"OPTIMIZED ALGORITHMS FOR DATA ANALYSIS 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

Wellington Cabrera

May 2017

"OPTIMIZED ALGORITHMS FOR DATA ANALYSIS IN PARALLEL DATABASE SYSTEMS"

Wellington Cabrera

APPROVED:

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

Edgar Gabriel, Ph.D.
Dept. of Computer Science

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

Zhu Han, Ph.D.
Cullen College of Engineering

Dean, College of Natural Sciences and Mathematics

Acknowledgements

First of all, I thanks the Almighty God for His abundant mercy, for the gift of life, health, and strength. I thank my advisor, Dr. Carlos Ordonez, for his valuable guidance and support through my academic life in University of Houston. I would like to thank the committee members, Dr. Omprakash Gnawali, Dr. Edgar Gabriel and Dr. Zhu Han, for sharing their knowledge and experience. Also, I appreciate the feedback and technical support from my research group partners, Yiqun Zhang and Dr. David Matusevich. Thanks to my wife, Johanna, for her loving care. Finally, my appreciation to my children and my parents for their emotional support and permanent encouragement.

"OPTIMIZED ALGORITHMS FOR DATA ANALYSIS 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

Wellington Cabrera

May 2017

Abstract

Large data sets are generally stored on disk following an organization as rows, columns or arrays, with row storage being the most common. On the other hand, matrix multiplication is frequently found in machine-learning algorithms as an important primitive operation. Since database management systems do not support matrix operations, analytical tasks are commonly performed outside the database system, in external libraries or mathematical tools. In this work, we optimize several analytic algorithms that benefit from a fast in-database matrix multiplication. Specifically, we study how to compute in-database parallel matrix multiplication to solve two major family of big data analytics problems: machine-learning models and graph algorithms. We focus on three cases: the product of a matrix by its transposed, the powers of a square matrix and iteration of matrix-vector multiplication. Based on this foundation, we introduce important optimizations to the computation of fundamental linear models in machine-learning: linear regression, variable selection and principal components analysis. On the other hand, we present parallel graph algorithms that take advantage of matrix powers and parallel vector multiplication to solve several graph problems: transitive closure, all pairs shortest paths, reachability from a single source vertex, single source shortest paths, connected components and PageRank.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Nowadays the world has become more interconnected than before, with billions of devices generating some kind of data. Consequently, data is generated, gathered, and stored in a faster pace. The analysis of large amounts of data, generally stored in parallel clusters, becomes critical for corporations and government.

Database Management Systems (DBMS) remain the most common technology to store transactions and analytical data, due to optimized I/O, robustness and security control. Even though, the common understanding is that DBMSs cannot handle demanding analytics problems, because relational queries are not sufficient to express important algorithms, and scarce support of matrix operations. Therefore, DBMS are considered cumbersome and too slow for complex analytical tasks,

such as supervised/unsupervised learning and graph analytics. Consequently, most of the analytical tasks become a complex data flow between the DBMS and external analytics systems, especially popular tools on the Hadoop-Mapreduce platform. Such complex data flows have several drawbacks: Time consuming exporting processes, security risks, handling additional systems and tools, and additional hardware resources.

In this work we concentrate on devising algorithms for data sets stored in parallel database systems, working on clusters under a shared-nothing architecture. We face several challenges: limited programmability for complex analytics, specially Machine Learning and Graph Algorithms; lack of support for matrix operations in the relational model; inflexibility in some systems due to the enforcement of a data model.

1.2 Contribution Overview

Our contribution targets two families of algorithms: Computation of linear models and graphs algorithms. The common foundation of our algorithms is the optimized multiplication of matrices stored in a distributed manner through the cluster. Our contributions are enumerated in a more detailed way below:

1. We show how to accelerate the computation of linear models, exploiting fast parallel computation of a special matrix product.

2. We present optimized queries to compute matrix-matrix multiplication for matrices stored in parallel DBMSs, and an application for graph algorithms: All pair shortest paths, triangle counting and transitive closure.

3. We propose a unified algorithm for parallel DBMSs to compute reachability from a source vertex, single source shortest path, connected components and PageRank.

On the other hand, our research generates technical benefits to the data analytics community: (1) in-database analytics avoids complex data flows between diverse systems and applications, alleviating the burden of transferring data and programming scripts for multiple systems; (2) since DBMSs possess mature data security and recovery features, security and privacy risks are minimized; (3) because our parallel algorithms are in general several times faster than current solutions, the process can be executed in a cluster of smaller size, which generates financial savings by using less energy and hardware resources.

1.3 Dissertation Organization

The organization of this dissertation is presented below:

Chapter 2 presents a literature review about linear models and graph analytics computation in parallel data systems. In Chapter 3 we present notation, definitions and background information about linear models and graph analytics. We discuss the representation of data sets in database systems. Chapter 4 explains details about

the computation of a summarization matrix, and how it is exploited to accelerate linear regression, variable selection, and Principal Component Analysis. Chapter 5 is devoted to the computation of powers of a square matrix in parallel database systems; furthermore, we present an application to solve three graph problems: transitive closure computation, all pairs shortest paths, and triangle counting. In Chapter 6 we explain how to optimize the computation of iterative matrix-vector multiplication, and we apply it to four graphs problems: reachability from a source vertex, single source shortest path, connected components and PageRank. Finally, in Chapter 7 we present the conclusions of our work as well as research issues for future work.

Chapter 2

Related Work

2.1 Analytics in Parallel Systems

The MADlib Analytics library [19] is a comprehensive suite of statistical and machine learning methods, that runs on top of a DBMS. Matrix multiplication is one important primitive of MADlib, solved with call to specialized linear algebra libraries (i.e., LAPACK). Although MADLib supports linear regression, it does not address the problem of variable selection. SciDB [46] is a parallel array-storage system providing some analytical functions including Singular Value Decomposition (SVD), quartiles, average and standard deviation, as well as matrix multiplication. Like MADLib, SciDB's linear algebra functions calls LAPACK. On the Hadoop platform, Mahout is a set of Java libraries of primitives for statistics and Machine Learning, to facilitate development of analytics algorithms under the MapReduce model. Apache Spark is another system built on top of Hadoop DFS. Spark[49] is an in-memory system,

designed to overcome MapReduce limitations running interactive loads. Specialized libraries for machine learning (MLlib [33]) and graph algorithms (GraphX [16]) have contributed to the increased popularity of Spark.

2.2 Graph Algorithms in Parallel Systems

In recent years the problem of solving graph algorithms in parallel DBMS with relational queries has received limited attention. Recently, the authors of [24] studied Vertica as a platform for graph analytics, focusing in the conversion of vertex-center programs to relational queries, and in the implementation of shared-memory graph processing via User Defined Functions (UDFs), in one node. [13] describe the enhancements to SAP HANA to support graph analytics, in a columnar in-memory database. In [48], the authors show that their SQL implementation of shortest path algorithm has better performance than Neoj4. Note that the later work runs in one node with a large RAM (1 TB). In our previous work [36], we proposed optimized recursive queries to solve two important graph problems with SPJA queries: Transitive closure and All Pairs Shortest Path. This recent work also shows that columnar DBMS technology performs much better than array or row DBMSs. Running on top of Hadoop, Pegasus is a graph system based on matrix multiplications; the authors propose grouping the cells of the matrix in blocks, to increase performance in a large-RAM cluster. Our work differentiates of the previously described in several ways: 1) we present a unified framework for compute graph algorithms with relational queries; 2) we present optimizations for columnar and array DBMS based in a careful data

distribution; 3) the out-of-core graph computation allows us to analyze graphs with hundreds millions edges with minimum RAM requirements.

Chapter 3

Background and Definitions

3.1 Parallel DBMSs under Shared-nothing Architecture

The algorithms in this work are conceived for parallel DBMSs under a shared-nothing architecture. Although our optimized algorithms can work in any DBMS, in our previous work [36] we showed experimentally that columnar and array DBMSs present performance substantially better than row DBMSs for graphs analysis. For this reason in the current study we concentrate on columnar and array DBMSs. These systems are designed for fast query processing, rather than transaction processing. In this work, we consider parallel DBMSs under a sharing-noting architecture: a cluster of N nodes, each one with its own storage and memory.

Row DBMS

The pioneer parallel database management systems stored data by records, whose fields are stored in contiguous space. This systems were aimed to exploit the I/O bandwidth of multiple disks [12], improving in this way reading and writing performance, and allowing the storage of data too big to fit in only one machine. Large tables are to be partitioned through the parallel system. Three common methods of partitioning are: 1) splitting the tables to the nodes by ranges with respect to an attribute's value; 2) distributing records to the nodes in a round-robin assignment; 3) using a hash function to assign records to the nodes. Currently, the last method is the most commonly used.

Columnar DBMSs

Columnar DBMSs emerged in the previous decade presenting outstanding performance for OLAP. C-Store [44] and Monet-DB [21] are among the first systems that have exploited the columnar storage. Columnar DBMSs can evaluate queries faster than traditional row-oriented DBMSs, specially queries with join or aggregation operations. While row DBMSs generally store data in blocks containing a set of records, columnar DBMSs store columns in separate files, as large blocks of contiguous data. Storing data by column benefit the use of compression. Due to the low entropy of the data in a column, low-overhead data compression has been exploited for further performance improvements. This data compression does not hinder parallelism.

Array DBMSs

Array store is a technology aimed to provide efficient storage for array-shaped data. Most of the array DBMSs support vectors, bi-dimensional arrays and even multi-dimensional arrays. Array stores organize data by data blocks called chunks [43] [45], distributed across the cluster. In bi-dimensional arrays, chunks are square or rectangular blocks. The chunk map is a main memory data structure that keeps the disk addresses of every chunk. Each cell of the array has a predefined position in the chunk, just as regular arrays are stored in main memory. Because the access to the data is relative to the initial address of the chunk, user-defined indexes are not necessary. Parallel array DBMSs distribute data through the cluster's disk storage on a chunk basis.

3.2 Data Set as a Matrix

We first define X, the input matrix (data set obtained after preprocessing [38]). Let $X = \{x_1, ..., x_n\}$, where x_i is a vector in $\mathbb{R}^d$. In other words, X is a $d \times n$ matrix. Notice that for notational convenience X is defined as a large set of column vectors. Intuitively, X can be pictured as a wide rectangular matrix. Supervised (predictive) models require an extra attribute [5]. For regression models X is augmented with a $(d+1)$th dimension containing an output variable Y. For classification models, there is an extra discrete attribute C, where is C is generally binary (e.g., false/true, bad/good). We use $i = 1...n$ and $j = 1...d$ as matrix subscripts. We use Θ, a set of vectors, matrices and associated statistics, to refer to a machine learning model in a

generic manner. Thus, Θ can represent models such as principal component analysis (PCA), linear regression (LR), among others.

3.3 Graph Data Set

Let $G = (V, E)$ be a directed graph, where V is a set of vertices and E is a set of edges, considered as an ordered pairs of vertices. Let $n = |V|$ vertices and $m = |E|$ edges. The adjacency matrix of G is a $n \times n$ matrix such that the cell i, j holds a 1 when exists an edge from vertex i to vertex j. In order to simplify notation, we denote as E the adjacency matrix of G. The *outdegree* of a vertex v is the number of outgoing edges of v and the *indegree* of v is the number of incoming edges of v. The algorithms in this work use a vector S to store the output and intermediate results. The ith entry of S is a value corresponding to vertex i, and it is denoted as $S[i]$.

Chapter 4

Linear Models with Optimized Parallel Matrix Multiplication

4.1 Summarization Matrix for Linear Models

A statistic is a function of a random sample of a population. Assuming a population described by a specific distribution with parameter Θ, a sufficient statistic as good as the complete sample to compute the model Θ. Under a Gaussian distribution, a fundamental set of sufficient statistics is:

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

$$Q = XX^T = \sum_{i=1}^{n} x_i x_i^T \tag{2}$$

$$n = |X| \tag{3}$$

As we will explain in detail in section 4.5 , the linear model uses the *augmented* X matrix, represented as $\mathbf{X}$. We denote $\mathbf{Q}$ as equivalent to the matrix product $\mathbf{X}\mathbf{X}^T$. In [40], we introduced a more general augmented matrix $\mathbf{Z}$, created by appending to $\mathbf{X}$ one additional row, which contains the vector Y. Because $\mathbf{X}$ is $(d+1) \times n$, $\mathbf{Z}$ contains $(d+2)$ rows: The $(d+1)$ rows of $\mathbf{X}$, plus the additional row of Y. Thus:

$$\Gamma = \mathbf{Z}\mathbf{Z}^T \tag{4}$$

contains a summary of the whole data set, sufficient to compute the linear model.

$$\begin{aligned}
\Gamma &= \begin{bmatrix} n & \sum x_i^T & \sum y_i \\ \sum x_i & \sum x_i x_i^T & \sum x_i y_i \\ \sum y_i & \sum y_i x_i^T & \sum y_i^2 \end{bmatrix} \\
&= \begin{bmatrix} n & L^T & \sum y_i \\ L & Q & XY^T \\ \sum y_i & YX^T & YY^T \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{Q} & \mathbf{X}Y^T \\ Y\mathbf{X}^T & YY^T \end{bmatrix}
\end{aligned} \tag{5}$$

In Big Data problems, the number of data point is much larger than the number of dimensions. Under this conditions, the $(d+2) \times (d+2)$ matrix Γ contains a very compact set of quantities, useful for computing linear regression, variable selection and PCA. Since the quantities on Γ are summations, this matrix computed in parallel.

4.2 Parallel Computation in Multi-core CPUs, Single Node

4.2.1 Aggregate UDF in row DBMS for $n >> d$

The parallel computation of sufficient statistics with multiple CPUs in a single node was presented in a previous research work of Ordonez et. al [39]. The parallel computation is implemented by aggregated User Defined Functions (UDFs). The parallel computation of the summarization matrix occurs in four steps:

1. Initialize: In this step and variables and data structures for the aggregation get their initial value.

2. Accumulate: This is the key step in parallel processing. Separate threads computes partial versions of the summarization matrix.

