promote saturated models (many variables). On the other hand, $\tilde{\beta}$ is a hyper-parameter [28] (i.e. a new parameter not belonging to the original linear regression model) to impose a Gaussian on $\beta$.

$$\beta|\sigma^2, X \sim \mathcal{N}_{k+1}\left(\tilde{\beta}, c\sigma^2\left(XX^T\right)^{-1}\right) \tag{4.9}$$

$$\sigma^2 \sim \pi\left(\sigma^2|X\right) \propto \sigma^{-2} \tag{4.10}$$

For notational convenience $\mathbf{X}_\gamma, \tilde{\beta}_\gamma, \mathbf{Q}_\gamma$ mean projecting $\mathbf{X}, \tilde{\beta}, \mathbf{Q}$ on selected variables. Recall $\mathbf{X} = [1, X]$. The full equation to get the posteriors of $\gamma$ involves:

$$\begin{aligned}\pi(\gamma|X,Y) \propto \quad & (c+1)^{-\frac{k+1}{2}}\left(YY^T\right. \\ & -\tfrac{c}{c+1}(Y\mathbf{X}_\gamma{}^T)(\mathbf{Q}_\gamma)^{-1}(\mathbf{X}_\gamma Y^T) \\ & \left.-(c+1)^{-1\tilde{\gamma}}\beta_\gamma^T\mathbf{Q}_\gamma\tilde{\beta}_\gamma\right)^{-n/2}\end{aligned}$$

The Zellner G-prior is computationally convenient for big data because equations can be expressed in terms of $\Gamma$. We will use the sufficient statistics by equations 4.1, 4.2 and 4.3 in order to avoid recomputing those matrix products at each iteration. Therefore, our optimized algorithm becomes

1. Compute $\Gamma$, which contains $\mathbf{Q}_\gamma$, $\mathbf{X}_\gamma Y^T$ and $YY^T$.

2. Iterate the Gibbs sampler a sufficiently large number of iterations to explore $\pi(\gamma|X,Y)$.

## 4.5 Time Complexity and Parallel Speedup

Our analysis is based on the 2-phase algorithm introduced in Section 4.3. We start with time complexity. Phase 1, which corresponds to the summarization matrix operator, is the most important. Computing $\Gamma$ with the dense matrix operator is $O(d^2n)$. On the other hand, computing $\Gamma$ with the sparse matrix operator is $O(k^2n)$ for the average case assuming $k$ entries from $x_i$ are non-zero. Assuming $X$ is hyper-sparse $k^2 = O(d)$ then the matrix operator is $O(dn)$ on average. Space required by $\Gamma$ in main memory with a dense representation is $O(d^2)$. In Phase 2 we take advantage of $\Gamma$ to accelerate computations involving $X$. Because we are computing matrix factorizations derived from $\Gamma$ time is $\Omega(d^3)$, which for a dense matrix it may approach $O(d^4)$, when the number of iterations in the factorization numerical method is proportional to $d$. In short, the time complexity for Phase 2 for the models we consider does not depend on $n$. I/O cost is just reading $X$ with the corresponding number of chunk I/Os in time $O(dn)$ for dense matrix storage and $O(kn)$ for sparse matrix storage.

We now analyze the parallel speedup assuming $N$ processors on a shared-nothing parallel architecture. Recall that we assume $d < n$ and $n \to \infty$. For Phase 1 each chunk of $X$ is hashed to $N$ processors. Because we assume $x_i$ fits in one chunk, this results in $x_i$ being hashed as well. Our matrix operator can compute each outer product $x_i \cdot x_i^T$ in parallel, resulting in $O(d^2n/N)$ work per processor for a dense matrix and $O(dn/N)$ work per processor for a sparse matrix. Both dense and sparse matrix operators become optimal as $N$ increases or $n \to \infty$. We now consider Phase 2, which involves numerical methods from linear algebra. Since $\Gamma$ significantly reduces

problem complexity and the numerical methods used to solve SVD, least squares and matrix inversion are mathematically complex, we assume matrix factorizations are computed on one node ($N = 1$) calling the efficient LAPACK library. In other words, it is not worth reprogramming SVD, least squares, or matrix inversion to run in parallel across $N$ nodes, but they can indeed be run in parallel on one node with a multicore CPU, taking advantage of LAPACK. Notice the MCMC method used by VS is difficult to parallelize across iterations since each iteration depends on the previous one. It makes sense computing each iteration, requiring a matrix inversion, as fast as possible. Therefore, for Phase 2, there are two choices for computation: totally sequential or parallel with scale up (increasing number of cores and threads). In short, for Phase 2, we rely on the parallel and numeric capabilities provided by LAPACK, which is a gold standard for numeric accuracy and which provides superb performance on one node with multicore CPUs.

## 4.6 Data Summarization in Parallel Database Systems

In this section, we elaborate on the system implementation details in two parallel database systems with $N$ processing units in a shared-nothing architecture (i.e. $N$ nodes, each with its own memory and disk space). We also discuss in detail the general and system-dependent optimizations.

### 4.6.1 Data summarization in SciDB

A parallel array DBMS enables the manipulation of multidimensional arrays, removing RAM limitations in a large matrix computation. The fundamental difference between SciDB and a row or column DBMS is storage by chunk instead of block. Such chunk storage requires different programming and optimization, compared to SQL queries or UDFs. Moreover, it is feasible to develop an operator that receives a matrix as input and produces a matrix as output, something not easy to do in SQL.

SciDB organizes array storage by chunks [43], where each chunk is a large multidimensional block on secondary storage that stores a preferably large fraction of cells (values) for one numeric attribute (e.g., a subarray). Array storage in SciDB can be visualized as a grid of rectangular chunks (tiles in 2D), where each chunk comprises a rectangular area of contiguous cells. Storage for dense and sparse arrays is different, but SciDB allows an efficient migration to a dense representation as a matrix gets denser.

### 4.6.2 Programming Mechanisms in SciDB

The major question is how to program the computation of $\Gamma$ in SciDB. Choices are: (1) exporting (with a bulk mechanism) the $X$ matrix to a linear algebra package (LAPACK); (2) using relational style queries on arrays in the AQL (Array Query Language) language provided by the DBMS; (3) composing existing array operators in the AFL (array functional language) language; (4) developing a new array operator, tailored to $\Gamma$. Exporting the input matrix is a bad idea, as it defeats the purpose of the DBMS, no matter how fast the computation can happen outside. The

query language is a natural second option, but it is inefficient as it requires a similar evaluation query plan as an SQL query with a costly relational join operation. In fact, we will show such join is slow even on a fast column DBMS. Existing matrix operators are inefficient and limited by RAM. These alternatives were experimentally compared in [37] and our new matrix summarization operator was the winner.

### 4.6.3 The Gamma Array Operator to Summarize a Matrix

As pointed out by previous research [35], it is easy to compute only $n, L$, without $Q$, because they only require sums of a value (counting) and incrementally summing a $d$-vector. That is, $Q$ makes the computation more demanding. Recalling that $\Gamma$ contains $n, L$, and $Q$ as submatrices, the main challenge is to compute this matrix product:

$$\Gamma = \mathbf{Z}\mathbf{Z}^T = \sum_{i=1}^{n} z_i \cdot z_i^T,$$

which requires cross-products between all dimension pairs, an $O(d^2n)$ computation. That is, we study the summarization operator purely as a matrix multiplication that subsumes all previous approaches. A fundamental aspect is to optimize computation when $\mathbf{X}$ (and therefore $\mathbf{Z}$) is sparse; any multiplication by zero returns zero. Therefore, when computing $\mathbf{Z}\mathbf{Z}^T$ a multiplication should be evaluated only when both vector $z_i$ entries are different from zero. Since dense and sparse matrices have different storage and different processing in main memory, this leads specialized summarization operators for each of them.

### Dense Matrix Operator

The dense operator requires that all data of $z_i$ ($x_i$ in consequence) fits in one chunk. In other words, $z_i$ cannot be partitioned into several chunks in spanned storage. This requirement is important to accelerate I/O: larger chunks improve I/O because our $\Gamma$ computation requires a full scan of the data set, without join algorithms. Otherwise, if $d$ is so large that it is necessary to store $x_i$ spanning several chunks, then it would be necessary to call a join algorithm, joining $X$ with itself.