3. Merge: This step merges the partial results computed in the previous step by independent thread. Thus, the global aggregation is computed in this step.

4. Terminate: This step returns the final aggregation result to the user, after the

partial results have been merged.

When the data set has a moderated dimensionality (i.e., $d < 1000$) the data set can be stored in a relational table in the usual manner: rows represent data points and columns dimensions. Of course, the number of dimensions can not exceed the maximum number of columns allowed by the DBMS.

4.2.2 Aggregate UDF in row DBMS for $d >> n$

When $d >> n$, the number of dimensions might exceed the maximum number of columns (MaxCol) per a DBMS table [8]. In case of $n <$ MaxCol , we propose storing the data set in a transposed format: columns store data points and rows store dimensions. On the other hand, the computed matrix Γ may fit or not in main memory. This situation makes a substantial difference for the programming, since a large Γ must be stored in disk.

4.2.3 Computation in Array DBMS

While n and L can be efficiently computed in SciDB in a trivial manner, a straightforward computation of Γ with SciDB built-in operators may lead to inefficient queries, as we will show in our experiments. The straightforward (or intuitive) computation of Γ is based on calculating it by using matrix multiplication and transposition, just like its mathematical definition (see equation 4). Thus, two straightforward solutions based on SciDB custom operators are:

- use gemm() and transpose() built-in operators to compute Γ
- use crossprod() function, available in SciDB-R to compute Γ

We present below the sequential algorithm to compute Γ.

Data: $X = \{x1_1, ...x_n\}, Y = \{y1_1, ...y_n\}$
Result: Γ

```
Z ← [1, X, Z];
Γ ← 0 ;
for i = 1...n do
    for a = 0...d + 1 do
        for b = 0...d + 1 do
            Γ_ab ← Γ_ab + z_ia * z_ab
        end
    end
end
```

Algorithm 1: Gamma Computation

4.2.4 Computation with LAPACK

LAPACK is a high performance library for linear algebra operations, including matrix-matrix and matrix-vector multiplication. LAPACK performs matrix multiplication in main memory only, contrasting to our proposed algorithm which does not need to upload the complete data set in RAM. Notice our contribution does not rely in matrix multiplication with this library, but we run experiments computing Γ with LAPACK in order to understand the performance differences with our algorithm. In the experimental section we will present a performance comparison between computing Γ calling LAPACK and using our proposed method.

4.3 Parallel Computation in Multiple Nodes

4.3.1 Computation with ScaLAPACK in Array DBMS

ScaLAPACK [10] (Scalable Linear Algebra Package) is a library that redesigns several LAPACK functions to work in a cluster of distributed memory. ScaLAPACK requires properly partitioned matrices, and relies on MPI (Message Passing Interface). ScaLAPACK is widely used in numerical computations in HPC settings. SciDB has incorporated ScaLAPACK to solve Linear Algebra computations, including SVD and matrix multiplications. Since the summarization matrix Γ is defined as a matrix multiplication, we conduct experiments to compare Γ versus ScaLAPACK.

4.3.2 Computation with Custom Operator in Array DBMS

In [40], we have explained the computation of Γ in an array DBMS with multiple computing nodes. Exploiting parallelism in multiple nodes is a requirement that complicates programming, because we need to ensure that X can be evenly partitioned across N processing nodes, with minimal or no need of synchronization overhead [40]. The advantage is, of course, the ability to scale processing to more nodes as the data set size grows. Our Γ operator works fully in parallel with a partition of X into N database subsets $D_1 \cup D_2 \cup ... D_N = X$, where we compute Γ_I on D_I for each node I. When every node is done with its portion of the computation, all the partial Γ_I are concentrated in the coordinator nodes, which computed the global $\Gamma = \Gamma_1 + .. + \Gamma_N$. This sequential algorithm presented in section 4.2.3 can be easily generalized to a

parallel one, due to the additive properties of Γ

4.4 Principal Components Analysis (PCA)

Principal Components analysis is a well known analytical technique, applied in many domains as neuroscience, computer vision, semantic analysis of large databases, etc. The objective of PCA is to reduce the noise and redundancy by re-expressing the data set in a new orthogonal basis, which is a linear combination of the original basis [18]. The Principal Components are a set of vectors, ordered by their associates variances in decreasing order. Principal Components are efficiently calculated by computing SVD in the correlation matrix. Given a matrix M, SVD is a factorization expressed as $M = U\Sigma V^T$

4.5 Linear Regression

Let $X = \{x_1, \ldots, x_n\}$ be a set of n observations with d explanatory variables and $Y = \{y_1, \ldots, y_n\}$ be a set of numbers such that each y_i represents an outcome of x_i. Using the matrix notation to represent the data, Y ($[1 \times n]$) and X ($[d \times n]$), the linear regression model can be expressed as:

$$Y = \beta^T \boldsymbol{X} + \epsilon \tag{6}$$

where $\beta = [\beta_0, \ldots, \beta_d]$ is the vector of regression coefficients; ϵ represents the error, with a Gaussian probability distribution and $\boldsymbol{X}$ is X augmented with a row of n 1s stored in an extra dimension X_0. The canonical regression setup [15] assumes that Y has a normal distribution, as follows:

$$Y|\beta, \sigma^2, X \sim N_n(X\beta, \sigma^2 I_n) \tag{7}$$

The vector β is usually estimated using the Ordinary Least Squares method [42]. Table 4.1 summarizes the notation of the linear regression model.

Table 4.1: The Linear Regression Model.

Matrix	Dimensions	Name
Y	$1 \times n$	Dependent Variable
X	$d \times n$	Independent Variable
$\boldsymbol{X}$	$(d+1) \times n$	Augmented Independent Variable
β	$(d+1) \times 1$	Regression Coefficients

4.6 Bayesian Variable Selection

Variable selection is the search for the best subsets of variables that are good predictors of Y [34]). The assumption of this search is that the data contains features that are redundant and can be safely excluded from the model, since they provide little relevant information. There are many reasons to undertake this search:

- To express the relationship between the dependent variables and the explanatory variables in the simplest way possible [34]

- To identify the which variables are important and which are negligible predictors [34]
- To facilitate data visualization and interpretation [17]

Finding which subsets are the most appropriate is not computationally trivial, since there are 2^d potential combinations of variables. Evidently, this problem cannot be solved using a brute force method when the number of dimensions is high. Traditional greedy methods (stepwise methods, for instance [20]) focus on finding the best subset of variables, but the probabilistic Bayesian approach aims to identify several promising subsets of variables.

The set of selected variables can 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 M_γ the model that selects k of the d variables, corresponding to the vector γ. Given γ we can easily find how many variables were selected by performing the dot product $k = \gamma^T \cdot \gamma$. Throughout this thesis we will use γ as an index on selected variables such as β_γ and $\boldsymbol{X}_\gamma$.

Under the Bayesian approach, the unknown parameters of the model $\theta = \{\beta, \sigma, \gamma\}$ are considered random variables [37]. The goal under the Bayesian approach is to find the models with higher probabilities, instead of a single model.

4.6.1 SSVS

Stochastic Search Variable Selection (SSVS) is a classic Bayesian method for variable selection [15]. SSVS uses the Gibbs sampler, a Monte Carlo method, to find the Markov chain $\beta^0, \sigma^0, \gamma^0, \beta^1, \sigma^1, \gamma^1 ...$

Each one of these values is obtained by sampling, as follows:

$$\beta^j \propto f(\beta^j | Y, \sigma^{j-1}, \gamma^{j-1}) \tag{8}$$

$$\sigma^j \propto f(\sigma^j | Y, \beta^{j-1}, \gamma^{j-1}) \tag{9}$$

$$\gamma_i^j \propto f((\gamma_i)^j | Y, \beta^j, \gamma_{(i)}{}^j) \tag{10}$$

These expressions show that β and σ are required in order to compute γ. Moreover, the computation of σ requires the complete data set (Y and X). For such reason, we could not optimize SSVS, because we would need to scan the complete data set in each iteration. Instead, we consider Gibbs sampler with a different prior formulation, as we explain in the next subsections.

4.6.2 Zellner G-prior

The Zellner G -prior is a well know prior specification for linear models [31], which is a member of the family of conjugate priors. Zellner proposed a particular form of the Normal-Gamma family, with mean β and a constant c multiplying the variance.

$$\beta | \sigma^2, X \sim \mathcal{N}_{k+1} \left(\tilde{\beta}, c\sigma^2 \left(XX^T \right)^{-1} \right) \tag{11}$$

$$\sigma^2 \sim \pi \left(\sigma^2 | X \right) \propto \sigma^{-2} \tag{12}$$

A important advantage of the Zellner G-prior is that enables analytical margining out of β and σ from $\pi(\beta, \sigma, \gamma|Y)$. Thus, the computation of β and σ is not required, and the Markov chain becomes simpler: $\gamma^0, \gamma^1, \gamma^2$...

4.6.3 Bayesian Variable Selection with Gibbs Sampler and the Zellner G-prior

Since the search space of this computational problem is exponential (2^d subsets of variables), an exhaustive search is impractical for even a moderately large d. Though there are many techniques available in the literature [20] to deal with such a problem, a Bayesian approach provides information about the prior probabilities as an added bonus. Because it is very difficult to produce a solution to such a problem, several authors [15, 34] exploit Markov chain Monte Carlo techniques, such as the Metropolis-Hastings Method and the Gibbs sampler, to obtain accurate results and to sample from the posterior distributions. The Gibbs sampler is a Markov chain Monte Carlo method to obtain a sequence of observations approximated from the posterior probability. In the case of Variable Selection, this sequence of observations is characterized by a vector $\gamma^{[i]}$, that describes the variables selected at iteration i of the sequence [15] $\gamma^{[i]}$ is obtained from the previous vector, $\gamma^{[i-1]}$, as follows: For every variable x_j the normalized probabilities of $p(\gamma_j^{[i]} = 0)$ 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 position of vector $\gamma^{[i]}$ is updated. After N iterations we obtain the Markov chain sequence $\gamma^{[0]}, \ldots, \gamma^{[N]}$. To avoid introducing a bias into the prior computation

due to initial instabilities, the first B iterations of the Markov chain are usually discarded. This is called *the burn-in* period. After the burn-in, it is assumed that we have reached a stable distribution (the process becomes ergodic) and we can safely sample the priors.

4.6.4 Three-step algorithm

Very-high-dimensional problems are hard and very demanding in computing resources. To reduce the computational burden, we proposed in [32] a Bayesian Variable Selection method of three steps:

1. Pre-selection: This process performs an initial screening of variables using the correlation ranking method [17, 22]
2. Summarization: Compute the summarization matrix Γ
3. Iterative Computation: Compute Gibbs sampler with the Zellner prior, exploiting Γ.

4.7 Experimental Evaluation

We conduct experiments with the network intrusion data set from the KDD99 Cup, to evaluate the execution performance of our algorithms. To understand time complexity, we replicate the data set to increase the input size. In Table 4.2, we compare standard statistical functions from the R libraries versus our optimized algorithms,

Table 4.2: Comparing the computation of the model Θ using R and DBMS+R; data set KDD; local server; time in seconds

d	n	PCA		LR		VS	
		R	R+DBMS	R	R+DBMS	R	R+DBMS
10	100k	<1	<1	<1	<1	3	4
10	1M	5	1	6	1	8	5
10	10M	45	7	50	7	593	10
10	100M	fail	65	fail	70	fail	93
100	100k	6	3	6	3	34	15
100	1M	61	17	61	17	113	29
100	10M	fail	195	fail	195	fail	207

Table 4.3: Bayesian variable selection (VS) in gene data set ($d = 12506, n = 248$): Comparing R+SciDB with R at 1000 iterations; time in secs.

d	R+SciDB	R unoptimized	R optimized with Γ
100	17	1139	16
200	43	*	43
400	83	*	85
800	199	*	20

which compute the summarization matrix Γ in the parallel DBMS and the actual model in R. Only in the case of the smallest data set the execution time of plain R is comparable to R exploiting Γ. In general, our optimized algorithms are one or even two orders of magnitude faster. Moreover, plain R fails when the data set does not fit in RAM, whereas our algorithms keep working with promising performance.

Recall that Γ is by definition a matrix multiplication: $Z \times Z^T$. We run experiments to compare the performance of the Gamma operator versus the matrix multiplication of Math Kernel Library (MKL), an implementation of LAPACK for Intel microprocessors. Notice the time for the Gamma operator includes reading the data set from disk, but the times for MKL are measured when the data is already

Table 4.4: Comparing dense Gamma Operator vs MKL BLAS (parallel LAPACK) with synthetic data sets; time in secs.

n	d	Gamma Operator	MKL BLAS
1M	100	31	3
1M	200	107	8
1M	400	374	389
1M	800	1449	2160
10M	100	323	fail
10M	200	1058	fail
10M	400	stop	fail
10M	400	stop	fail

Table 4.5: Comparing SciDB programming mechanisms; data set KDDnet $d = 100$; 1 node; time in secs.

n	SCALAPACK	SciDB-R built-in	Gamma Operator
10k	3.1	19.1	0.3
100k	15.3	76.8	1.3
1M	177.8	stop	13.7
10M	fail	stop	150.2

loaded in RAM. Thus, we give to MKL an advantage. The results are shown in table 4.4. While MKL is faster for lower values of n and d, the Gamma operator is faster when $d \geq 400$. Finally, while MKL fails when the matrices do not fit in RAM, the Gamma operator is able to finish the computation.

We run additional experiments to compare the performance of the Gamma operator versus SCALAPACK, and a built-in operator available in SciDB-R. The Gamma operator shows a clear advantage: One order of magnitude over the closer competitor, matrix multiplication with SCALAPACK.