Smaller chunks would trigger seeks, resulting in poorer performance. Fitting many points $x_i$ in one chunk is not an unreasonable assumption because SciDB favors fairly large chunk sizes, typically ranging above 8 MB. Another major requirement and advantage is that the operator must work in parallel. It is a requirement that complicates programming because it is essential to design and develop the operator in such a way that $X$ can be evenly partitioned across $N$ processing nodes, with minimal or no need of synchronization overhead. The advantage is, of course, the ability to scale processing to larger $N$ as $n$ grows. Our $\Gamma$ operator works fully in parallel with a partition of $X$ into $N$ subsets $X^{[1]} \cup X^{[2]} \cup \cdots \cup X^{[N]} = X$, where we independently compute $\Gamma^{[I]}$ on $X^{[I]}$ for each node $I$ (we use this notation to avoid confusion with matrix powers and matrix entries). In SciDB terms, each worker, $I$, will compute $\Gamma^{[I]}$. When all workers are done, the coordinator node will gather all results and compute a global $\Gamma = \Gamma^{[1]} + \cdots + \Gamma^{[N]}$ with $O(d^2)$ communication overhead per node (much smaller than $O(dn)$. This is essentially a natural parallel computation [42], because we can push the actual multiplication of $z_i \cdot z_i^T$ into the array operator. Given the additive properties of $\Gamma$, the same algorithm is applied on

each node $I = 1 \ldots N$ to get $\Gamma^{[I]}$, combining all partial results $\Gamma = \sum_I \Gamma[I]$ in the coordinator node. Our parallel algorithm to compute $\Gamma$ is below.

```
/* X on secondary storage, Γ in main memory */
Data: X = {x_1, x_2, ..., x_n} = X^[1] ∪ ··· ∪ X^[N]
Output: Γ
/* Parallel for:
   N nodes working in parallel on partition X^[I] */
for I = 1 ... N do
    Γ^[I] ← 0
    for x_i ∈ X^[I] do
        z_i = [1, x_i, y_i]
        for a = 0 ... d + 1 do
            for b = 0 ... a do
                Γ^[I]_ab ← Γ^[I]_ab + z_ia * z_ib
            end
        end
    end
end
/* send local summaries to coordinator node */
Γ = Σ_{I=1}^{N} Γ^[I]
```

**Algorithm 3:** The parallel dense algorithm to compute lower triangular $\Gamma$ on a dense matrix.

### Sparse Matrix Operator

A sparse matrix uses less space on secondary storage, resulting in faster I/O and a smaller footprint in RAM per point. A second advantage is that since more cells fit in one chunk $d$ can be higher compared to the dense operator. Therefore, assuming a density threshold, $\psi$, $x_i$ can have higher dimensionality being able to fit in one chunk. If $p$ is the maximum dimensionality for a dense matrix representation, then $d = \lceil p/\psi \rceil$ can be the maximum dimensionality for a sparse matrix. For instance, if $\psi = 0.01$ then $d = 100 \times p$.

In a similar manner to a dense matrix, we assume $x_i$ fits in one chunk. Moreover, since $x_i$ is a sparse vector, its coordinates reside on the same chunk improving I/O locality. Under this assumption, the algorithm for a sparse matrix has the same outer loop on $i$ to scan partition $X^{[I]}$, compared to the algorithm for a dense matrix. The first major difference is that we *dynamically* build a main memory sparse vector representation for $z_i$, following a similar scheme to LAPACK [14], whose number of non-zero entries is $k = |z_i|$. The loop to build $z_i$ is $O(d)$ (and not $O(k)$) because SciDB does not store matrix entries as (subscript,value) pairs. Instead, SciDB uses Run-Length Encoding for a sequence of zeroes, which consumes more disk space, but allows an easy gradual change from sparse to dense representation. The second major difference is the two inner loops, which go $a = 1 \ldots k, b = 1 \ldots k$ which result in $O(k^2)$ flops. Such optimization results in a much better time complexity $O(k^2 n) < O(d^2 n)$. Assume that $X$ is hyper-sparse [9] so that $k^2 \leq d$ (i.e. $k = O(\sqrt{d})$; the ratio of the number of non-zero entries to $d$ asymptotically approaches zero). Then time complexity becomes $O(dn)$, which is significantly lower than $O(d^2 n)$. In a similar manner to the dense operator, there is $O(d^2)$ communication overhead per node (much smaller than $O(dn)$ if the data set had to be transferred) to get the global summary matrix, $\Gamma$, in time $O(d^2 N)$ using a locking mechanism to update the matrix. Finally, $n_{nz} \leq \sqrt{d}n$ can be used as a threshold to use the sparse algorithm, where $n_{nz}$ is the number of non-zero entries in $X$.

The sparse operator brings two computational aspects into question. The first issue is the ability to maintain $\Gamma$ in RAM for high $d$. The second one is developing a third operator for a sparse $\Gamma$ matrix. We emphasize that for most big data analytics problems $d < n$. For the first issue, it is reasonable to maintain $\Gamma$ for fairly high

```
/* X on secondary storage, Γ in main memory */
Data: X = {x_1, x_2, ..., x_n} = X^[1] ∪ ··· ∪ X^[N]
Output: Γ
/* Parallel for:
   N nodes working in parallel on partition X^[I] */
for I = 1...N do
    Γ^[I] ← 0
    for i = 1...|X^[I]| do
        k ← 1 /* z_i0 = 1 */
        z_i ← [1, x_i, y_i]
        for j = 0...d+1 do
            if z_ij ≠ 0 then
                v_k ← z_ij
                i_k ← j
                k ← k + 1
            end
        end
        /* k = |z_i| */
        for a = 1...k do
            for b = 1...a do
                Γ^[I]_{i_a i_b} ← Γ^[I]_{i_a i_b} + v_{i_a} * v_{i_b}
            end
        end
    end
end
/* send local summaries to coordinator node */
Γ = Σ_{I=1}^{N} Γ^[I]
```

**Algorithm 4:** The parallel sparse algorithm to compute lower triangular Γ on a sparse matrix.

$d$ in RAM given the rapid growth in RAM volume (we experimentally show this is feasible for $d \approx 1000$). For the second issue, in general, a sparse representation for $\Gamma$ does not make sense because that would imply the dataset contains zero-constant dimensions to start with (i.e. useless). Moreover, any constant dimension should be detected and eliminated, alerting the user. For such cases, it makes sense to discard constant dimensions on a pre-processing pass computing only $n, L$ and the diagonal of $Q$. That is, computing only row 0 (column 0) and the diagonal of $\Gamma$ (no dimension cross-products). The user can now safely discard any dimension, $j$, whose $\sigma_j = 0$. This a consequence of $\Gamma$ containing an extended set of sufficient statistics from which we can derive $\mu_j, \sigma_j$ for dimension $j$.

### 4.6.4 Additional Considerations

#### Data quality: Constant dimensions and missing values

Our $\Gamma$ summarization matrix can help detect and fix data quality problems. Because $\mu_a, \sigma_a$ can easily be derived from $\Gamma$, the $\Gamma$ matrix can be used to eliminate constant and near-constant dimensions where $\sigma_a \approx 0$.

Consider the trivial, but common, case that $\Gamma_{aa} = 0$ for $a = 1 \ldots d+1$ (remember that $\Gamma_{00} = n$). This means that dimension $a$ is completely full of zeroes: clearly such dimension is useless and should be eliminated from further consideration. In a more general case, consider constant dimensions with non-zero values. If dimension $a$ is constant then the variance of dimension $a$ is zero, then all covariances $\Sigma_{ab}$ or correlations $\rho_{ab}$ are undefined (null in SQL). The reason is straightforward: a constant dimension provides no statistical insight to the analyst (other than alerting

the analyst that something may be wrong in data collection). Moreover, PCA and linear regression require eliminating constant dimensions because their intermediate matrices cannot be factorized or inverted. In short, we assume that before computing model $\Theta$, all constant and near-constant dimensions have been eliminated, and any missing values have been replaced or their points eliminated.

### Systems Aspects

Our operators were programmed in C++ with the API provided by SciDB. The operator was programmed as an array aggregation (Figure 4.2), which has similarities with an aggregate UDF in a relational DBMS [35]. The similarities are: the ability to update variables in main memory, seamless integration with a scan over the input array, parallel execution with just one synchronization barrier at the end, programmed in an efficient language (the same as the DBMS: C++), similar processing phases (initialization, increment, final, and return), and not being able to call other array operators within the operator source code. However, as fundamental differences, we must mention: our array operator receives array chunks as input (not rows) and it produces an array as output, not a scalar value. Therefore, it is unnecessary to do data type conversion like in a relational DBMS [35]. The ability to define a parallel array operator that receives an array as input and produces an array as output was fundamental for us.