Chapter 5

Graph Analytics with Parallel Matrix-Matrix Multiplication

5.1 Matrix Powers with Linear Recursion

Matrix Powers can be computed elegantly using recursive queries. Legacy row-store database systems generally provide recursive queries, as in the case of Postgres, Oracle, Teradata and MS SQL Server. In contrast, recursive queries are not supported in array or columnar DBMS, with SAP Hana as the exception. Our contribution consists on presenting optimized recursive queries in columnar and array DBMS, with an application on graph algorithms. The standard ANSI SQL defines a syntax to create recursive views using RECURSIVE VIEW. A recursive view has one or more non-recursive SELECT statements, and one or more SELECT statements with recursive references. The non-recursive part is known as the "base step", and the

statement with a recursive reference is the "recursive step". Although the steps may appear in any order, we show firstly the base step, for the sake of clarity.

We study queries of the form: $R_d = R_d \bowtie E$, where the most common join predicate is equality in equi-join $R_d.j = E.i$ (finding connected vertices). Within linear recursive queries the most well-known problem is computing the transitive closure of G, which accounts for most practical problems. As noted above, transitive closure is deeply related to matrix multiplication. Linear recursion is specified by a join operator in a recursive select statement, where the declared view name appears once in the FROM clause [36]. In general, the recursive join condition can be any comparison expression, but we focus on equality (i.e., equi-join). To avoid long runs with large tables, infinite recursion with cyclic graphs or infinite recursion with an incorrectly written query, it is advisable to add a WHERE clause to set a threshold on recursion depth (k, a constant).

We define queries for the two problems introduced in Section 2. We start by defining the following recursive view R, which expresses the basic recursion to join E with itself multiple times. We emphasize R appears once in the FROM clause obeying a linear recursion

5.2 Recursive Query processing

We start by reviewing SQL queries for the standard algorithm to evaluate recursive queries with SQL: Seminaive. Such SQL queries do not depend on any specific storage mechanism or database system architecture. After understanding these basic aspects,

we revisit optimization of recursive queries. We focus on contrasting how recursive queries are optimized in a columnar DBMS compared to row and array DBMSs. We study the optimization of SPJ queries involving selection, projection and join operators where projection includes duplicate elimination and group-by aggregations as two particular cases. For each operator we first present its optimization from an algebraic perspective and then we discuss how the operator is evaluated considering each different DBMS architecture.

5.2.1 Seminaïve Algorithm

In order to make this chapter self-contained we review Seminaïve, using as input the graph G defined in Section 3.3 The standard and most widely used algorithm to evaluate a recursive query comes from deductive databases and it is called Seminaïve [3, 4]. The Seminaïve algorithm solves a general class of mathematical logic problems called fixpoint equations [3, 2]. Let R_k represent a partial output table obtained from $k-1$ self-joins with E as operand k times, up to a given maximum recursion depth k:

$$R_k = E \bowtie E \bowtie \cdots \bowtie E,$$

where slightly abusing notation each join uses $E.j = E.i$ (i.e., in SQL each table has an alias $E_1, E_2, \ldots, E_k$). The base step produces $R_1 = E$ and the recursive steps produce $R_2 = E \bowtie E = R_1 \bowtie_{R_1.j=E.i} E$, $R_3 = E \bowtie E \bowtie E = R_2 \bowtie_{R_2.j=E.i} E$, and so on. Notice that the general form of the recursive join is $R_{d+1} = R_d \bowtie_{R_d.j=E.i} E$, where

the join condition $R_d.j = E.i$ links a source vertex with a destination vertex if there are two edges connected by an intermediate vertex. Notice that at each recursive step a projection (π) is required to make the k partial tables union-compatible. Assuming graphs as input, π computes $d = d + 1$, $i = R_d.i$, $j = E.j$, $p = R.p * E.p$ and $v = R_d.v + E.v$ at each iteration:

$$R_{d+1} = \pi_{d,i,j,p,v}(R_d \bowtie_{R_d.j=E.i} E). \tag{13}$$

In general, to simplify notation from Equation 13 we do not show neither π nor the join condition between R and E: $R_{d+1} = R_d \bowtie E$. The final result table is the union of all partial results: $R = R_1 \cup R_2 \cup \cdots \cup R_k$. If R_d eventually becomes empty at some iteration, because no rows satisfy the join condition, then query evaluation stops. In other words, R reaches a fixpoint [2, 47]). The query evaluation plan is a deep tree with $k - 1$ levels, k leaves with table E and $k - 1$ internal nodes with a $\bowtie$ between R_d and E. Therefore, the query plan is a loop of $k - 1$ joins assuming recursion bounded by k.

The following SQL code implements Seminaïve [35] and it works on any DBMS supporting SQL. Notice graph cycles are filtered out to avoid double counting paths and reducing redundancy. It is a good idea to set a threshold k on recursion depth instead of reaching a fixpoint computation [4], in order to bound evaluation time on large or dense graphs. Since a real database may contain multiple edges per vertex pair it may be necessary to pre-process the graph. In a similar manner, we may insert into temporary tables multiple edges (i.e., bag semantics), which can be later

eliminated to compute the final set union in R.

/* pre-process E: delete duplicate edges per vertex pair

from some input table T with multiple edges per vertex pair */

SELECT $i, j, min(v), max(1)$

INTO E

FROM T

GROUP BY i, j ;

/* base step */

INSERT INTO R_1

SELECT $1, i, j, v, 1$

FROM E;

/* recursive step expansion */

FOR $d = 1 \ldots k-1$ DO

INSERT INTO R_{d+1}

SELECT $d+1$, $R_d.i$, $E.j$,$R_d.p * E.p$, $R_d.v + E.v$

FROM R_d JOIN E ON $R_d.j = E.i$

WHERE $(R_d.i \neq R_d.j)$ /* eliminate loops */ ;

END

/* $R = R_1 \cup R_2 \cdots \cup R_k$ */

FOR $d = 2 \ldots k$ DO

INSERT INTO R

SELECT i, j, p, v FROM R_d;

END

5.2.2 Optimizing Recursive Join: Storage, Indexing and Algorithm

We first study how to efficiently evaluate the most demanding operator in recursive queries: the join operator. We focus on computing G^+, the transitive closure of G, without duplicate elimination since it involves an expensive sort. As explained above, G^+ requires an iteration of $k-1$ joins between R_d and E, where each join operation may be expensive to compute depending on m and G structure. Notice that computing E^k requires a GROUP BY aggregation, which has important performance implications and which has a close connection to duplicate elimination. Duplicate elimination is studied as a separate problem and time complexity is analyzed at the end of this section, after all optimizations have been discussed.

The first consideration is finding an optimal order to evaluate the $k-1$ joins. From the Seminaïve algorithm recall $R_d \bowtie E$ with join comparison $R_d.j = E.i$ needs to be evaluated $k-1$ times. But since evaluation is done by the Semi-naïve algorithm there exists a unique join ordering. $R_1 = E$ and $R_d = E \bowtie E \bowtie \cdots \bowtie E = ((E \bowtie E) \bowtie E) \bowtie \ldots E) \bowtie E$ for $d = 2 \ldots k$. Clearly, the order of evaluation is from left to right. In this work, we do not explore other (associative) orders of join evaluation such as $((E \bowtie E) \bowtie (E \bowtie E)) \bowtie ((E \bowtie E) \bowtie (E \bowtie E)) \ldots$ because they require substantially different algorithms. Computing the final result

$R = R_1 \cup R_2 \cdots \cup R_k$ does not present any optimization challenge when duplicates are not eliminated (Naïve algorithm). Therefore, we will focus on evaluating $R_d \bowtie E$ and then discuss its generalization to $k-1$ joins.

In a columnar DBMS [13, 21, 28, 44] each column is stored on a separate file, where column values are sorted by a specific, carefully selected, subset of columns. Since repeated values end up being contiguous it is natural to use compression. In this case the natural compression algorithm is Run-Length Encoding (RLE), where instead of storing each value the system stores each unique value and its frequency. When there are many repeated values such compressed storage dramatically reduces I/O cost and it can help answering aggregations exploiting the value frequency. It is noteworthy there are no indexes from the DBA perspective: the columnar DBMS maintains internal sparse indexes to the first and last value of each compressed block. The join optimization involves sorting E ordering by i, j and creating a sorted temporary table R_d ordering by j, i (i.e., inverting the two ordering columns) which enables a hash join or a merge join, with a merge join being preferable because for the columnar DBMS merge joins work in time $O(m)$, skipping the sort phase of a sort-merge join. Otherwise, hash joins are a good alternative, with average time $O(m)$, but they are sensitive to skewed key distributions.

In a row DBMS the fastest algorithms are hash joins ($O(m)$ average), followed by merge-sort joins ($O(m \log(m))$ worst case). If both tables are sorted by the joining key the row DBMS can choose the same merge join algorithm, explained above, bypassing the sorting phase resulting also in time $O(m)$. On the other hand,

if one table is sorted, but the other is not the row DBMS generally chooses a sort-merge join taking time $O(m \log(m))$. We should point out that because the join condition is $R_d.j = E.i$, in general there are multiple connecting edges per vertex, resulting in many duplicates. When using hash joins such high number of duplicate values produces many collisions which must be handled. In a row DBMS there are two major choices to accelerate joins: physically sorting rows in R_d or E (or both) or creating an index on i or j to speed-up joins. In general, creating an index is more expensive than sorting in multiple iterations. Since index creation on a large temporary table is expensive and there are $k-1$ temporary tables we sort E edges by i and R_d by j, as the default join optimization. This tuned optimization is equivalent to the sorted projection used in a columnar DBMS.

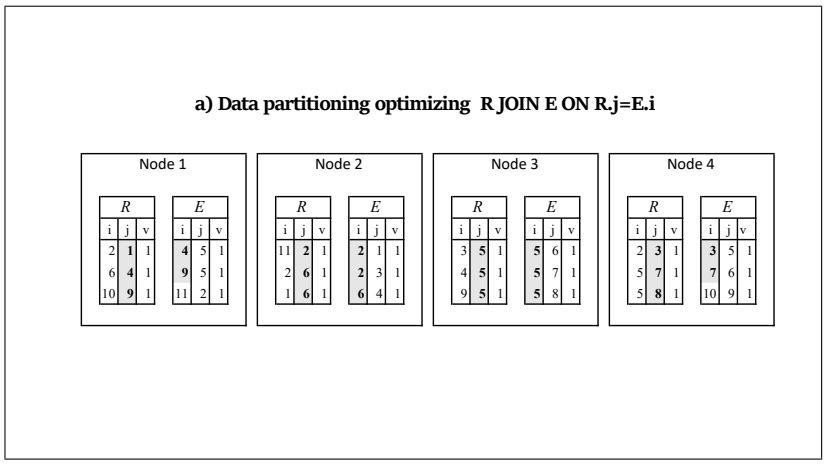
In the array DBMS [46] there are two major features to improve join evaluation: (1) chunk size, which involves setting sizes of a 2-dimensional subarray (i.e., similar to a block of records). (2) sparse or dense storage, which requires knowledge on the fraction and distribution of zeroes across chunks. For a 2-dimensional array setting chunk size further requires deciding if the chunk shape should be squared or rectangular. Since E is squared we believe it is more natural to choose a squared chunk. Deciding sparse or dense storage is easy since it simply involves deleting zeroes. However, manipulation in main memory must be carefully considered: a dense chunk is transferred almost directly into a dense array in main memory, whereas a sparse chunk requires either converting to a dense representation or using a special subscript mechanism for sparse arrays. Needless to say, a dense array in RAM for a dense E is faster. Chunks are automatically indexed based on the chunk boundaries

with a grid-like data structure, which may saturate RAM if chunk size is too small (i.e., a fine grid). The array DBMS physically sorts cells within each chunk in major column order (i.e., 1st chunk dimension, 2nd chunk dimension and so on). Therefore, changing cell order on secondary storage is not possible. We emphasize that in general it is necessary to tune chunk size depending on the graph density. But there is a catch-22 situation: tuning chunk size cannot be done without having some knowledge about G and knowing G structure requires loading G into the array DBMS with a chunk size already set. Therefore, this is a fundamental optimization difference with the column and the array DBMS, where the block size plays a less significant role.

5.3 Data Partitioning

The computation of analytics algorithms under a shared nothing architecture requires a careful distribution of the data. This is specially important for the computation of matrix operations, that may require several passes to the input matrices. In general, the objective of data partitioning is to improve data locality, for instance, exploiting the CPU cache to get the operators closer to the CPU. Our data partition strategy is aimed to improve locality at the level of a computing node in the cluster. Specifically, we study how to speed up the parallel join evaluation, to ensure data locality.

Recall that our algorithm relies on the multiplication of two matrices: R_i and E. To ensure join locality, the data in R_i is distributed through the cluster nodes by a hash function applied to the join column j. Table E is distributed by the same



ure

Figure 5.1: Partitioning of Table E in 4 nodes

hash function, applied in this case to column i. As a result, rows matching on the join condition are found in the same computing node. A second consideration about the data storage is its order. When the two tables in the join are stored sorted by the respective column on the join condition, the matching algorithm does not need to order the data; therefore, a merge join is feasible to join the data. When the data is already sorted, merge join is the most efficient algorithm, with linear time complexity.

5.4 Applications for Graph Algorithms

5.4.1 Triangle Counting

A triangle in a graph is a path of length three that starts in a vertex v_1 and returns to the same v_1. The value of the entry i, j of matrix E^3 corresponds to the number of paths of length 3 from i to j. Thus, the value in $E^3_{i,i}$ is equal to the number

of triangles that the vertex i participates in. Clearly, if G is a directed graph, the number of triangles in G is computed by $\frac{1}{3}\sum_{i=1}^{n} E^3_{i,i}$.

Although elegant, the solution by a summation of the diagonal of E^3 is not as efficient as the following: Compute E^2. Then, compute the diagonal of E^3 only. We express this computation with relational algebra as follows: Compute E^2 to get the number of paths of length 2. Then, join E^2 to E, with joining condition $E^2_j = E_i \wedge E^2_i = E_j$. The result of the join operation is the number of triangles per vertex. To get the total quantity of triangles in the graph, we sum up the number of triangles per vertex and divide by 3.

$$\sigma\pi_{E^2.i,E^2.j,E^2.v}(E^2 \bowtie_{E^2_i=E_j \wedge E^2_j=E_i}) \tag{14}$$

Optimizations for Undirected Graph

When G is undirected, the adjacency matrix becomes symmetric, because for every edge i, j a backward edge j, i shall to be included. As a consequence, the expression in 14 returns many redundant cyclic paths. More over, the computation of E^2 is affected by an unnecessary burden of redundant paths.

5.4.2 Transitive Closure

We consider π as a general operator that projects chosen columns, eliminates duplicates and compute GROUP BY aggregations. We start by discussing duplicate elimination, from which we generalize to group-by aggregations.

Optimizing Duplicate Elimination

This optimization corresponds to the classical transitive closure problem (reachability): edge existence in G^+ instead of getting multiple paths of different lengths (in terms of number of edges, distance or weight). In fact, getting all paths is a harder problem since each path represents a graph and therefore, storage requirements can grow exponentially for dense graphs.

Recall $\pi_{i,j}(R_d) = \pi_{i,j}(E \bowtie E \bowtie \cdots \bowtie E)$. Then the unoptimized query is $\pi_{i,j}(R) = \pi_{i,j}(R_1 \cup R_2 \cup \cdots \cup R_k)$. n UNiOn the other hand, the equivalent optimized query is

$$\pi_{i,j}(R) = \pi_{i,j}(\pi_{i,j}(R_1) \cup \pi_{i,j}(R_2) \cup \cdots \cup \pi_{i,j}(R_k)).$$

Notice a $\pi_{i,j}(R)$ is required after the union. In general, pushing $\pi_{i,j}(R)$ alone requires more work at the end than pushing $\pi_{i,j,sum}(R)$ due to the additional pass, but it does not involve computing any aggregation.

When this optimization is turned off duplicates are eliminated only at the end of the recursion.

SELECT DISTINCT i, j FROM R;

On the other hand, when this optimization is turned on duplicates are incrementally eliminated at each recursion depth d. Notice an additional pass on R is still needed at the end.

FOR $d = 2 \ldots k$ DO

INSERT INTO R

SELECT DISTINCT d, i, j FROM R_d;

END

SELECT DISTINCT i, j FROM R;

Notice a GROUP BY i, j a with a sum() aggregation, does not make sense: the sum $E + E^2 + .. + E^k$ does not make sense since it would overlap path length information, but it makes sense for a min() aggregation. Therefore, this is an important difference between both classes of queries.

In the array DBMS duplicates must be eliminated at each iteration due to the array storage model. Otherwise, it would be necessary to add a new array dimension with the recursion depth, resulting in a 3D array. Therefore, this optimization cannot be turned off in the array DBMS.

Optimizing GROUP BY Aggregation

Considering a graph $G = (V, E)$, the transitive closure of G is a graph G^+ where a directed edge exists from v to v' if v' is reachable from v in graph G. G^+ is defined as: $G^+ = (V, E^+)$, where $E^+ = \{(i, j) |$ ∃a path between i and $j\}$. Notice that G^+ is a new graph with the same vertices as G, but with new edges representing connectivity in a vertex pair. When the set of edges is stored as an Adjacency Matrix E, the

Transitive closure is computed by successive multiplications of E by itself, as follows:

$$E^{+} = \sum_{k=1}^{d} E^{k} \tag{15}$$

5.4.3 All Pairs Shortest Paths

All Pairs Shortest Paths (APSP) can be solved with matrix multiplication, under the min-plus semi-ring. Moreover, in parallel database systems APSP can be computed following the same algorithmic pattern as the transitive closure computation: Recursive queries applying the semi-naive algorithm. Besides, the computation of APSP befits from the same data partitioning strategy. Under the min plus semiring, the min operator replaces $\sum$, and the + operator replaces $\times$. Computing the matrix multiplication in this way, the matrix $E^{k}_{i,j}$ contains the shortest k-edged path from i and j. To compute the shortest path between i and j, regardless of the number of edges of the path, an additional operation is required: compute the overall minimum value between the successive matrices $E...E^{d}$.

$$\text{shortest path from } i \text{ to } j = \min_{k=1}^{d} E^{k}_{i,j} \tag{16}$$

5.5 Experimental Evaluation

Since our SQL-based algorithm and optimizations produce correct results (i.e., we do not alter the basic Semi-naive algorithm), our experiments focus on measuring query processing time. In order to evaluate query processing under challenging conditions we analyze a wide spectrum of graphs having different structure, shape and

connectivity. Our experimental evaluation analyzes three major aspects:

1. Evaluating the impact of classical query optimizations.

2. Understanding the impact of G structure on query processing.

3. Comparing column, row and array DBMSs with each other.

We conducted a careful benchmark comparison tuning each DBMS, but results may vary with other DBMSs, especially if they provide hybrid storage (i.e., row+column) or specialized subsystems for graphs. Also, we aim to understand how effective query optimizations are on new generation DBMSs. We report the average time of three runs per recursive query. Table entries marked with "stop" mean query evaluation could not finish in reasonable time and thus queries were stopped; to evaluate query optimizations we stopped at 30 minutes (1800 seconds) and to analyze the most challenging graphs with optimizations turned on we stopped at 2 hours (7200 seconds). When a DBMS crashed for any reason (insufficient RAM, temporary file/array overflowing temporary storage, bugs) we report "fail". All measured times are given in seconds.

5.5.1 Experimental Setup

Here we provide an overview of how we conducted experiments in order to replicate them.

DBMS Software and Hardware

We compared the three database systems under demanding conditions to force continuous I/O between a single disk and small main memory. To make sure the input table was read from disk, the buffers of each DBMS were cleared before processing each recursive query. We conducted experiments on two identical servers, each with an Intel Quad Core 2.13 GHz CPU, 4 GB RAM and one 3TB disk, each running the Linux Ubuntu operating system. Following DBMS user's guide recommendations, each DBMS was tuned to exploit parallel processing with multi-threading in the multicore CPU. A benchmark on a parallel cluster is out of scope of this paper since DBMSs vary widely on hardware supported, exploiting distributed RAM and parallel capabilities. However, trends should be similar and gaps in performance wider.

We used a columnar DBMS and a row DBMS supporting ANSI SQL. The array DBMS was SciDB [46], which supports AFL, a functional language to define arrays and write queries on arrays and AQL, an SQL-like language based on AFL calls. Our choice of SciDB was motivated by being parallel, matrix-compatible, fully functional and providing the AFL language, capable of expressing SPJ queries, including group-by aggregation. In order to preserve anonymity of the other DBMSs, we do not mention the DBMS name or whether the DBMS is open source or industrial. However, since our benchmark study is based on analyzing query processing without modifying DBMS internal source code, our major research findings should be valuable to users, developers and DBAs trying to decide which system to use.

SQL Code Generator

We developed a generic SQL code generator in the Java language connecting to each DBMS via JDBC (i.e., aiming to generate standard SQL queries). This Java program had parameters to specify the input table (and columns), choose DBMS SQL dialect and turn each optimization on/off. The recursive view was unfolded by creating the iteration of of k SQL statements, following the Seminaïve algorithm from Section 5.2.1. Query evaluation was performed using temporary tables for each step populating each table with SELECT statements. Time measurements were obtained with SQL with timestamps for maximum accuracy.

Experimental Parameters

The buffers of each DBMS were cleared before evaluating the recursive query (i.e., clearing the DBMS cache). That is, we made sure table E was initially read from disk. In order to get evaluation times within one hour and produce a uniform set of intermediate results, we did not run recursion to get the full G^+, which would require a practically unbounded recursion depth (i.e., $k = n$). We initially tested queries on several graphs to investigate a maximum recursion depth k, so that evaluation could finish in less than 1 hour. Based on our findings, we consider $k = 2$ a shallow recursion depth (equivalent to matrix multiplication), $k = 4$ medium and $k = 6$ deep. Only for trees it was feasible to go beyond $k = 6$. We shall convince the reader these seemingly "low" recursion depth levels stress the capabilities of each DBMS.

Table 5.1: Type of graph G.

G	cycles	cliques	density	m edges	complexity
tree	N	N	very sparse	$O(n)$	best
cyclic	Y	N	sparse	$O(n)$	fair
clique-tree	Y	Y	medium	$O(MK^2)$	medium
clique-cyclic	Y	Y	medium	$O(MK^2)$	bad
clique-complete	Y	Y	dense	$O(M^2K^2)$	very bad
complete	Y	Y	very dense	$O(n^2)$	worst

Graph Data Sets

We analyzed synthetic and real graph data sets. Synthetic data sets vary in size and structure, whereas real data sets are fixed.

Synthetic Graphs: We evaluated recursive queries with synthetic graphs, but we were careful to generate realistic graphs with complex structure and varying degree of connectivity, summarized in Table 5.1. Our experimental evaluation used two major classes of graphs: simple graphs where cliques are not part of data generation (tree, cyclic, complete) and graphs where cliques are initially generated and then connected (having prefix "clique-"). Within each class there are three graph types based on their density (connectivity): trees (binary, balanced), cyclic (long cycles) and complete (no edges missing), going from easiest to hardest. Notice a complete graph represents a worst, unrealistic, case, full of cliques from size 3 (triangles) to n (i.e., G itself) to test recursive query processing. In order to understand how recursive queries behave with different graphs we applied a 2-phase data generation approach. In Phase 1 we decide if the graph will have cliques (also called "fat" nodes), which is a major factor impacting query processing time. Then during Phase 2 vertices (or cliques)

are connected. Graphs and their parameters are summarized in Table 5.1. If G has cliques each "fat node" is a clique, whose size we control. We decided to call this parameter K, after the well-known Kuratowski graph K_n, (an important observation is that K, clique size, is denoted in uppercase, not be confused with recursion depth k, in lower case). If G has no cliques each node is "lean", representing a simpler time complexity case. In graphs with "lean" nodes we connect vertices directly with an edge, according to the graph structure, with the number of edges going from n to n^2. For graphs with fat nodes (i.e., prefixed with "clique-") we assume there are initially M "fat nodes", then connected by $M-1$ edges (clique-tree), M edges (clique-cycle) and $M(M-1)$ edges (clique-complete). In this case, we connect some vertex in clique i with some vertex in clique j, in a random manner, guaranteeing cliques are connected with each other. Our graph definitions are comprehensive and subsume disconnected graphs, where each disconnected component can be any of the graphs above. In short, our synthetic graph generator has the following input parameters: n nodes, m edges, M fat nodes and clique size K. Since m is the actual storage size in SQL we generated graphs with m growing in log-10 scale. For graphs with "lean" nodes m determines n, whereas for graphs with "fat" nodes M and K determine n and m. To simplify study we maintain K fixed (e.g., $K = 4$, which represents a family or close mutual friends in a social network, emphasizing solving with K=4 is much harder than with triangles). Needless to say as $K \to n$, G starts resembling a complete graph, making the problem of computing recursive queries intractable.

Real Graphs: For the real data set we picked two well-known data sets from the Stanford SNAP repository: (1) wiki-vote with n=8k and m=103K. (2) Web-Google

Table 5.2: Optimization: recursive join to compute transitive closure G^+ (recursion depth $k = 6$; clique size $K = 4$; times in seconds).

			columnar projection		row order		array storage	
G	n	m	N	Y	N	Y	dense	sparse
tree	10M	10M	112	101	454	437	stop	stop
cyclic	1M	1M	11	12	48	47	stop	1314
clique-tree	312k	1M	1124	1055	stop	stop	fail	771
clique-cyclic	312k	1M	1082	1004	stop	stop	fail	405
clique-complete	1300	100k	stop	stop	stop	stop	41	41
complete	100	10k	stop	stop	stop	stop	25	25

with $n = 916k$ and $m = 5.1M$. Both data sets have a significant number of cliques (including many triangles) and medium diameter, resulting in long paths. Because real data sets are particularly challenging because we cannot totally understand their structure, we analyze them with the best query optimizations turned on in each DBMS.

5.5.2 Evaluating Query Optimizations

We proceed to test the effectiveness of each optimization on each relational operator: $\bowtie$, π, σ (in importance order). In the following experiments the computation is stopped at 30 minutes (1800 seconds).

Optimizing Recursive Join

We start by analyzing the most-demanding operator: the recursive join. As explained in Section 5.2.1, it is necessary to evaluate an iteration of $k-1$ joins. Since storage is different in each DBMS the physical join operator is different and therefore the specific optimization is different as well: projections for the columnar DBMS, sorting rows (edges) in the row DBMS and choosing between dense/sparse storage in the array DBMS. Table 5.2 compares query processing time turning each optimization on and off.

We start by discussing the columnar DBMS, which did not require major tuning. Projections help the columnar DBMS when the graph is very sparse, especially with large trees. For denser graphs, including graphs with cycles, the time gain becomes smaller. Assuming that in general the structure of G is not known, projections (sorted tables by the join key) in the columnar DBMS are a good optimization. Therefore, projections are turned on by default in our remaining experiments.