We now discuss why the dense and sparse operators working together with R can compute models faster than alternative approaches. Specifically, our array operator is much faster than an implementation on Hadoop-MapReduce systems (e.g., Spark [50]) and it is much faster than any SQL-based engine, including UDFs. This is a

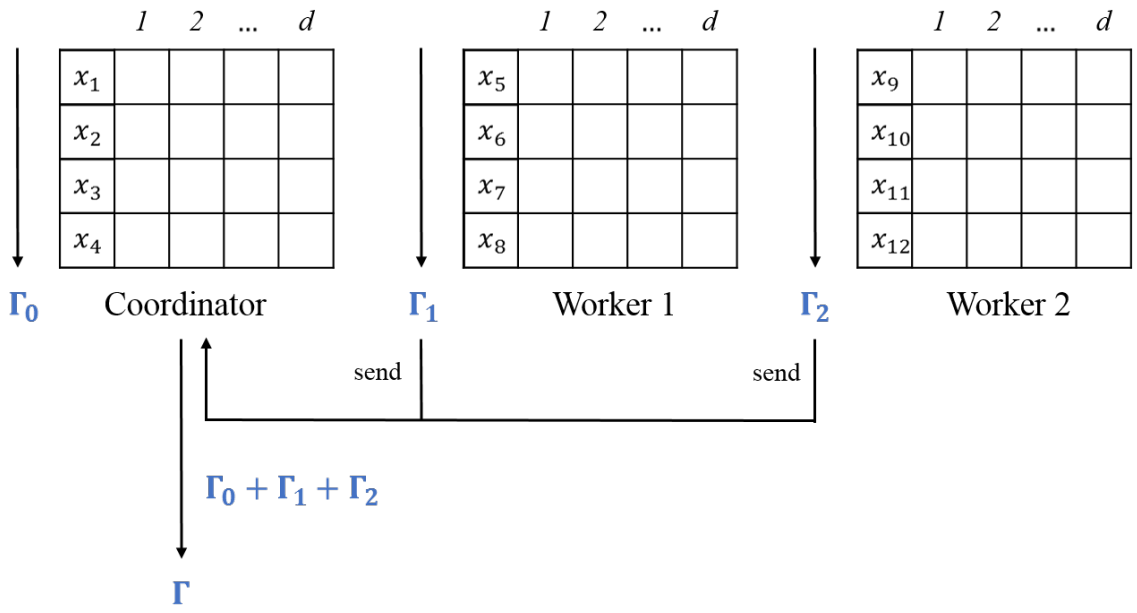



Figure 4.2: How the Gamma operator works in SciDB

list of main reasons. Our operator works in a single array scan with large disk blocks (chunks) and these chunks are fairly large (>8MB) [43] (and growing in the future as RAM grows). The operators work with compiled C++ code: there is no overhead or delay parsing a query, query optimization at run-time, or ODBC communication overhead. Moreover, compiled C++ code is faster than Java byte code and faster and more memory-efficient than Java compiled code (widely used on Hadoop). Since we use an array operator that takes an array as input and an array as output, there is no overhead converting matrices between different representations. Our operator cannot be as efficient as compiled FORTRAN code used by LAPACK for small matrices, where small means a matrix that can fit in the cache memory in the CPU. However, our operator scales much better as matrices get bigger and they surpass cache memory size. In fact, we will show our operator surpasses LAPACK speed

for very large sparse matrices, which is the state of the art for high performance computing. Such comparison settles any speed concerns about our operator. On the parallel processing side, the input matrix storage layout is key to enable a fully parallel computation without needing to send data from one node (or processing thread) to another one. Our main assumption is that $x_i$ can fit completely in one chunk: we will show experiments where $d = 800$, a truly high dimensionality. Moreover, we can solve even higher $d$ problems with sparse matrices, the norm in big data analytics. While it is true, data shuffling in a distributed system may be needed for high dimensional data we will experimentally show we can solve fairly high $d$ problems. Considering pure C++ code reading a binary file our operator will likely be faster simply because SciDB has an efficient I/O subsystems, processing is multi-threaded and access is block-based. In other words, a well-crafted C++ program would need to mimic or redo the basic SciDB array storage. Therefore, our system runs much faster and is more scalable than any other C++, LAPACK, SQL or Hadoop-based system. Our experiments will prove that is the case.

### 4.6.5 Accelerating Summarization with a GPU

Computing $\Gamma$ by evaluating $\mathbf{Z}\mathbf{Z}^T$ is a bad idea because of the cost of computing and materializing $\mathbf{Z}^T$. Instead, we evaluate the sum of vector outer products $\sum_{i=1}^{n} z_i \cdot z_i^T$, which is easier to parallelize and update in RAM. Moreover, since $\Gamma$ is symmetric, we only compute the lower triangle of the matrix during computation to save execution time. Our $\Gamma$ matrix multiplication algorithm works fully in parallel with a partition of $X$ into $N$ subsets $X^{[1]} \cup X^{[2]} \cup \cdots \cup X^{[N]} = X$, where we independently compute $\Gamma^{[I]}$ on $X^{[I]}$ for each processor $I$. A fundamental aspect is

to optimize computation when $\mathbf{X}$ (and therefore $\mathbf{Z}$) is sparse: any multiplication by zero returns zero. Specifically, $n_{nz} \leq \sqrt{dn}$ can be used as a threshold to decide using a sparse or algorithm, where $n_{nz}$ is the number of non-zero entries in $X$. Computing $\Gamma$ on sparse matrices with a GPU is challenging because tracking and partitioning non-zero entries may reduce parallelism by adding more complex logic in the operator code. In this dissertation, we focus on the dense matrix algorithm.

Our following discussion focuses on integration with an array DBMS, given its ability to manipulate matrices. From a systems perspective, we integrated our model computation using $\Gamma$ with the SciDB array DBMS and the R language, with a 2-phase algorithm: (1) data summarization in one pass returning $\Gamma$; (2) exploiting $\Gamma$ in intermediate computations to compute model $\Theta$. We emphasize $z_i$ is assembled in RAM (we avoid materializing $z_i$ to disk). Unlike normal UDFs in a traditional DBMS, which usually need to serialize a matrix into binary/string format then deserialize it in succeeding steps, our operators in SciDB returns $\Gamma$ directly in array format, which is a big step forward compared to previous in-DBMS approaches. Phase 2 takes place in R on the master node leveraging R's rich mathematical operators and functions. Although this phase does not run in parallel across nodes, it does not impact the overall performance since $\Gamma$ passed from SciDB in the first step is much smaller than the data set, resulting in much faster iterations working in RAM. That is, Phase 1, computing $\Gamma$, is the main task to parallelize.

Parallel processing happens as follows. In SciDB, arrays are partitioned and stored as chunks, and such chunks are only accessible by C++ iterators. In our previous work [38], we compute the vector outer product $z_i \cdot z_i^T$ as we scan the data set in the operator. We emphasize $z_i$ is assembled in RAM (we avoid materializing

$z_i$ to disk). In general, interleaving I/O with floating point computation is not good for GPUs because it breaks the SIMD paradigm. In our GPU accelerated operator, we first extract matrix entries in each chunk into main memory. Then we transfer the in-memory subsets of $X$ to GPU memory, one chunk at a time. Then the GPU computes the vector outer products $z_i \cdot z_i^T$ fully in parallel with its massive amount of processing units (cores). The sum is always maintained in the GPU memory. It takes $log(n)$ time to accumulate $n$ partial $\Gamma$s into one using the reduction operation. When the computation finishes for the whole dataset, the $\Gamma$ matrix is transferred back from GPU memory to CPU main memory. The C++ operator code is annotated with OpenACC directives to work with GPU. In our current GPU version, the CPU only does the I/O part. Since the DBMS becomes responsible for only I/O, our approach has promise in relational DBMSs.

## 4.7 Data summarization in Columnar DBMS

We start by discussing storage in the columnar DBMS, where indexing is not required. The dataset and model matrices are stored in tables, having a primary key with a specific combination of matrix subscripts, depending on their storage layout. We use two fundamental dataset layouts for $X$: horizontal and vertical. The first layout is a standard tabular representation which has $n$ rows, each having $d$ columns. The second layout is based on a pivoted table with one attribute value per row and up to $dn$ rows, which enables efficient processing of sparse matrices and which also removes DBMS limitations on the maximum number of columns. The I/O efficiency of each layout depends on matrix sparsity and the specific query physical operator

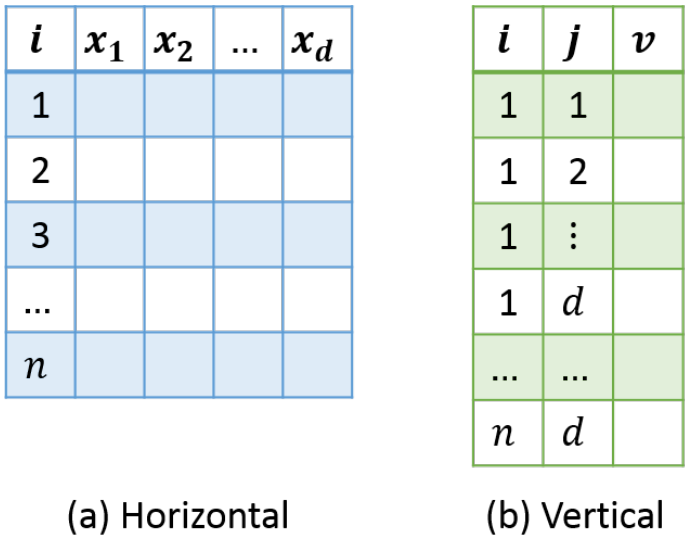



Figure 4.3: The horizontal and vertical layout for $X$ in the columnar DBMS.

(scan, join, sort). Figure 4.3 illustrates the two types of layout.