We now discuss tuning the row DBMS. We experimentally tried two optimizations: (1) indexes on the join vertices $R_d.j$ and $E.i$ and (2) physically sorting rows in R_d and E by the join vertices, as explained in Section 5.2.2, to evaluate the iteration of $k-1$ joins. We found out that physically sorting rows in E with an ORDER BY clause in the CREATE TABLE (i.e., clustered storage for edges) was faster than creating a separate index on E (based on source vertex). Sorting R_d, after being created, was expensive when the DBMS used a hash join. Maintaining an index on

R_d was expensive as well. Notice that under pessimistic conditions, for every recursive query evaluation we included the initial time to sort or index E in our total times. In practice, however, E is sorted or indexed once, but queried multiple times. Therefore, we identified ORDER BY $E.i$ as the default row optimization to accelerate joins. From Table 5.2 we can see sorting rows is moderately effective for sparse graphs, but it does not help anyway with denser graphs (we had to stop the query). Based on these results, we decided to initially sort E to accelerate join processing. Therefore, this optimization is turned on by default.

For the array DBMS the optimization choice are sparse and dense storage for arrays, which are the respective choice for sparse and dense graphs, respectively. As discussed in Section 5.2.2 it is necessary to tune chunk size depending on the graph density. Based on chunk tuning experiments we use two default chunk sizes for the remaining experiments: (1) 1000×1000 for dense graphs; (2) $100,000 \times 100,000$ for sparse graphs, which produced chunks of average size of 8 MBs, as recommended by the DBMS User's Guide. As can be seen from Table 5.2, sparse storage is preferable since times are always smaller and because array storage becomes dense when G is complete. Overall the array DBMS is the fastest with dense graphs (cliques, complete), but it is slower by two orders of magnitude than the columnar DBMS with sparse graphs (trees). The pattern is the same compared to the row DBMS, but with a smaller gap (i.e., row DBMS faster one order of magnitude). The main reason the array DBMS is so slow using dense storage for trees is that it evaluates joins on arrays with almost empty chunks, full of zeroes (i.e., doing unnecessary work). On the other hand, joins are still significantly slow using sparse storage for

Table 5.3: Optimizing projection: Pushing duplicate elimination (recursion depth $k = 6$; clique size $K = 4$; times in seconds).

			columnar optimization		row optimization		array
G	n	m	Y	N	Y	N	default=Y
tree	10M	10M	148	112	728	577	523
cyclic	1M	1M	16	11	67	57	109
clique-tree	312k	1M	49	1103	297	stop	226
clique-cyclic	312k	1M	44	963	229	stop	223
clique-complete	1300	100k	310	stop	stop	stop	616
Complete	100	10k	2	stop	20	stop	16

trees (i.e., zeroes are deleted) because the graph is too sparse and chunks remain sparsely populated (helped a bit by RLE compression). When embedding cliques into the graph array size explodes as depth k grows: only sparse storage works well. These results highlight that the array DBMS is inefficient to evaluate joins on sparse graphs or semi-dense graphs (with cliques) that produce a dense transitive closure graph as they are explored. In conclusion, in further experiments we store G in sparse matrix form by default, eliminating all zeroes.

Optimizing Projection: Pushing Duplicate Elimination and Group-by

The previous experiments do not give the column and row DBMS the opportunity to eliminate duplicates. Table 5.3 helps understanding the impact of duplicate elimination when computing transitive closure G^+. Recall from Section 5.4.2 that in the array DBMS duplicates must be eliminated due to the array storage model. All graphs, except trees, produce duplicates in intermediate results during recursion.

Table 5.4: Optimizing projection: Pushing GROUP BY aggregation to compute E^k (recursion depth $k = 6$; clique size $K = 4$; times in seconds).

			columnar optimization		row optimization		array
G	n	m	Y	N	Y	N	default=Y
tree	10M	10M	288	114	964	689	stop
cyclic	1M	1M	29	11	90	70	1314
clique-tree	312k	1M	60	1450	503	stop	771
clique-cyclic	312k	1M	42	1419	434	stop	405
clique-complete	1300	100k	601	stop	stop	stop	666
Complete	100	10k	3	stop	29	stop	25

Therefore, it is necessary to know whether to eliminate duplicates during recursion or it is better to wait until the end of recursion. Duplicate elimination is unnecessary for trees. Therefore, it is expected time are worse on every DBMS. However, the negative impact is not equally significant: it is significantly worse on the columnar DBMS. Our explanation is that rows must be assembled from separate files for each column and then sorted at each iteration in order to detect duplicates. For the row DBMS, the impact is small, whereas for the array DBMS duplicates are automatically eliminated at each iteration. On the other hand, for dense graphs (with cliques, complete) this optimization becomes a requirement to make the problem tractable: without it times are more than an order of magnitude bigger. With this optimization column and row DBMSs become much more competitive with the array DBMS. In fact, the columnar DBMS becomes uniformly faster. Therefore, the effectiveness of this optimization depends on G structure and recursion depth k. In the absence of information about G structure and because G most likely contains cliques it is better to apply this optimization by default.

Computing the GROUP BY aggregation for E^k should produce a similar trend to computing transitive closure since the query plan is the same. The main difference is computing the aggregated value v, which requires an extra column. Table 5.4 compares pushing GROUP BY aggregation through recursion, as explained in Section 5.4.2. Recall pushing GROUP BY acts as a compression operator since it reduces the size of intermediate results. As we can be seen from Table 5.4 this optimization works very well for column and row DBMSs for dense graphs: without it they crawl. Overall, with this optimization the columnar DBMS becomes the fastest and the row and array DBMS exhibit similar performance with each other. Only with the largest complete graph, an unrealistic worst case, the row DBMS is the worst. In summary, the trends are the same as duplicate elimination. In big data analytics, G is likely to contain cycles and cliques. Therefore, this optimization should be turned on by default.

Optimizing Row Selection: Pushing Selection Filters

Evaluating the effectiveness of pushing σ requires deciding some comparison predicate. By far, the most common is equality. In the case of G the most common predicate is equality on a vertex attribute (e.g., id, name, description). Since G structure varies significantly in our synthetic graphs and in order to have repeatable results, we decided not to choose some random vertex. Instead, we chose vertex i=1, making clear the DBMS has no specific knowledge about such vertex. In this manner, our experiments are repeatable and explainable. That is, optimizing row

Table 5.5: Optimizing row selection: Pushing row filtering on power matrix E^k (recursion depth $k = 6$; clique size $K = 4$; times in seconds).

			columnar optimization		row optimization		array optimization	
G	n	m	Y	N	Y	N	Y	N
tree	10M	10M	18	100	27	361	13	stop
cyclic	1M	1M	2	9	3	41	9	1314
clique-tree	312k	1M	2	990	4	stop	12	771
clique-cyclic	312k	1M	2	976	3	stop	12	405
clique-complete	1300	100k	1	stop	2	stop	7	666
complete	100	10k	1	stop	1	stop	6	25

filtering is done with a WHERE predicate $i = 1$, as shown on Table 5.5. Confirming decades of research, this optimization works well across all DBMSs, regardless of storage mechanism. However, the relative impact is different: in the columnar DBMS and the array DBMS the speed gain is two orders of magnitude with dense graphs, whereas in the row DBMS three or more orders of magnitude with dense graphs (the recursive query cannot finish in fewer than 30 minutes). Therefore, this optimization confirms that a highly selective predicate should be pushed all the way up through recursion when possible. In summary, with this optimization turned on all DBMSs come much closer to each other (assuming the user knows which vertex to explore), but the columnar DBMS still has the leading edge.

5.5.3 Comparing Column, Row and Array DBMSs

In this section we compare the three DBMSs with the best optimization settings based on previous experiments, analyzing challenging graphs at a recursion level as

Table 5.6: Comparing DBMSs with best optimization to get power matrix E^k (fastest join, push group by, duplicates eliminated, no row filtering; stop at 2 hours; times in secs).

					DBMS		
G	cliques	n	m	k	columnar	row	array
tree	N	10M	10M	8	57	1158	3391
clique-cyclic	Y	1M	1M	6	258	322	405
clique-complete	Y	1300	100k	6	601	stop	666
complete	Y	100	10k	6	3	29	25
wiki vote	Y	8k	100k	4	85	4500	426
wiki vote	Y	8k	100k	6	187	stop	1461
web-Google	Y	916k	5M	3	1068	stop	stop
web-Google	Y	916k	5M	4	4232	stop	stop

deep as possible. We analyze synthetic and real graphs. We emphasize real graphs are "harder" than trees, but "easier" than complete graphs. These experiments aim to understand strengths and weaknesses of each system when facing with the task on analyzing a large graph whose structure is not well understood. In this case we stop the computation at 2 hours (7200 seconds), giving each DBMS full opportunity to evaluate the recursive query.

We made a point G structure (shape and connectivity) plays a big role on query processing time. Experiments in Section 5.5.2 uncovered two important facts: (1) there is big time gain when tuning the query plan or graph storage to get faster joins; (2) pushing projection significantly reduces the size of intermediate tables and the additional time to do it in acyclic graphs is small (but not negligible). Therefore, we evaluate recursive query processing with: (1) the fastest join algorithm provided by each DBMS; (2) pushing projection (duplicate elimination, aggregation) at each

iteration. In order to make the comparison in a more challenging manner and having a more realistic (informative) query we analyze the computation of the power matrix E^k, which requires computing a GROUP BY aggregation. Since selecting vertices assumes knowledge about the graph and makes query evaluation much easier (i.e., all DBMSs have similar performance) pushing selection is not applied (e.g., WHERE $i = 1$).

Table 5.6 provides a comparison under a "tuned" configuration, but still without assuming anything about G. Results are interesting: The columnar DBMS is the fastest overall, being faster than both the row DBMS and the array DBMS. For the second place there is no winner: the array DBMS is faster for dense graphs, but loses with sparse graphs. Neither the row DBMS nor the array DBMS can finish analyzing the Google graph. In summary, the columnar DBMS is the fastest, but certainly struggles with the Google graph.

5.5.4 Evaluating Matrix Multiplication Performance

We compare the performance to evaluate E^2 with three mechanisms: (1) Query in array DBMS; (2) ScaLAPACK matrix multiplication, available in SciDB; (2) Query in columnar DBMS. We run experiments with two real graph data sets: webGoogle and LiveJournal. On Table 5.7, the result shows columnar DBMS as a clear winner, about 10 times faster than ScaLAPACK.

Table 5.7: Comparing Matrix Multiplication Performance E^2

Column Vs Array vs ScaLAPACK; times in secs.

G	n	m	$\|E^2\|$	Array DBMS	ScaLAPACK	Columnar DBMS
web-Google	1M	5M	297M	240	189	17
LiveJournal	4.8M	69M	3369M	4232	stop	2889

Table 5.8: Transitive Closure and APSP Execution time; Columnar DBMS

Data set	m	k	Transitive Closure			All Pairs Shortest Path		
			1 Node	4 Nodes	Speed-up	1 Node	4 Nodes	Speed-up
tree-clique	1M	6	50	24	2.1	35	22	1.6
web-Google	5M	4	4843	1256	3.7	3441	1002	3.4
cit-Patents	16M	4	3540	873	3.8	3010	837	3.6

5.5.5 Evaluating Parallel Speed-Up

We run experiments to evaluate the parallel performance and speed-up of our algorithms. We present execution time for N=1 and N=4 for three graph data sets: tree-clique, webGoogle and cit-Patents. The three graphs has a very different structure: tree-clique is a synthetic data set with cliques of size $K = 4$, linked as a binary tree. This is a very sparse data set, and cycles are possible only inside the cliques. On the other hand, web-Google and cit-Patents are real-world graph data sets, from the SNAP repository. Table 5.8 shows results for experiments with a columnar DBMS, and 5.9 with an array DBMS. Results can be visualized in Figure 5.2. The columnar DBMS shows slightly better speed-up, compared to the array DBMS. Notice the small data set presents the lowest speed-up. We have observed that, while the density of the resulting matrix after each multiplication grows very fast for the rel graphs (web-Google, cit-Patents), in case of three clique grows very slow, and the resulting matrix remains very sparse.

Table 5.9: Transitive Closure; Array DBMS

Data set	m	k	Transitive Closure		
			1 Node	4 Nodes	Speed-up
tree-clique	1M	6	355	182	2.0
webGoogle	5M	3	6253	1737	3.6
cit-Patents	16M	3	7420	2592	2.9 6

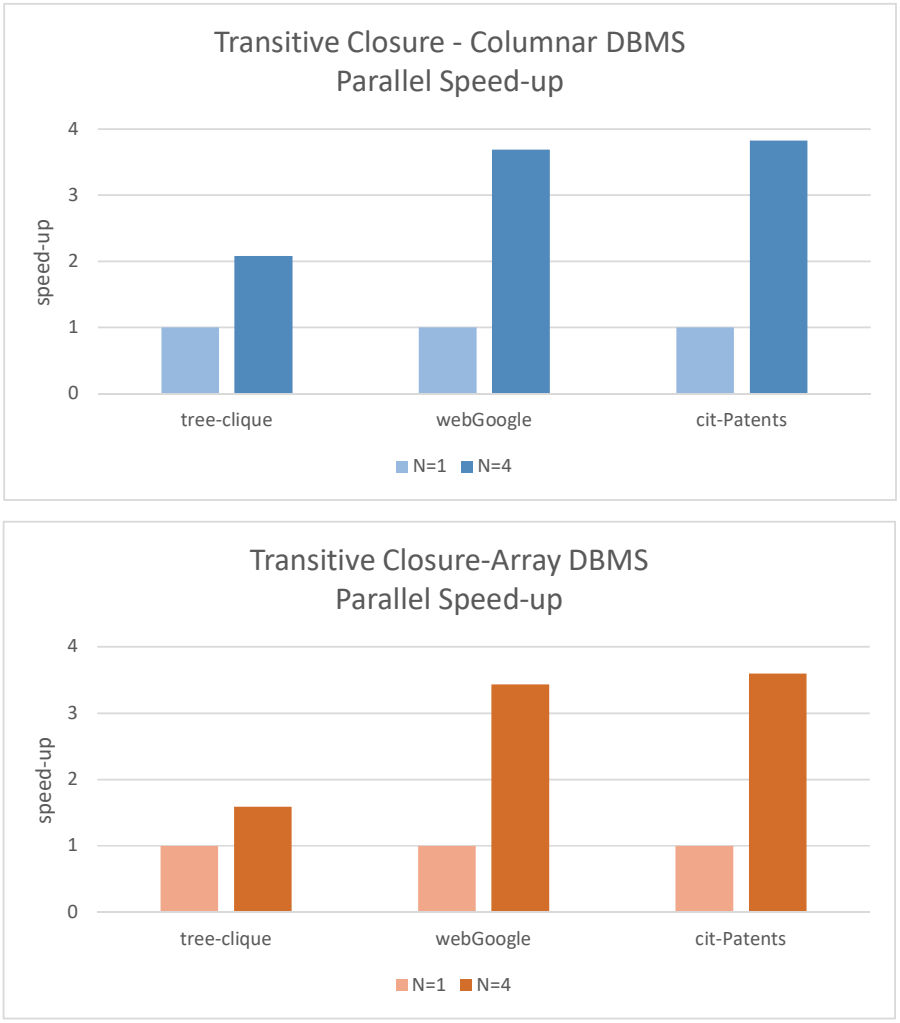


Figure 5.2: Parallel speed-up for Transitive Closure

Chapter 6

Graph Analytics with Parallel Matrix-Vector Multiplication

6.1 Computing an Iteration of Matrix-Vector Multiplications

Matrix-vector multiplication is an important primitive in many Machine learning algorithms. Solving graph problems with matrix vector multiplication has been studied in previous research. But there is scarce literature about solving graph algorithms via matrix-vector multiplication. This way to compute graph algorithms is important for this work because: 1) provides a common framework for several graph problems; 2) the challenges of parallel matrix-vector multiplication are already known; 3) matrix-vector multiplication can be expressed with relational operators in a simple

way.

6.1.1 Semirings and Matrix Multiplication

Semirings are algebraic structures defined as a tuple $(R, \oplus, \otimes, 0, 1)$ consisting of a set R, an additive operator $\oplus$ with identity element 0, a product operator $\otimes$ with identity element 1, and commutative, associative and distributive properties holding for the two operators in the usual manner. The regular matrix multiplication is defined under $(\mathrm{R}, +, \times, 0, 1)$. A general definition of matrix multiplication expands it to any semiring. For example, on the min-plus semiring , min is the additive operator $\oplus$, and + is the product operator $\otimes$. The min-plus semiring is used to solve shortest path problems, as in [11]. Table 6.1 shows examples of relational queries to compute matrix-vector multiplication under different semirings.

6.1.2 Unified Algorithm

Solving large graphs with iterative matrix-vector multiplication may look counterintuitive: a large graph with one million vertices would lead to a huge adjacency matrix with one trillion cells; the multiplication of such a large matrix times a large vector is clearly unfeasible. Though, since real world graphs are sparse, the adjacency matrix would need in general $O(n)$ space. Moreover, when the input matrix and vectors are stored sorted, the computation of the multiplication can be done with a merge join in $O(n)$ time, and a group-by, whose time complexity can be done in $O(n)$ time when a grouping by hashing is possible.

Table 6.1: Matrix-Vector Multiplication with Relational Queries under common semirings

$E \cdot S$ (+ , x) semiring
SELECT E.i, sum(S.v * E.v) FROM E JOIN S on E.j=S.i GROUP BY i

$E^T \cdot S$ (+ , x) semiring
SELECT E.j, sum(S.v * E.v) FROM E JOIN S on E.i=S.i GROUP BY i

$E \cdot S$ (min, +) semiring
SELECT E.i, min(S.v +E.v) FROM E JOIN S on E.j=S.i GROUP BY i

$E \cdot S$ ($agg()$,$\bigotimes$) general semiring
SELECT E.i, agg(S.v $\bigotimes$ E.v) FROM E JOIN S on E.j=S.i GROUP BY i

Data: Table E, Table S_0, ϵ, optional: source vertex s
Result: S_d

$d \leftarrow 0$; $\Delta \leftarrow \infty$;
while $\Delta > \epsilon$ **do**
  $d \leftarrow d + 1$;
  $S_d \leftarrow \pi_{i:\oplus(E.v \otimes S.v)}(E \bowtie_{j=i} S_{d-1})$;
  $\Delta = f(S_d, S_{d-1})$;
end
return S_d ;

Algorithm 2: Graph Algorithms Evaluated with Relational Queries

Table 6.2: Comparison of four graphs algorithms

Characteristic	Reachability	SSSP	WCC	PageRank
Computed as	$S_d \leftarrow E^T \cdot S_{d-1}$	$S_d \leftarrow E^T \cdot S_{d-1}$	$S_d \leftarrow E^T \cdot S_{d-1}$	$S_d \leftarrow T \cdot S_{d-1}$
Semiring op.$(\oplus, \otimes)$	$sum(), \times$	$\min(), +$	$\min(), \times$	$sum(), \times$
Value $S[i]$	number of paths	distance s to i	id of component	probability
S_0 defined as	$S_0[s] = 1$	$S_0[s] = 0$	$S_0[s] = s$	$S_0[i] = 1/n$
$\|S_0\|$	1	1	n	n
Output	S_k	S_k	S_k	S_k
Time per iteration	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Scope	from source s	from source s	$\forall i \in V$	$\forall i \in V$

Algorithm 2 is a pattern to solve several graph problems with an iteration of relational queries. We base Algorithm 2 in our previous work[6], where we expressed some graph algorithms with relational algebra. Table 6.2 is useful to understand similarities between four graphs algorithms, which come to light when they are expressed as matrix-vector operations. This algorithm pattern can be applied in relational databases and array databases [7]. Furthermore, we keep the query as simple as possible, as follows:

1. The query joins two tables
2. The query performs an aggregation, grouping by 1 column
3. The output of the query is inserted in an empty table. We do not do updates. The size of $S_d \leq n$

In a relational DBMS, the actual operation to compute the matrix multiplication is a regular query with a join between E and S, and a subsequent aggregation. In array databases, the operation can be implemented either via join-aggregation or calling the built in matrix multiplication operator but we demonstrate later that the

first option presents better performance.

We use matrix-vector multiplication: (1) as an abstraction, for a better understanding of the algorithms; (2) because it has been extensively proved that some graph algorithms are equivalent to matrix vector multiplication.

The entries of E might be weighted or not, depending on the problem: unweighted entries for PageRank and WCC, and weights representing distances for shortest paths. Prior to the first iteration, the vector S has to be set to an initial state accordingly to the problem: infinite distances for Bellman-Ford, isolated components in WCC, or a default initial ranking on PageRank. In the *pth* iteration, the vector S_p is computed as $E \cdot S_{p-1}$. A function $f(S_d, S_{d-1})$ returns a real number to check convergence. The algorithm iterates when Δ (the value returned by f) is less than some small value ϵ, or when it reaches the max number of iterations.

6.2 Data Partitioning

In this section, we present a graph partition strategy to improve data locality in the computation of the parallel join, as well as even data distribution. In Section 6.4 we will present an experimental evaluation of our strategy.

In the case of the family of algorithms studied in this work, keeping a low data transfer between nodes is critical to achieve good performance. The core computation of our algorithms is the query which solves the matrix-vector multiplication, comprised of a join and an aggregation. Moreover, because of the iterative nature of

Table 6.3: Data partitioning and Physical Ordering in Columnar DBMS

Algorithm	Join	Partition	Order
SSSP	$E \bowtie_{i=i} S$	hash($E.i$);hash($S.i$)	$E.i, S.i$
WCC	$E \bowtie_{i=i} S$	hash($E.i$);hash($S.i$)	$E.i, S.i$
PageRank	$T \bowtie_{j=i} S$	hash($T.j$);hash($S.i$)	$T.j, S.i$
Reachability	$E \bowtie_{i=i} S$	hash($E.i$);hash($S.i$)	$E.i, S.i$

these algorithms, this query is computed many times. We focus on optimizing the parallel join, the most demanding operation. The parallel join runs efficiently when excessive data transfer between nodes is avoided. By a careful data partition, we ensure that rows in S matching rows in E are found in the same worker node. The joining column in E can be either i or j, depending on the algorithm (see Table 6.3).

6.2.1 Partitioning in a Columnar DBMS

The illustration in Figure 6.1 shows a graph G, with 11 vertices. In the same figure, we show the representation of the graph as a list of edges, stored in a database table E. The graph should be partitioned in such a way that uneven data distribution and costly data movement across the network is avoided. The latter is possible when the parallel join occurs locally on each worker node. To ensure join data locality, we partition table E and S by the join key. Depending on the algorithm, the join key for table E is either i ot j. Table S is clearly partitioned by the vertex id.

Specifically, if the join condition is $E_i = S_i$ (Figure 6.2.a), the edges having the same **starting** vertex are stored in only one computing node, along with the corresponding vertices in S. When the join condition is $E_j = S_i$ (Figure 6.2.a), the edges having the same **ending** vertex are stored in only one computing node,

along with the corresponding vertices in S. The benefit of this partition is that any vertex in E has the corresponding matching vertex in S in the same computing node, avoiding costly data movement.

6.2.2 Partitioning in an Array DBMS

Our goal is two-folded: To balance the computation among the nodes, and to allow locality in the parallel join, in order to avoid costly data transfer. In general, big-data graphs are characterized by a sparse adjacency matrix. In the case of array DBMSs, the adjacency matrix is partitioned by chunks, blocks of homogeneous size. SciDB assigns chunks in a rigid way: considering a matrix split in chunks numbered from 1 to κ, and considering the workers numbered as $1...N$, chunks are assigned to the workers just by the formula *chunknumber* mod N. Further than the problem of skewed degree distributions, in some cases the top vertices (ordered by degree) might be identified by a "close" number-id. For instance, in the Live Journal data set, a small group of vertices located in the first rows of the adjacency matrix have hundreds and even thousands links. As a result, a few blocks of the matrix concentrates a large amount of data. To alleviate that problem, we partition the data with a simple strategy: re indexing of the adjacency matrix, as follows:

$$i' \rightarrow H * (i \bmod N) + i/N; \tag{17}$$

$$j' \rightarrow H * (j \bmod N) + j/N; \tag{18}$$

Where H is the chunk size. Figure 6.3 shows a plot of the density of the adjacency matrix before and after the re-indexing. Furthermore, Figure 6.4 shows the data

distribution among the workers. Clearly, the reindexing is helpful to alleviate the problem of uneven partitions.

6.2.3 Partitioning in Spark-GraphX

GraphX includes a set of built-in partitioning functions for the edges collection. Following the vertex-cut approach, edges are never cut. Edges are partitioned by several strategies.

- Random Vertex Cut: The graph is partitioned by assigning edges to computing node in random way
- Edge Partition 1D: the adjacency matrix is partitioned by horizontal cuts.
- Edge Partition 2D: the adjacency matrix is partitioned in a grid manner, both horizontal and vertical cuts.

6.3 Applications for Graph Algorithms

Columnar DBMS. We programmed simple but efficient SPJA queries that perform matrix multiplication. In the parallel columnar DBMS, three factors are important for a good performance per iteration:

1. Local match: Rows that satisfy the join condition are always in the same computing node. This is key to avoid data transfer between nodes.

2. Presorted data: The join between E and S can achieve a linear time complexity when the tables are presorted by the columns participating in the join condition. The algorithm is the MERGE join. This is critical for very large graphs.

3. Data Compression: Columnar data storage is favorable for efficient data compression [1]; in this way the I/O consumption is reduced.

Array DBMS. We propose to compute the matrix-vector multiplication with a combination of join and aggregation operations, and we compare our approach to the standard way: call the built-in *spgemm()* SciDB operator; this operator internally calls the high performance linear algebra library SCALAPACK [9]. In the array DBMS, a carefully data partition let us to compute the join minimizing data transfer: the matches of the join condition are always in the same node. The (sparse) array-like data organization makes possible a merge join, since data is stored in order. On the other hand, data partitioning needs to consider skewed data distribution. It is natural to assign the edge (i, j) to the position (i, j) in the disk array. But due to the power low, this naive procedure may lead to uneven data partitioning. To alleviate this problem, we allocate the data with a function that redistributes the data when the graph is skewed. Like the columnar DBMS, queries in the array DBMS can be optimized to exploit: (1) Local match for parallel joins; (2) Presorted data, which is inherent of the array-based data organization.

Spark-GraphX. This Biga data system stores graphs with two main data structures, namely EdgeRDD and VertexRDD, which are extensions of the Spark RDD data structure. The fundamental operation to solve graph problems in GraphX is *aggregateMessages*, which receives as parameters a *sendmsg* (or map) function, and an *aggregate* (or reduce) function. As output, aggregateMessages returns an RDD which associates every vertex with the computed value. In [16], Gonzalez et al. state "We identified a simple pattern of join-map-groupby dataflow operators that forms the basis of graph-parallel computation. Inspired by this observation, we proposed the GraphX abstraction".

6.3.1 Reachability from a Source Vertex

Preliminaries

Reachability from a source vertex s is the problem aimed to find the set of vertices S such that $v \in S$ iff exists a path from s to v. It is well known that this problem can be solved with a Depth-first search (DFS) from s, a Breadth-first search (BFS) from s, or via matrix multiplications. In [27], the authors explain that a BFS starting from s can be done using a sparse vector S_n (initialized as $S[s] = 1$, and 0 otherwise), and multiplying iteratively E^T by S, as in Eq. 19

$$S_k = (E^T)^k \cdot S_0 = E^T \cdot ... \cdot (E^T \cdot (E^T \cdot S_0)) \text{ (k vector-matrix products)} \tag{19}$$

where $\cdot$ is the regular matrix multiplication and S_0 is a vector such that:

$$S_0[i] = 1 \text{ when } i = s, \text{ and 0 otherwise} \tag{20}$$

Initialization Like Bellman-Ford, reachability from a source vertex starts the iterative process with a sparse vector S_0, initialized as $S_0[s] = 1$. In the same way, $E[s, s]$ is set to 1.