In order to achieve better performance, the data set will be sorted by the matrix subscripts, and will be segmented and distributed to all the $N$ processing nodes evenly according to the value of a hash function on the row number $i$. This process will be automatic in the columnar DBMS during the data loading phase once we properly define the table at creation time. The following SQL command shows a proper way to define the input table $X$ in Vertica:

**CREATE TABLE** $X$ (
$i$ **INT** NOT NULL **ENCODING** RLE,
$j$ **INT** NOT NULL **ENCODING** COMMONDELTA_COMP,
$v$ **FLOAT**)
ORDER BY $i$, $j$
SEGMENTED BY **MODULARHASH**($i$) ALL NODES;

Computing matrix equations with SQL queries has proven to be hard to optimize [32]. To get started, it is not possible to create fixed queries that can efficiently evaluate mathematical equations because the table definition for the input data set is not fixed (especially for the horizontal layout with $d$ columns). Second, evaluating matrix equations with pure SQL queries usually involves table joins, which are always quite expensive for DBMSs especially when they are evaluated in parallel. Therefore, SQL code generation at run-time from R and developing specialized mathematical UDFs to bypass table joins are necessary. Statistical methods and data mining algorithms are also programmed with R in order to take full advantage of its rich library of models and techniques.

The R program is based on two sets of parameters. The first set of parameters controls SQL code generation (i.e. table name and layout) and database systems optimizations, which are tailored to each algorithm (with some essential math optimizations across multiple models). The second set of parameters controls the mathematical behavior of each technique. Examples include the number of clusters, tolerance threshold for convergence, numeric stability issues, and so on.

We must emphasize that SQL code generated by R represents a Turing-complete language. That is, we can evaluate any equation with matrices, no matter how complex.

As we have discussed in Chapter 4.2, $\Gamma = Z \cdot Z^T$ [37]. $\Gamma$ can be computed by a single SQL query using self-joins like the following:

**SELECT**

a.$j$ **AS** $i$,

b.$j$ **AS** $j$,

**SUM**(a.$v$ * b.$v$) **AS** $v$
**FROM** $X$ a **JOIN** $X$ b **ON** a.$i$ = b.$i$
**GROUP BY** a.$j$, b.$j$;

To bypass the expensive self-join in the SQL query, we can also compute the $\Gamma$ by calling a user-defined transform function that builds vector $z_i$ and updates $\Gamma$ in parallel and incrementally:

**SELECT** $Gamma(i, j, v$ **USING PARAMETERS** $d = \mathbf{d})$
**OVER** (**PARTITION BY MOD**$(i, N)$
**ORDER BY** $i, j)$
**FROM** $X$;

## 4.8 Experiments

We present experiments focusing on scalability and parallel processing. Because our operator and the 2-phase algorithm do not change the accuracy of matrices and final results, it is unnecessary to measure statistical or numeric accuracy. We should emphasize that our summarization operator was used to produce the $\Gamma$ matrix to be consumed by the R package. That is, the final model $\Theta$ computation happened in R. Therefore, the model computed by R alone and the model computed by SciDB in tandem with R are the same.

This is an overview of experiments with a focus on scalability and parallel processing. We first review existing system and justify our choice of specific systems for comparison. We start our evaluation presenting time benchmarking experiments comparing our operator with popular analytic systems, including the R package and

Table 4.2: Data sets.

| Data set | $d$ | $n$ | Description |
|---|---|---|---|
| KDDnet | 38 | 5M | Network intrusion detect (KDD Cup) |
| Gene | 12506 | 248 | Gene data to predict cancer survival |

a fast columnar DBMS on a single computer with a powerful multicore CPU. We should stress R and many math packages internally use LAPACK for the most numerically intensive computations. Therefore, in order to test our matrix operators with large matrices, we also make a careful comparison with the LAPACK [14, 13] library (the state of the art in numerical linear algebra) using its Intel MKL variant (the fastest variant for Intel multicore CPUs). Finally, we analyze scalability and speedup on a large cluster in the cloud with large data sets comparing our R+SciDB system with Spark, currently the most popular analytic platform from the Hadoop stack.

### 4.8.1 Setup

We used two data sets: the network intrusion data set and a microarray genomics data set, summarized in Table 4.2, obtained from the KDD Cup repository (KDD 99), and a cancer data set repository, respectively. We sampled and replicated the KDDnet to get varying $n$ (data set size) and $d$ (dimensionality), without altering its statistical properties. Its columns were replicated three times and rows were replicated to get $d = 100, n = 10\text{M}$. Based on this "reference" data set rows were sampled and replicated in log-10 scale and columns were replicated in log-2 scale. On the other hand, the Gene data set was left as is.

Table 4.3: Hardware and operating system.

| Item | Local | Cloud (Amazon) |
|---|---|---|
| OS | Linux Ubuntu | Linux Ubuntu |
| CPU | 4 cores: 2.15 GHz | 2 cores (2VCPU): 2GHz/worker<br>4 cores (4VCPU): 2GHz/coord. |
| $N$ nodes | 1 | 1, 10, 100 |
| threads | SciDB: 2 instances<br>Spark: NA | SciDB: 2,20,200 instances<br>Spark: 2,20,200 executors |
| RAM | 4 GB | 7.5 GB/node (SciDB/Spark)<br>15 GB/coordinator (Spark) |
| Storage | 3 TB | 10 TB |

#### 4.8.1.1 Mathematical Models Functions and Parameters

For PCA and LR, we used the R default parameter settings to get accurate results. On the other hand, for VS we used a small number of iterations for the MCMC Gibbs sampler since this algorithm is CPU bound (after being optimized) and we were interested in comparing performance, rather than getting an accurate Bayesian model, which would require many trial/error experiments and tens of thousands of iterations (i.e. exploring the posterior probability and evaluating convergence of the Markov Chain). For Spark, we used the available functions it had for matrix multiplication, LR, PCA, and MLlib, the best library currently available. We should mention Spark did not offer functions for variable selection. For LR and PCA, we used the recommended number of iterations by Spark.

#### 4.8.1.2 System Setup and Tuning

We tested our operator and competing systems on two hardware configurations (summarized in Table 4.3): (1) one node ($N = 1$) (4-core CPU) and (2) $N$ nodes

(2-core VCPU). Our default system configurations were two instances for SciDB and two executors per node for Spark (i.e. 2 threads per node). For $N = 1$ data sets were copied to the Linux file system and then loaded into the respective DBMS (array, column). For the $N$ nodes, we used the Amazon cloud, which already offered optimal configuration for HDFS, and where Spark was pre-installed and tuned. The times to transfer, copy, and load the data sets into chunk format for SciDB or the time to copy CSV files from Unix to HDFS for Spark are not included in our time measurements, since it was a one-time event. Processing times include the I/O time to read arrays in chunk format in SciDB, and the time to read the CSV files, and transform them into RDD format [50] which is the distributed memory format in Spark.

For the array DBMS and the column DBMS, we cleared the buffers (DBMS cache) before each run to make sure measurements include the time to read from disk. We ran each program three times on each system and we reported the average. When the R system crashed, mainly due to RAM limitations, we reported "fail". For Spark, we made sure the data set was cleared from distributed memory before each run, but Spark could cache the data set for iterative algorithms (LR, PCA). That is, we attempted to make a fair comparison, giving each system the best opportunity. In general, when the computation could not finish in 30 min it was automatically stopped and we reported "stop". We show execution times in seconds. In some isolated cases we let the system run for 1 hour.

### 4.8.2 Comparing Analytic Systems (N = 1 node)

#### 4.8.2.1 Comparing R+SciDB and R alone

We start by comparing our proposed hybrid solution combining the array DBMS and the R package to the R package alone, assuming a dense matrix as input. We did not compare with a sparse matrix storage in R because most statistical models in R are computed with an input matrix with a dense layout. Moreover, a sparse data set requires a significantly different storage structure in RAM, which would require making at least two passes to convert the data set from dense to sparse storage. For PCA, we called the princomp() function and for LR, we called solve(), both standard R function calls widely used by R analysts. For VS in R, we use the R script available from [28] to solve Bayesian variable selection with the same method. The original R script from [28] was so slow that it could not run in under one hour even for $n$=1k (i.e. when the data set fits completely in RAM). Therefore, we were forced to incorporate our optimization based on $\Gamma$, but as a matrix multiplication. Incorporating a faster optimization based on vector outer products would require developing a C++ function working by block in R, converting the R matrix representation to native C++ arrays, back and forth, which we consider a topic for future research. The SciDB dense matrix operator computes $\Gamma$, which is passed to an R script further optimized by us. We would like to mention our time measurements on SciDB and the column DBMS do not include the time to load the data set and apply an array redimension operator to convert it from table to matrix because, in practice, they are a one-time event.