Data: Table E, source s
Result: Table S_d
1 **Initialization** $S_0[s] \leftarrow 1$; $E[s, s] \leftarrow 1$;
/* Iterations */
2 $d = 0$; $\Delta = 1$;
3 **while** $\Delta > \epsilon$ **do**
4 $\quad d = d + 1$;
5 $\quad S_d \leftarrow \pi_{j:min(E.v*S.v)}(E \bowtie_{i=i} S_{d-1})$;
6 $\quad \Delta \leftarrow \sum S_d - \sum S_{d-1}$
7 **end**
8 return S_d ;

Algorithm 3: Reachability from a Source Vertex

Iterations Like Connected Components, this algorithm stops when $S_d = S_{d-1}$ Since $S_d[i] >= S_{d-1}[i]$, then $S_d = S_{d-1}$ if $\sum S_d - \sum S_{d-1}$. The relational query to compute the matrix product is presented below. Since the query in array DBMS has small syntactical differences, it is omitted.

```
insert into S1
select E.j, sum(S0.v*E.v) v
from E join S0 on S0.i = E.i
group by E.j;
```

Computation in Spark-GraphX Reachability from a Source has not be implemented in Spark-GraphX. Even though, it is possible to use the SSSP routine as an alternative.

6.3.2 Bellman Ford (SSSP)

Preliminaries

Bellman-Ford is a classical algorithm to solve the Single Source Shortest Path problem (SSSP). In contrast to Dijkstra's algorithm, Bellman-Ford can deal with negative-weighted edges. The algorithm iterates on every vertex, and execute a *relaxation* step for each edge of the current vertex [11]. A way to express Bellman-Ford with matrix-vector multiplication under the min-plus semi-ring is explained in [14]. The shortest path of length k from a source vertex s to every reachable $v \in E$ can be computed as:

$$S_k = (E^T)^k \cdot S_0 = E^T \cdot ... \cdot (E^T \cdot (E^T \cdot S_0)) \text{ (k vector-matrix products)} \tag{21}$$

where $\cdot$ is the min-plus matrix multiplication and S_0 is a vector such that:

$$S_0[i] = 1 \text{ when } i = s, \text{ and } \infty \text{ otherwise} \tag{22}$$

Notice that the expression to compute SSSP looks similar to the computation of reachability, but the initialization and the multiplication (min,+), are different. At each iteration, the resulting vector holds the "current" minimum value. From a relational point of view, the vector S_d is is stored S_d in a database table with schema

$S_d(j, v)$, where j is a destination vertex, and v is the minimum distance known at the current iteration (*relaxed* distance). Both the standard and the linear algebra algorithms require to initialize as ∞ every vertex, but the source. Instead, in a relational database we only include vertices in S_d when an actual path from s has been found. When the algorithm starts, S_d is sparse; only one non-zero value. The matrix multiplication under the min-plus semi-ring reproduces the relaxation step: In the dth iteration, the minimum distance is computed considering the relaxed value from the iteration, stored in S_{d-1}, as well as the value of new edges discovered in the current iteration.

Initialization The table S_0 representing the initial vector is initialized inserting a row with values (s,0), where s is the source vertex. Also, an artificial self-loop with value zero (no distance) is inserted to E, which has the effect to keep shortest path found in previous iterations in the current S. While initializing S, Bellman-Ford requires that entries of the vector different than s be set to ∞. In the database systems, those values are not stored.

Iterations Following our algorithmic pattern, the iterative process stops when Δ is equal or less a value ϵ. The value Δ is assigned to zero only when the current vector S_d is equal to the previous, S_{d-1}. The relational query that computes the min-plus matrix vector multiplication with relational queries is presented below.

```
insert into S1
select E.j, min(S0.v+E.v) v
```

Data: Table E, source s
Result: Table S_d
1 **Initialization** $S_0[s] \leftarrow 0$; $E[s,s] \leftarrow 0$;
/* Iterations */
2 $d = 0$; $\Delta = 1$;
3 **while** $\Delta > \epsilon$ **do**
4 $\quad d = d + 1$;
5 $\quad S_d \leftarrow \pi_{j:min(E.v*S.v)}(E \bowtie_{i=i} S_{d-1})$;
6 $\quad \Delta \leftarrow$case $S_d == S_{d-1}$ then 0 else 1;
7 **end**
8 return S_d ;

Algorithm 4: Single Source Shortest Path

```
from E join S0 on S0.i = E.i
group by E.j;
```

Computation in Array DBMS The computation of the vector S_p can be done either by matrix-vector multiplication using SPGEMM() or by a join-aggregation. As demonstrated in the Experimental section, a cross join operation presents better performance, taking advantage of data locality.

```
insert into S1
select E.j, min(S0.v+E.v) v
from cross_join(E,S, E.i,S.i)
group by E.j
```

Computation in Spark-GraphX The SSSP routine in the Spark-GraphX library is a standard implementation based on message-passing and aggregation. The full code is available in the Spark-GraphX source code repository.

6.3.3 Weakly Connected Components (WCC)

Preliminaries

A weakly connected component of a directed graph G is a subgraph G' such that for any vertices $u, v \in G'$, exists an un-directed path between them. A recent, but well known algorithm is HCC, proposed in [26]. The algorithm is expressed as an iteration of a special form of matrix multiplication between the adjacency matrix E and a vector (called S to unify notation) initialized with the vertex-id numbers. The $sum()$ operator of the matrix multiplication is changed to the $min()$ aggregation. Each entry of the resulting vector is updated to the minimum value between the result of matrix computation and the current value of the vector. Intuitively, vertex v receives the ids of all its neighbors as a message. The attribute of the vertex is set to the minimum among its current value, and the minimum value of the incoming messages. The iterative process stops when S remains unchanged after two successive iterations. Our Connected Components algorithm is an improvement of HCC, an iterative algorithm proposed in [26]. The algorithm in [26] can be explained as follows: Let S a vector where each entry represents a graph vertex. Initialize each value of S with the corresponding vertex ids. In the iteration d, the connected components vector S_d is updated as:

$$S_d = assign(E \cdot S_{d-1}) \tag{23}$$

where $assign$ is an operation that updates $S_d[i]$ only if $S_d[i] > S_{d-1}[i]$ and the dot represents the min,* matrix multiplication. This algorithm has been applied in

Map-Reduce and Giraph. Recently, the authors of [23] applied HCC in a RDBMS. As showed in this work, the authors implemented the algorithm joining three tables: edges, vertex, and v_update.

We propose to compute the new vector just with the SPJA query for matrix vector multiplication (join between two tables plus aggregation). In contrast with HCC [26], we avoid the second join, necessary to find the minimum value for each entry of the new and the previous vector. We avoid the three-table join proposed by [23], too. We propose inserting an artificial self loop in every vertex; by setting $E(i, i) = 1$, for every i.

Initialization As explained, we have to insert 1s in the diagonal of E, to simplify the query. Each entry of the table S_d is initialized with the vertex-id.

Data: Table E,
Result: Table S_d
1 **Initialization** $S_0[i] =\leftarrow i$; $E[i, i] \leftarrow 1$;
`/* Iterations */`
2 $d = 0$; $\Delta = 1$;
3 **while** $\Delta > 0$ **do**
4 $\quad d = d + 1$;
5 $\quad S_d \leftarrow \pi_{j:min(E.v*S.v)}(E \bowtie_{i=i} S_{d-1})$;
6 $\quad \Delta \leftarrow \sum S_d - \sum S_{d-1}$
7 **end**
8 return S_d ;

Algorithm 5: Connected Components

Iterations The algorithm stops when the current vector is equal to the second. Since $S_d[i] <= S_{d-1}[i]$, then $S_d = S_{d-1}$ if $\sum S_d - \sum S_{d-1}$. The relational query is

presented below. The array DBMS query has small syntactical differences.

```
insert into S1
select E.i, min(S0.v*1) v
from E join S0 on S0.i = E.j
group by E.i;
```

Spark-GraphX Graphx includes in its library an implementation of Connected Components similar to HCC, propagating minimum vertex-ids through the graph. The implementation follows the Pregel's message-passing abstraction.

6.3.4 PageRank

Preliminaries

PageRank [41] is an algorithm created to rank the web pages in the world wide web. The output of PageRank is a vector where the value of the ith entry is the probability of arriving to i, after a random walk. Since PageRank is conceived as a Markov process, the computation can be performed as an iterative process that stops when the Markov chain stabilizes. The algorithms previously described in this section base their computation on E. Conversely, it is well known that PageRank can be computed as powers of a modified transition matrix [25]. The transition matrix T is defined as $T_{i,j} = E_{j,i}/outdeg(j)$ when $E_{j,i} = 1$; otherwise $T_{i,j} = 0$. Notice that if $outdeg(j) = 0$, then the jth column of T is a column of zeroes. Let $T' = T + D$, where D is a $n \times n$ matrix such that $D_{i,j} = 1/n$ if the column j is a 0 column. To

overcome the problem of disconnected graphs, PageRank incorporates an artificial low-probability jump to any vertex of the graph. This artificial jump is incorporated by including a matrix A. Let A be a $n \times n$ matrix, whose cells contains always 1, and p the damping factor. The power method can be applied on $T^{''}$ defined as: $T^{''} = (1-p)T^{'} + (p/n)A$, as presented in Equation 24.

$$S_k = (T'')^k \cdot S_0 \tag{24}$$

Although Equation 24 seems to be simple, computing it with large matrices would be unfeasible. While T might be sparse, $T^{'}$ is not guaranteed to be sparse. Moreover, since A is dense by definition, T'' is dense, too. Equation 24 can be expressed as based in the sparse matrix T as follows:

$$S_d = (1-p)T \cdot S_{d-1} + (1-p)D \cdot S_{d-1} + (p/n)A \cdot S_{d-1} \tag{25}$$

This full equation of PageRank computes exact probabilities at each iteration. Because $(1-p)D \cdot S_{d-1}$ is a term that adds a constant value to every vertex, it is generally ignored. After simplification, the expression for PageRank becomes:

$$S_d = (1-p)T \cdot S_{d-1} + P \tag{26}$$

where every entry of the vector P is equal to p/n. It is recommended to set $p = 0.15$ [41].

PageRank is simple, but it is necessary to consider carefully the relational query to avoid mistakes. According to Equation 26, the main computation in PageRank is the multiplication $T \cdot S$, that is solved in parallel DBMSs as a join. As a collateral effect of using sparse data, the join between T and S does not return rows for those

vertices having in-degree equal to zero (no in-coming edges). When the in-degree of a vertex v is zero, it does not exist any row in E such that $E.j = v$. Thus a row $T.i = v$ does not exist, either. Therefore, in the next iteration the PageRank value of v is lost. Moreover, vertex v will be neglected in further iterations. One solution to this problem is to compute the PageRank vector with two queries: The SPJA query for matrix vector multiplication, and a second query to avoid missing vertices, inserting the constant value p/n for such vertices having in-degree equal to zero, previously stored in a temporary table *VertexZeroIndegree*.

```
insert into S1  /* query 1 */
select T.i, p/n + (1-p)*sum(T.v*S0.v)
from T join S0 on S0.i=T.j
group by T.i;

insert into S1 /* query 2 */
select S0.i, p/n
from S0
where S0.i in
( select v from VertexZeroIndeg)
```

To keep the algorithm elegant and efficient, we avoid using two queries. To avoid "query 2", we insert an artificial zero to the diagonal of the Transition Matrix as part of the initialization. This is equivalent to the two-queries solution, and it does not alter the numerical result.

Initialization: Our first step is to compute the transition matrix T, which requires the computation of the out-degree per vertex. T is carefully partitioned, to enforce join locality. The vector S_0 is initialized with a uniform probability distribution. Therefore, $S[i] = 1/n$.

```
/* Initialization: Computing the transition matrix */
insert into T
select E.j i, E.i j, 1/C.cnt v
from E,
(select i, count(*) cnt
from E
group by i) C
where E.i = C.i;
```

Data: Table E,
Result: Table S_d
1 **Initialization** $S_0[i] =\leftarrow 1/n$; $T[i,j] \leftarrow E[j,i]$/outdeg(i); $T[i,i] = 0$;
/* Iterations */
2 $d = 0$; $\Delta = 1$;
3 **while** $\Delta > \epsilon$ **do**
4 $\quad d = d + 1$;
5 $\quad S_d \leftarrow \pi_{i:sum(T.v*S.v)}(T \bowtie_{j=i} S_{d-1})$;
6 $\quad \Delta \leftarrow max(S_d[i] - S_{d-1}[i])$
7 **end**
8 return S_d ;

Algorithm 6: PageRank

Iterations: Algorithm 2 shows that in every iteration a new table is created. Since we just need the current S and the previous one, we actually use only table S_0 and

table S_1, swapping them at each iteration. PageRank algorithm keeps iterating until convergence, meaning that for every entry of the output vector, the difference with respect to the same entry of the vector of the previous iteration is less than a small value ϵ. The relational query is defined as follows:

```
/* SQL query for a PageRank iteration  */
insert into S1
select T.i, p/n + (1-p)*sum(T.v*S0.v) v
from T join S0 on S0.i=T.j
group by T.i;
```

Like in the columnar DBMS, the base of the computation in the array DBMS is iterative matrix vector multiplication. The input is the matrix E stored as a "flat" array, a uni-dimensional array where i, j, v are attributes. This flat array is used to compute the Transition matrix as a sparse bi-dimensional array, and it is partitioned to avoid unbalances due to skewed distributions. The query in the array DBMS uses the built-in operator `cross_join()` and `group by`. Note that the first pair of parameters in cross_join are the two tables, and the second pair of parameter are the joining attributes.

```
/* AQL query for a PageRank iteration in array DBMS */
insert into S1
select T.i, p/n + (1-p)*sum(T.v*S0.v) v,
from cross_join(T,S0,S0.i,T.j)
group by T.i;
```

Table 6.4: Data Sets

Data Set	Description	n	m	Avg degree	Max degree	Max WCC
web-Google	Hyperlink	0.9M	5M	11.66	6,353	855,802
soc-pokec	Social Network	1.6M	30M	37.51	20,518	1,632,803
LiveJournal	Social Network	4.8M	69M	28.25	22,887	4,843,953
wikipedia-en	Hyperlink	12.4M	378M	62.24	963,032	11,191,454
Web Data Commons	Hyperlink	42.9M	623M	29.04	3.9 M	39.4 M

6.3.4.0.1 Computation in Spark-GraphX: We explain the algorithm included as part of the GraphX library. PageRank is solved iteratively; `aggregateMessage` is the main operation at each iteration. This operation is conceptualized as a map function applied to messages sent from neighbor nodes, and a reduce function that performs an aggregation. Specifically, the map function is a scalar multiplication, and the aggregation is a summation. The output of `aggregateMessage` is a VertexRDD. Though a different data structure, the content of the VertexRDD is similar to the output of the join-aggregation in columnar DBMS.