Table 4.4 compares the total time to get $\Theta$ in R alone and the combination

Table 4.4: Comparing computation of model $\Theta$ using R and DBMS+R; dense matrix operator; data set KDDnet; local server; times in secs.

| $d$ | $n$ | PCA | | LR | | VS | |
|---|---|---|---|---|---|---|---|
| | | R | R+SciDB | R | R+SciDB | R | R+SciDB |
| 10 | 100k | 0.5 | 0.6 | 0.5 | 0.6 | 3.4 | 3.6 |
| 10 | 1M | 4.8 | 1.3 | 5.6 | 1.3 | 7.6 | 4.7 |
| 10 | 10M | 45.2 | 7.0 | 50.1 | 7.1 | 58.8 | 9.6 |
| 10 | 100M | fail | 64.7 | fail | 69.8 | fail | 93.4 |
| 100 | 100k | 5.7 | 2.5 | 6.3 | 2.6 | 34.4 | 14.8 |
| 100 | 1M | 61.2 | 16.8 | 60.5 | 16.9 | 113.0 | 29.1 |
| 100 | 10M | fail | 194.9 | fail | 194.9 | fail | 207.2 |

DBMS+R. We can see R scales well, but eventually crashes upon reaching RAM capacity. For all models, our $\Gamma$ operator on R+SciDB is faster than R working alone. These times prove the superiority of our operator and our 2-phase algorithm. A combination of factors make R slower: parsing the input text file, single-threaded processing, but mainly not exploiting our summarization matrix.

We now turn our attention to the Gene data set, with very high $d$, analyzed in Table 4.5. The authors of [28] provide an R program that computes a state-of-the-art Bayesian model that was our reference Gibbs sampler algorithm. This R program was so slow that we had to incorporate our $\Gamma$ summarization matrix computed in R as a matrix product. Since $d > n$ temporary matrices (on selected variables) were ill-conditioned and the MCMC method did not converge properly. That is, they could not be inverted or factorized. Convergence with such a high $d$ required hundreds of thousands of iterations, becoming prohibitive. Finally, variables with marginal correlation to $Y$ had little statistical significance. Therefore, we ran an initial variable pre-selection step computing correlations between each variable and $Y$, to obtain a reduced dimensionality $p$, in which the MCMC method converged

Table 4.5: Bayesian variable selection (VS) in gene data set ($d = 12506, n = 248$): Comparing R+SciDB with R at 1000 iterations; times in secs.

| $p$ | R+SciDB | R unoptimized | R optimized with $\Gamma$ |
|---|---|---|---|
| 100 | 17 | 1139 | 16 |
| 200 | 43 | * | 43 |
| 400 | 83 | * | 85 |
| 800 | 199 | * | 200 |

properly. As seen in Table 4.5, $\Gamma$ is essential to accelerate computation and R (optimized) was competitive with R+SciDB because the data set could fit in RAM. We emphasize our optimization had a significant impact on R alone, highlighting its generality.

#### 4.8.2.2 Computing $\Gamma$ on R, Array DBMS, and Column DBMS

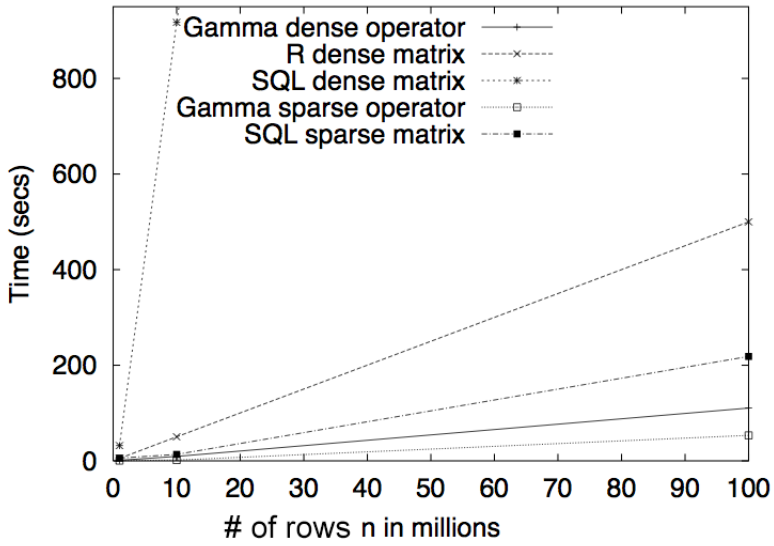
It is natural to wonder if $\Gamma$ can be computed by the other systems, beyond the array DBMS. With this question in mind, we investigated how $\Gamma$ can be computed on each system, making a reasonable effort to develop an efficient program within each system constraints.

For R, the data set was read from disk and loaded in RAM and then $\Gamma$ was obtained with a standard matrix multiplication using %*% (i.e. the fastest mechanism available, commonly used by R analysts). For the column DBMS, we stored $X$ on a vertical layout with one entry $X_{ij}$ per row $(i, j, v)$, where $i = 1 \ldots n$ and $j \in \{1 \ldots d\}$. The query to get $\Gamma$ was a self-join on $i$ grouping the two dimension subscripts. We defined the table sorted by $i$ so that all dimension values were clustered on disk; in this manner, the query optimizer could use a merge-join, the fastest join possible

Table 4.6: Comparing matrix summarization algorithms; data set KDDnet; dense ($dn$ entries) and sparse (zeroes deleted); Local 1/2; times in secs.

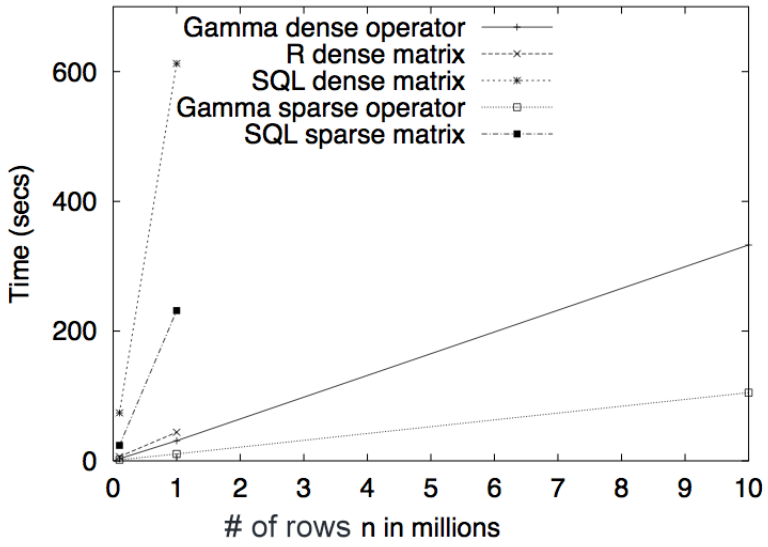
| | | Dense matrix | | | Sparse matrix | |
|---|---|---|---|---|---|---|
| $d$ | $n$ | R | SQL | $\Gamma$ dense op. | SQL | $\Gamma$ sparse op. |
| | | | | | density=12% | |
| 10 | 100k | 0.6 | 3.8 | 0.1 | 0.6 | 0.1 |
| 10 | 1M | 5.1 | 31.9 | 1.1 | 5.8 | 0.5 |
| 10 | 10M | 50.2 | 917.2 | 9.2 | 13.4 | 1.6 |
| 10 | 100M | fail | stop | 110.3 | 218.4 | 53.3 |
| | | | | | density=27% | |
| 100 | 100k | 6.5 | 74.0 | 3.3 | 23.9 | 1.8 |
| 100 | 1M | 44.1 | 612.5 | 31.0 | 231.6 | 10.6 |
| 100 | 10M | fail | stop | 332.7 | stop | 105.2 |
| 100 | 100M | fail | stop | 3423.7 | stop | 1015.3 |

in time $O(n)$. Figure 4.4 shows the comparison of the three systems: the upper plot compares at low dimensionality, $d = 10$, and the lower plot compares at a high dimensionality, $d$=100. For sparse matrix experiments, zeroes are deleted, resulting in a smaller SQL table and a smaller (compressed) array on the disk. As can be seen in the plots, the fastest computing mechanisms are our sparse and dense operators, with the sparse operator being at least twice as fast compared to the dense one, with matrices having the same $d$, and $n$ sizes. In general, our operator is about an order of magnitude faster than R and about two orders of magnitude faster than the SQL query. Table 4.6 complements Figure 4.4, providing a "drill down" view on the specific numbers, demonstrating our operator is both scalable and uniformly more efficient, showing when R failed due to insufficient RAM and indicating when the SQL DBMS had to be stopped for waiting for more than 30 minutes. These results make clear SQL cannot solve dense matrix problems, but it shows promise for sparse matrices of low dimensionality.




(a) d=10

-




(b) d=100

Figure 4.4: Comparing systems to compute $\Gamma$ on a local server: Array DBMS, R, and SQL.

Table 4.7: Comparing the dense operator (working on disk+RAM), the sparse operator (working on disk+RAM), and the dense matrix multiplication in MKL BLAS (parallel LAPACK, working only on RAM) to compute Γ; local server 4GB RAM; synthetic data sets; times in secs.

| | | $d = 100$ | | | $d = 200$ | | | $d = 400$ | | | $d = 800$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ | density | dense | sparse | MKL | dense | sparse | MKL | dense | sparse | MKL | dense | sparse | MKL |
| 1M | 0.1% | 27.5 | 0.2 | 3.0 | 94.4 | 0.2 | 8.2 | 374.2 | 0.4 | 379.4 | 1395.4 | 0.9 | fail |
| 1M | 1.0% | 27.5 | 0.5 | 3.0 | 96.8 | 1.1 | 8.2 | 374.2 | 3.8 | 364.7 | 1404.4 | 4.0 | fail |
| 1M | 10.0% | 29.4 | 4.0 | 3.0 | 100.7 | 7.0 | 8.2 | 374.2 | 16.3 | 367.5 | 1416.9 | 46.4 | fail |
| 1M | 100.0% | 30.7 | 44.0 | 3.0 | 106.4 | 148.8 | 8.3 | 374.2 | 542.3 | 389.1 | 1449.4 | 2159.6 | fail |
| 10M | 0.1% | 290.3 | 1.1 | fail | 1003.2 | 2.0 | fail | stop | 3.2 | fail | stop | stop | fail |
| 10M | 1.0% | 294.8 | 6.7 | fail | 1006.4 | 12.8 | fail | stop | 26.6 | fail | stop | stop | fail |
| 10M | 10.0% | 315.0 | 50.8 | fail | 1052.3 | 104.7 | fail | stop | 195.3 | fail | stop | stop | fail |
| 10M | 100.0% | 323.5 | 450.6 | fail | 1058.3 | 1582.5 | fail | stop | stop | fail | stop | stop | fail |

#### 4.8.2.3 Comparing Γ Algorithm with Fast Matrix Multiplication: SciDB versus Intel MKL (LAPACK)

Here we study in more depth the computation of Γ. As mentioned above, R and many other mathematical packages use the LAPACK library [13, 14], which is both extremely fast and highly accurate. Therefore, it is natural to ask whether it is worth computing Γ with LAPACK (in RAM). Moreover, it is important to understand how LAPACK behaves with truly large matrices, reaching RAM capacity and much larger than L1/L2 cache memory. Parallel LAPACK variants are well tested and used by most systems with dense matrices. On the other hand, as of today, developing parallel programs with distributed memory for sparse matrices is an ongoing effort [9]. Here we compare our operator with dense matrix functions available in MKL for multi-core CPUs. Later, we compare with a sparse matrix multiplication available in SciDB that works in distributed memory in a parallel cluster (using ScaLAPACK [14]), which requires partitioning the input matrix and using MPI (Message Passing Interface).

Table 4.7 compares our Gamma operators with MKL, the fastest LAPACK parallel version for multicore CPUs. We emphasize MKL is working on RAM only, whereas our operator includes both I/O and CPU time. Therefore, we give MKL an advantage. As a major observation, MKL fails due to insufficient RAM at $n$=1M, $d$=800 and at $n$=10M with all $d$ values; there is no quick fix to solve this issue other than partitioning $X$ on secondary storage and developing a new algorithm.

Evidently, MKL is very efficient with small dense matrices and therefore, it is worthless optimizing our operators C++ code for small matrices. The main reason MKL is much faster is that it takes advantage of cache memory (L1 and L2) in the CPU, reading matrix blocks from RAM in the worst case. MKL incorporates sophisticated matrix multiplication algorithms that are very efficient when the matrix is small enough (relative to L1/L2 cache size). Also, MKL incorporates an efficient algorithm to move matrix blocks between the cache memory and the RAM repeatedly, a low-level optimization out of scope for us. Nevertheless, MKL is marginally faster than our dense matrix operator at $d$=400. These results prove our dense matrix operator is slower but scalable. For sparse matrices, the gap gets wider. At the lowest density extreme (density 0.1%) our sparse operator is much faster than MKL and our dense operator. In fact, for large matrices ($n$=1M), density 0.1% and $d$=400 our sparse operator is *three orders of magnitude* faster than MKL and the dense operator. For $d$=800, our sparse operator is three orders of magnitude faster than the dense operator. For $n$=10M, MKL fails with all $d$ values due to insufficient RAM, the dense operator must be stopped at $d = 400$, but our sparse algorithm can still efficiently work at $d$=400. Considering that the sparse algorithm is just 50% slower than the dense one with 100% density, it comes out as the overall winner.

Table 4.8: Parallel speedup: Γ dense matrix summarization operator with multi-threaded processing varying $M$=# SciDB instances ($N$=1 node, 4-core CPU); times in secs.

| | $n$=100k | | $n$=1M | | $n$=100M | |
|---|---|---|---|---|---|---|
| $M$ | time | speedup | time | speedup | time | speedup |
| 1 | 1.4 | 1.0 | 13.0 | 1.0 | 146.6 | 1.0 |
| **2** | 0.7 | 2.0 | 6.8 | 1.9 | 82.8 | 1.8 |
| 4 | 0.8 | 1.8 | 6.2 | 2.1 | 60.8 | 2.4 |

In summary, our dense and sparse matrix operators working together beat MKL in scalability and speed.

### 4.8.3 Parallel Processing (N = 1, 10, 100 nodes)

All speedup figures are computed as $s_N = T_1/T_N$ [14], where $T_1$ is the time for a purely sequential version and $T_N$ is the time on $N$ nodes; ideal speedup means $S_N \approx N$.

Table 4.8 presents speedup experiments varying the number of threads on our local server. The goal is to determine the optimal number of threads in the 4-core CPU (i.e. a scale-up analysis). As can be seen, as the number of threads grows time decreases, but the time gain shrinks. The optimal number of threads is 2 since those times are closest to the theoretical maximum speedup. Intuitively, 1 thread cannot interleave the intensive CPU computation for Γ with I/O to scan $X$ and at the other extreme, 4 threads incur too much overhead, switching thread context, and interleaving scan access to different chunks with one disk. Evidently, it is not worth going beyond the number of cores in the CPU, 4 in this case (results for 8 threads are much worse). In conclusion, the only alternative to achieve better speedup is to

Table 4.9: Parallel processing: Comparing R+SciDB and Spark varing $N$ (# of nodes); data set KDDnet $d = 38$; dense matrix; times in secs.

| | | R+SciDB | | Spark | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $N$ nodes | $n$ | Gamma/PCA/LR | speedup | Gramian | speedup | LR (20 iters) | speedup | PCA | speedup |
| 1 | 1M | 7.0 | 1.0 | 78 | 1.0 | 314 | 1.0 | 98 | 1.0 |
| 10 | 1M | 1.0 | 7.0 | 84 | 0.9 | 246 | 1.3 | 101 | 1.0 |
| 100 | 1M | 0.3 | 25.0 | 80 | 1.0 | 364 | 0.9 | 106 | 0.9 |
| 1 | 10M | 69.0 | 1.0 | 200 | 1.0 | 2524 | 1.0 | 352 | 1.0 |
| 10 | 10M | 6.8 | 10.1 | 94 | 2.1 | 376 | 6.7 | 120 | 2.9 |
| 100 | 10M | 0.7 | 98.6 | 98 | 2.0 | 414 | 6.1 | 120 | 2.9 |
| 1 | 100M | 610.0 | 1.0 | 1342 | 1.0 | stop | NA | 2932 | 1.0 |
| 10 | 100M | 56.2 | 10.9 | 210 | 6.4 | 2594 | NA | 432 | 6.8 |
| 100 | 100M | 5.7 | 107.8 | 224 | 6.0 | 920 | NA | 220 | 13.3 |

increase $N$ in the parallel cluster (i.e. scale out).

We conclude our study of parallel processing analyzing speedup and scalability in a large computer cluster in the Amazon cloud. Since $\Gamma$ is a summarization matrix whose computation is highly parallel it is natural to wonder if it works well on another parallel system, different from SciDB. The current trend in big data analytics is Apache Spark [50], which has proven to be easier to program, more scalable and more general than MapReduce. With such motivation, we studied how to compute the same machine learning models on Spark. Spark has a rich library of linear algebra and machine learning functions called MLlib, which offers matrix multiplication, PCA, and LR, but not VS. After trying several alternatives to compute $\Gamma$, we picked the Gramian() function because it was the most efficient, because it multiplies a matrix by itself, using $Z$ as input. We should stress that according to Spark documentation, Gramian() calls ScaLAPACK to evaluate the matrix multiplication in parallel in the most efficient manner. On the other hand, the PCA and LR models were directly available in MLlib passing the data set as the main input. The main difference is that PCA is solved by SVD like ours, but LR uses a gradient descent method [18, 24],

which today is the fastest method to compute machine learning models. That is, we compare two very different approaches to analyze big data: our summarization matrix versus gradient descent.

Before discussing results, we must explain installation, configuration, and tuning of both systems. We had to install SciDB as a collection of SciDB instances, where we assigned two instances (threads) to each node. One of the nodes served both as a coordinator and a worker. That is, this node was "overworked". Such limitation was an architecture constraint of SciDB since it is assumed that all nodes are uniform and the coordinator incurs on minimum overhead. On the other hand, Spark had the advantage of having a separate coordinator node, which had more RAM as shown in Table 4.3. Another important aspect was using sparse or dense matrices. Since Spark offered algorithms only with a dense (row) storage that is the one we used in SciDB; time differences would be even larger processing the KDDnet data set in sparse form. Finally, we used the parameter defaults in Spark for the LR and PCA models (20 iterations for LR, 5 components for PCA).

Table 4.9 compares SciDB and Spark running in the cloud varying $N$ (the number of nodes) and $n$ (data set size). The first major observation was that in R+SciDB the computation of $\Theta$ (the model) taking $\Gamma$ as input, took less than one second for LR and PCA. Therefore, we reported the three times together in one column for SciDB. From a raw time perspective, SciDB was two orders of magnitude faster than Spark to compute $\Gamma$. To compute models, the time difference is even more significant, close to three orders of magnitude, because the actual model computation took less than one second. In summary, R+SciDB exhibited optimal speedup and significantly better performance. From a speedup perspective, to compute $\Gamma$, Spark had good speedup

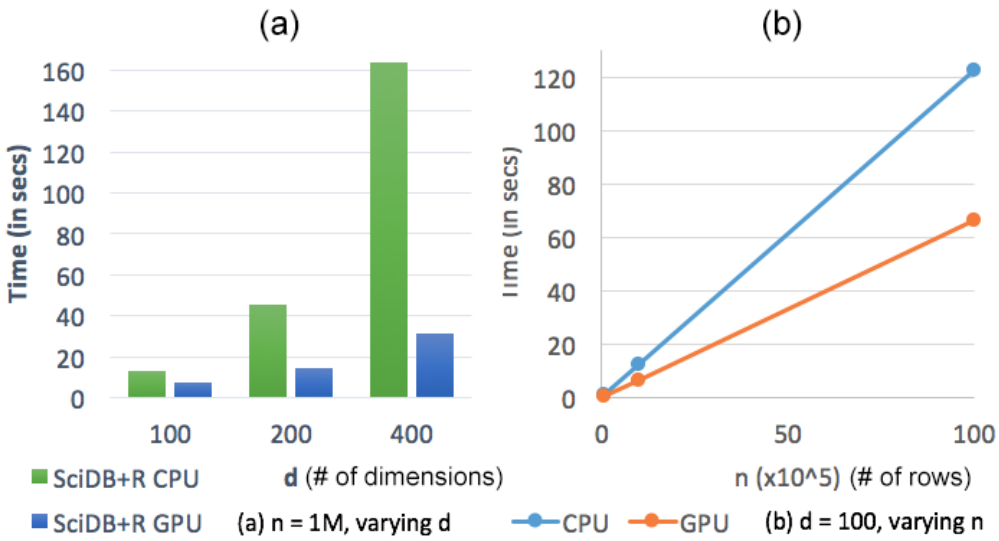



Figure 4.5: $\Gamma$ computation time: CPU vs. GPU.

at $N = 10$, but it was bad at $N = 100$ (explained by a slow matrix transposition). To compute LR, there was a better speedup up to $N = 100$, but far from ideal. Finally, for PCA, the results were the best, with decent speedup up to $N = 100$. On the other hand, SciDB exhibited an ideal speedup to compute $\Gamma$ both at $N = 10$ and $N = 100$. Notice, that due to SciDB architecture, when $N = 1$ a single node is overworked (coordinator+worker), which explains why speedup is better than linear when $N$ grows. Since the time to compute the LR and PCA models was negligible given the fast multicore CPU, the speedup was the same for $\Gamma$, LR and PCA.

### 4.8.4 Fast Data Summarization with a GPU

Figure 4.5 illustrates GPU impact on data summarization, which is the most time-consuming step in our 2-phase algorithm. Figure 4.5a shows the GPU has little

impact at low $d$ (we do not show times where $d < 100$ since the GPU has a marginal impact), but the gap between CPU and GPU rapidly widens as $d$ grows. On the other hand, Figure 4.5b shows the GPU has linear speedup as $n$ grows, an essential requirement given the I/O bottleneck to read $X$. Moreover, the acceleration w.r.t CPU remains constant.

#### 4.8.4.1 GPU Impact on Total Time to get Model

We now analyze GPU impact on the overall model computation, considering machine learning models require iterative algorithms. Table 4.10 shows the overall impact of the GPU. As can be seen, SciDB removes RAM limitations in the R runtime and it provides significant acceleration as $d$ grows. The GPU provides further acceleration despite the fact the dense matrix operator is already highly optimized C++ code. The trend indicates the GPU becomes more effective as $d$ grows. Acceleration is not optimal because there is overhead moving chunk data to contiguous memory space and transferring data to GPU memory and because the final sum phase needs to be synchronized. However, the higher $d$ is, the more FLOP work done by parallel GPU cores.

Table 4.10: Comparing computation of model $\Theta$ using R, DBMS+R, and DBMS+R+GPU; dense matrix operator; N=1 node (CPU=8 cores); times in secs.

| $n$ | $d$ | model | CPU | | GPU |
|---|---|---|---|---|---|
| | | | R | R+SciDB | R+SciDB |
| 1M | 100 | PCA | 29 | 14 | 8 |
| 1M | 200 | PCA | 90 | 46 | 16 |
| 1M | 400 | PCA | fail | 165 | 33 |
| 10M | 100 | PCA | fail | 147 | 104 |
| 10M | 200 | PCA | fail | 466 | 215 |
| 10M | 400 | PCA | fail | 1598 | 455 |
| 10M | 100 | LR | fail | 147 | 103 |
| 10M | 200 | LR | fail | 464 | 212 |
| 10M | 400 | LR | fail | 1594 | 451 |

# Chapter 5

# Conclusions

We proposed a generalized form of data cube, namely the percentage cube, that takes percentages as the fundamental aggregated measure. We introduced minimal SQL syntax extensions to compute percentage queries and to materialize the percentage cube. Specifically, we introduced the $pct()$ aggregate function and we considered alternative evaluation methods based on standard SQL (i.e. ensuring portability and wide applicability). We studied two alternative evaluation methods: OLAP functions and the GROUP-BY method using standard aggregate functions. We studied query optimizations for both methods, including efficient reuse of intermediate results, bypassing sorting in the query plan. We introduced pruning strategies extending previous techniques from iceberg queries. From a theoretical perspective, we showed percentages are in a higher complexity class, doubly exponential. We characterized percentages within the cube function hierarchy. We justified percentages are an algebraic function. On the other hand, selecting the top-$k$ percentages represents a holistic function. This was a big step beyond the standard cube. Fortunately, since percentage aggregations are algebraic, we showed that it was feasible to incrementally compute percentage queries and the percentage cube. Experimental results showed that our GROUP-BY method worked much faster than existing OLAP window functions in SQL. We also showed the direct and cascaded pruning strategies reduced evaluation time. We then showed pruning the cube lattice was essential. Filtering out low percentages added little time. We showed the additional time to get top $k$ percentages was significant, but still acceptable. Finally, the incremental

computation was effective for a percentage query, but not effective for the percentage cube, given the doubly exponential number of cuboids.

Since percentage cubes are a new concept, there are many research issues for future work. The percentage cube explores all possible grouping column combinations and the results have to be computed through joins. Due to the large amount of grouping column combinations, it was difficult to take advantage of the projections in the columnar DBMS to exploit merge joins (bypassing a sort phase in a sort-merge join). We expected better performance will be achieved by a tighter level integration with the DBMS exploiting aggregate UDFs [11]. We have shown selecting the top $k$ percentages in the cube represented the most demanding class of percentage aggregation, which offers many opportunities for optimization, especially reducing combinatorial and sort cost. Currently, we store the cube as a relational table, which is inefficient as cube dimensionality $d$ grows. A potential improvement is to exploit a non-tabular data structure, like a hash tree or FP-tree, but the caveat is that such data structures become incompatible with SQL tables. Therefore, it is necessary to explore a compromise between our purely relational solution and non-relational data structure solution. Incremental algorithms are fundamental in big data analytics since the number of records keeps growing and it is always preferable to reuse previous results. Our experimental results indicated incremental algorithms are good for individual percentage queries, but inefficient for the percentage cube. Therefore, a promising direction is to materialize the percentage cube on a carefully chosen set of dimensions.

We introduced Gamma, a comprehensive, but small, summarization matrix, which accelerated the computation of many machine learning models. We developed

parallel summarization algorithms based on a special form of matrix multiplication to efficiently compute Gamma. A key optimization was to evaluate matrix multiplication via a sum of vector outer products, instead of computing a product between the input matrix and its transpose. Based on the summarization matrix, algorithms to compute linear models were transformed to work in two phases: Summarization and iteration. Considering that dense and sparse matrices have different storage and require different processing in RAM, we introduced specialized summarization algorithms for dense and sparse matrices, respectively. We explained how to implement our algorithms in the SciDB array DBMS, to remove main memory limitations from R, and enable parallel processing. We presented an extensive experimental section. R was significantly accelerated by Gamma, and our algorithms working on SciDB eliminated R main memory limitations. Moreover, the array DBMS was an order of magnitude faster than a fast column-based DBMS, which evaluates matrix multiplication with SQL queries. We also compared $\Gamma$ with the fast matrix multiplication functions from LAPACK (MKL) which was the math library internally called by R. LAPACK (and MKL) failed when the input matrix did not fit in RAM, whereas our algorithms scaled beyond main memory limits. Our algorithms were slower than LAPACK with small dense input matrices, but became significantly faster with large sparse matrices. Further benchmarking compared the array DBMS with Spark on a large parallel cluster in the cloud showed our algorithms running on the array DBMS were two orders of magnitude faster than Spark calling functions from MLlib. From a parallel perspective, Spark did not have a good speedup to compute the summarization matrix (with a Gramian product via LAPACK), but had a good (almost linear) speedup to compute PCA and LR. On the other hand, our algorithms running on the array DBMS showed a linear speedup for both summarization and model

computation. Our experiments provide evidence that data summarization with fast matrix multiplication is a promising research direction. We need to study how other machine learning models, which require matrix multiplications, can exploit or generalize our Gamma summarization matrix. Large, high dimensional, data sets, are generally sparse matrices. Therefore, they deserve more attention than lower dimensional, dense, data sets. Based on current hardware trends, another reasonable assumption we made is that the partial and result matrices fit in main memory at each processing node. Nevertheless, an ideal summarization operator should work with input and output matrices of unlimited size. Clearly, our summarization matrix is promising to analyze big data, where we showed our array-based algorithms vastly outperform Spark, the dominating big data system. This big gap motivated integrate our summarization algorithms with Spark. We envision two alternatives: via a low-level transfer interface between the array DBMS and Spark, or programming them in Scala or Java.

# Bibliography


[1] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 5(6):914–925, 1993.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, 1993.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Databases*, pages 487–499, 1994.

[4] K. Akbudak and C. Aykanat. Simultaneous input and output matrix partitioning for outer-product-parallel sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing (SISC)*, 36(5):C568C590, 2014.

[5] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. D. Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, pages 2–11, 1990.

[6] P. Baumann, A. M. Dumitru, and V. Merticariu. The array database that is not a database: file based array query answering in rasdaman. In *Advances in Spatial and Temporal Databases*, pages 478–483. Springer, 2013.

[7] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg CUBE. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, volume 28, pages 359–370. ACM, 1999.

[8] P. Bradley, U. Fayyad, and C. Reina. Scaling clustering algorithms to large databases. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, pages 9–15, 1998.

[9] A. Buluç and J. Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing*, 34(4), 2012.

[10] Z. Chen and C. Ordonez. Efficient OLAP with UDFs. In *Proceedings of the ACM 11th International Workshop on Data Warehousing and OLAP*, pages 41–48, 2008.

[11] Z. Chen, C. Ordonez, and C. Garcia-Alvarado. Fast and dynamic OLAP exploration using UDFs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 1087–1090, 2009.

[12] J. Clear, D. Dunn, B. Harvey, M. Heytens, and P. Lohman. Non-stop SQL/MX primitives for knowledge discovery. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 425–429, 1999.

[13] J. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1st edition, 1997.

[14] J. Dongarra, I. Duff, D. Sorensen, and H. van der Vost. *Numerical Linear Algebra for High-Performance Computers*. SIAM, 1998.

[15] P. Drineas, M. Mahoney, and S. Muthukrishnan. Sampling algorithms for l2 regression and applications. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithm*, pages 1127–1136, 2006.

[16] W. DuMouchel, C. Volinski, T. Johnson, and D. Pregybon. Squashing flat files flatter. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 6–15, 1999.

[17] A. Gelman, J. Carlin, H. Stern, and D. Rubin. *Bayesian Data Analysis*. Chapman and Hall/CRC, 2003.

[18] R. Gemulla, E. Nijkamp, P. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 69–77, 2011.

[19] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab and sub-total. In *Proceedings of the 12th International Conference on Data Engineering*, pages 152–159, 1996.

[20] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, San Francisco, 2nd edition, 2006.

[21] J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2001.

[22] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, New York, 1st edition, 2001.

[23] J. Hellerstein, C. Re, F. Schoppmann, D. Wang, E. Fratkin, A. Gorajek, K. Ng, and C. Welton. The MADlib analytics library or MAD skills, the SQL. *Proceedings of the VLDB Endowment*, 5(12):1700–1711, 2012.

[24] S. Hoi, J. Wang, P. Zhao, R. Jin, and P. Wu. Fast bounded online gradient descent algorithms for scalable kernel-based online learning. In *Proceedings of the International Conference on Machine Learning*, 2012.

[25] ISO-ANSI. *Amendment 1: On-Line Analytical Processing, SQL/OLAP*. ANSI, 1999.

[26] V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas, and N. Koziris. An extended compression format for the optimization of sparse matrix-vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 24(10):1930–1940, 2013.

[27] F. Li and S. Nath. Scalable data summarization on big data. *Distributed and Parallel Databases*, 32(3):313–314, 2014.

[28] J. Marin and C. Robert. *Bayesian Core: A Practical Approach to Computational Bayesian Statistics*. Springer, 2007.

[29] A. Menon and C. Elkan. Fast algorithms for approximating the singular value decomposition. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 5(2):13, 2011.

[30] L. Muñoz, J.-N. Mazón, and J. Trujillo. Automatic generation of ETL processes from conceptual models. In *Proceedings of the ACM 12th International Workshop on Data Warehousing and OLAP*, DOLAP '09, pages 33–40, 2009.

[31] C. Ordonez. Vertical and horizontal percentage aggregations. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 866–871, 2004.

[32] C. Ordonez. Integrating K-means clustering with a relational DBMS using SQL. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 18(2):188–201, 2006.

[33] C. Ordonez. Models for association rules based on clustering and correlation. *Intelligent Data Analysis*, 13(2):337–358, 2009.

[34] C. Ordonez. Optimization of linear recursive queries in SQL. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(2):264–277, 2010.

[35] C. Ordonez. Statistical model computation with UDFs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(12):1752–1765, 2010.

[36] C. Ordonez and Z. Chen. Horizontal aggregations in SQL to prepare data sets for data mining analysis. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 24(4):678–691, 2012.

[37] C. Ordonez, Y. Zhang, and W. Cabrera. The Gamma operator for big data summarization on an array DBMS. *Journal of Machine Learning Research (JMLR): Workshop and Conference Proceedings (BigMine 2014)*, 36:61–96, 2014.

[38] C. Ordonez, Y. Zhang, and W. Cabrera. The Gamma matrix to summarize dense and sparse data sets for big data analytics. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 28(7):1906–1918, 2016.

[39] L. Oukid, O. Asfari, F. Bentayeb, N. Benblidia, and O. Boussaid. CXT-cube: Contextual text cube model and aggregation operator for text OLAP. In *Proceedings of the Sixteenth International Workshop on Data Warehousing and OLAP*, pages 27–32. ACM, 2013.

[40] L. Oukid, N. Benblidia, F. Bentayeb, and O. Boussaid. TLabel: A new OLAP aggregation operator in text cubes. *International Journal of Data Warehousing and Mining*, 12(4):54–74, 2016.

[41] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Databases*, pages 553–564, 2005.

[42] M. Stonebraker, D. Abadi, D. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.

[43] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science and Engineering*, 15(3):54–62, 2013.

[44] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very large Databases*, pages 1150–1160, 2007.

[45] M. Stonebraker, L. Rowe, and M. Hirohama. The implementation of Postgres. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, 1990.

[46] A. Witkowski, S. Bellamkonda, T. Bozkaya, G. Dorman, N. Folkert, A. Gupta, L. Sheng, and S. Subramanian. Spreadsheets in RDBMS for OLAP. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 52–63, 2003.

[47] R. Xin, J. Rosen, M. Zaharia, M. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 13–24, 2013.

[48] H. Xiong, S. Shekhar, P. Tan, and V. Kumar. TAPER: A two-step approach for all-strong-pairs correlation query in large databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 18(4):493–508, 2006.

[49] A. Yzelman and D. Roose. High-level strategies for parallel shared-memory sparse matrix-vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):116–125, 2014.

[50] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, pages 10–10, 2010.

[51] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An efficient data clustering method for very large databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 103–114, 1996.

[52] Y. Zhang, C. Ordonez, and W. Cabrera. Big data analytics integrating a parallel columnar DBMS and the R language. In *Proc. of IEEE CCGrid Conference*, 2016.

[53] Y. Zhang, C. Ordonez, J. García-García, and L. Bellatreche. Optimization of percentage cube queries. In *Proceedings of the 20th International Conference on Extending Database Technology (EDBT)*, 2017.

[54] Y. Zhang, C. Ordonez, J. Garcia-Garcia, L. Bellatreche, and H. Carrillo. The percentage cube. *Information Systems*, 2017. conditionally accepted.

[55] Y. Zhao, P. Deshpande, and J. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, volume 26, pages 159–170. ACM, 1997.