6.4 Experimental Evaluation

We conduct experiments to compare performance and results of three graph algorithms, under three different systems: An industrial columnar database, an open source array database (SciDB) and the well known GraphX for Apache Spark. The three systems were installed in the same hardware: a four node cluster, each node with a Quad core Intel CPU. The cluster has in total 16 GB RAM and 4TB of disk storage, running Linux Ubuntu.

6.4.1 Data Sets

In graph analysis, hyperlink and social networks are considered challenging, not only because their size, but the skewed distribution of the degree of their vertices. In the case of web graphs, a popular page can be referenced for many thousands pages. Likewise, a social network user can be followed for thousands of users, too. We study our algorithms with three data sets from the SNAP repository [30], one dataset from Wikipedia, and a very large web graph data set from Web Data Commons with m=620M [29]. All the data sets are well known, and statistics are publicly available, as maximum degree, average degree, number of triangles, and size of the largest weakly connected component.

6.4.2 Evaluation of Query Optimizations

In section 4, we explained that the objectives of our optimizations are: 1) local computation of the join $E \bowtie S$; 2) presorted data to compute the join with merge-join algorithm. We run experiments to validate our optimizations, both in parallel columnar and array DBMSs.

Evaluating Optimizations in a Columnar DBMS

We compare the benefits of our proposed partitioning strategy, identified in this experiment as A, versus B, a classical optimization in parallel DBMS by replication of the smallest table in the join to every node in the parallel cluster. We load the Living Journal data set, and we grew it up two, three and form times, to run

experiments with several input sizes. The table E is partitioned in the same way both in A and in B. In case A, the table S is partitioned with the same join key as table E. In case B, the table S (smaller than E) is replicated through the cluster. Figure 6.5 compares the execution time of the two strategies for an iterative computation of PageRank in various data set sizes. Our strategy (labeled in the figure as "A") is superior than the replication of the smaller table (labeled as "B"). How can our optimization be better than a data replication through the cluster? Recall that the algorithms in this study work with several iterations, and that in each iteration the vector is recomputed. Therefore, keeping a local copy has as drawback that the result table needs to be replicated to the complete cluster in every iteration. We run a second set of experiments to measure the average time that takes to solve a PageRank query in the columnar database. Figure 6.6 show the execution time for several social network sizes. This query runs in a time complexity close to linear.

Evaluating Optimizations in an Array DBMS

We partition the data set according our strategy, and then compute an iteration of PageRank, in three different ways: (1) SciDB's built-in matrix multiplication operator, calling ScaLAPACK; (2) our join-aggregation query (3) Our join-aggregation query, plus re-partitioning to ensure even distribution of the data across the cluster. Let us consider the results in Figure 6.7. Even though the data is partitioned to ensure a local join, ScaLAPACK (right bar) takes longer time to evaluate the query. Instead, our join-and-aggregation query performs better, taking advantage of a local join. Further performance improvements are presented by the left bar,

Table 6.5: Comparing Columnar DBMS vs Array DBMS vs Spark-GraphX

Algorithm	Data set	m	Columnar	Array	GraphX
Reachability	web-Google	5M	19	141	34
	soc-pokec	30M	25	164	59
	LiveJournal	69M	60	386	166
	wikipedia-en	378M	364	4311	crash
	Web Data Commons	620M	2139	stop	crash
SSSP	web-Google	5M	13	145	34
	soc-pokec	30M	25	172	59
	LiveJournal	69M	58	405	166
	wikipedia-en	378M	487	4574	crash
	Web Data Commons	620M	2763	stop	crash
WCC	web-Google	5M	24	175	32
	soc-pokec	30M	53	345	83
	LiveJournal	69M	125	919	451
	wikipedia-en	378M	443	5091	crash
	Web Data Commons	620M	3643	stop	crash
PageRank	web-Google	5M	18	143	58
	soc-pokec	30M	72	380	153
	LiveJournal	69M	99	1073	477
	wikipedia-en	378M	507	stop	crash
	Web Data Commons	620M	2764	stop	crash

where the execution time is improved due to re-partition of the data that balances the computation through the cluster nodes.

6.4.3 Comparing performance of Columnar DBMS, Array DBMS and Spark-GraphX

Results of our experiments are presented in Table 6.5, and in Figure 6.8. The vertical axis represents the execution time The time measurement for the three systems (columnar DBMS, array DBMS and Spark-GraphX) includes the iterative step and

the time to partition the data set. We allow a maximum execution time of 120 minutes; after that time, the execution is stopped. The experiment with the largest data set (Web Data Commons, 620 millions of edges) was successfully completed by the columnar DBMS, but could not be finished neither for the array database (stopped after 120 minutes) nor for Spark-GraphX (program crashes). The Spark program works well for those data sets that fit in RAM, but crashes when the data set is larger than RAM, after struggling to solve the join on data distributed through the cluster. Our experimental results show that in general, the algorithms presented in this work have superior performance in the columnar DBMS than the array DBMS. Besides, in the columnar DBMS our algorithms present equal results and better performance than standard implementations in GraphX, specially when the data sets are large. Even when the data set fits in the cluster RAM, our algorithms running on top of a columnar DBMS run at least as fast as in Spark-GraphX. Our experiments show also that columnar and array DBMS can handle larger data sets than Spark-Graphx, under our experimental setup.

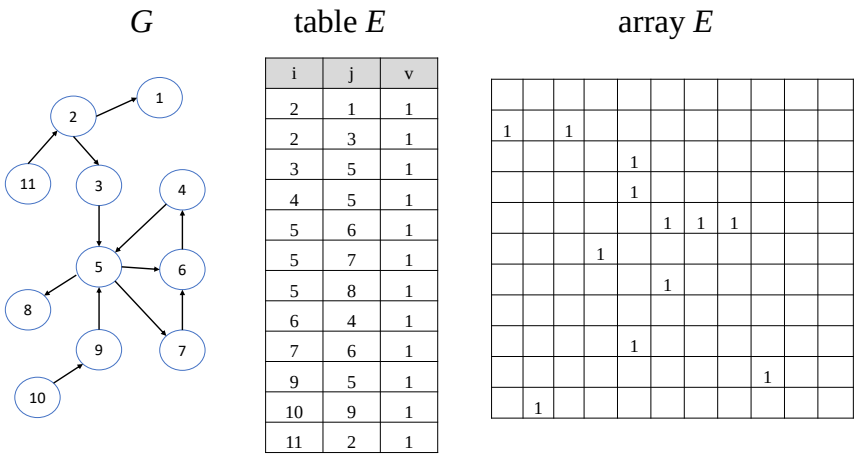
6.4.4 Parallel speed-up experimental evaluation

We present a parallel speed-up evaluation comparing the execution time of the columnar parallel DBMS in a four-node cluster versus the parallel DBMS running in one node. We show the experimental results in Table 6.6. By definition, the parallel speed-up is $S = t_1/t_N$. Our experiments shows that larger graph data sets benefit from parallel processing, obtaining a speed-up from 2.50 up to 3.8. In contrast, the

Table 6.6: Serial vs Parallel Execution performance for 3 algorithms in Columnar DBMS

Data set	m	SSSP		WCC		PageRank	
		1 Node	4 Node	1 Node	4 Node	1 Node	4 Node
web-Google	5M	14	13	49	24	34	18
LiveJournal	69M	156	58	452	125	272	99
wikipedia-en	378M	1243	487	1725	443	1195	366

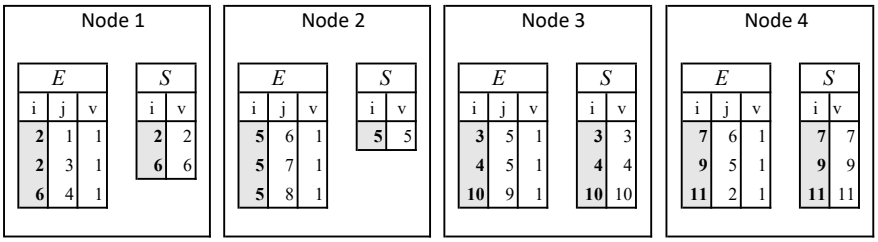
experiments with the small data set (5 million edges) present a lower speed-up. Recall that the concept behind of our algorithms is a recursive matrix multiplication of a matrix E and a vector S, which are stored in a DBMS as relational tables. The superior parallel performance of weakly connected components and PageRank can be explained considering the two tables that are read in the relational query at every iteration. In the case of WCC and PageRank algorithms the vector S starts as dense, and remains dense in the whole computation. With a dense S, the processing happens in an even way, promoting parallelism. In contrast, in the case of SSSP and Reachability the initial vector S is very sparse (only one entry), though the density of the vector gradually increases in every iteration.



i	j	v
2	1	1
2	3	1
3	5	1
4	5	1
5	6	1
5	7	1
5	8	1
6	4	1
7	6	1
9	5	1
10	9	1
11	2	1

Figure 6.1: A sample graph G stored as a table E or as a bi-dimensional array E .

a) Data partitioning optimizing E JOIN Z ON E.i=S.i



b) Data partitioning optimizing E JOIN Z ON E.j=S.i

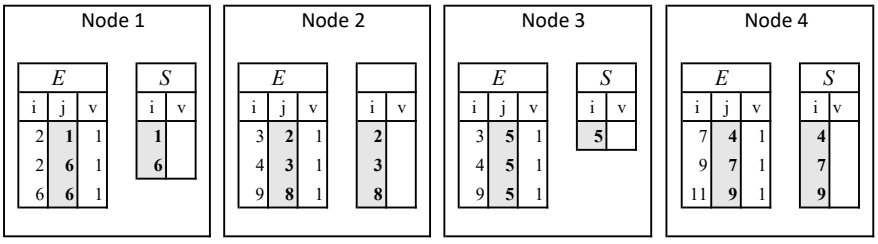


Figure 6.2: Partitioning of Table E in four nodes.

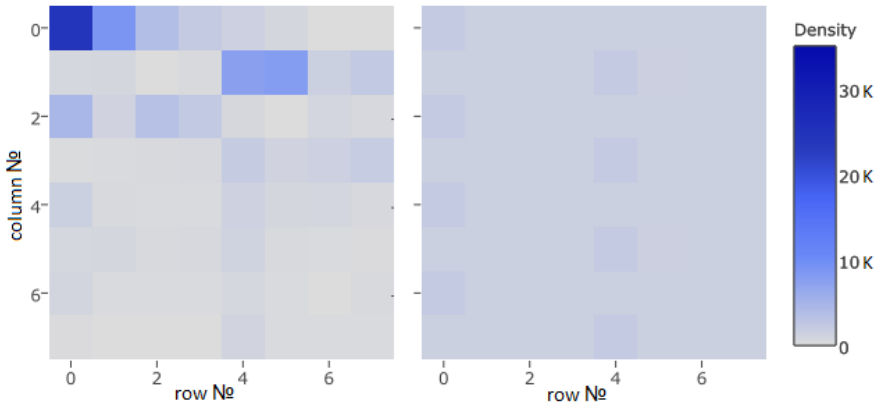


Figure 6.3: Array DBMS: Live Journal data set. Adjacency matrix heat map before (left) and after (right) repartitioning.

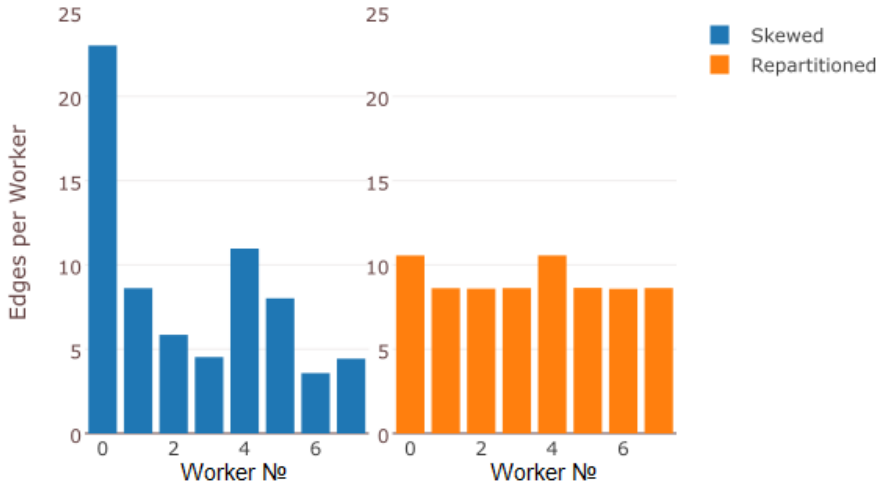


Figure 6.4: Array DBMS: Live journal data set: Another perspective of the data density.

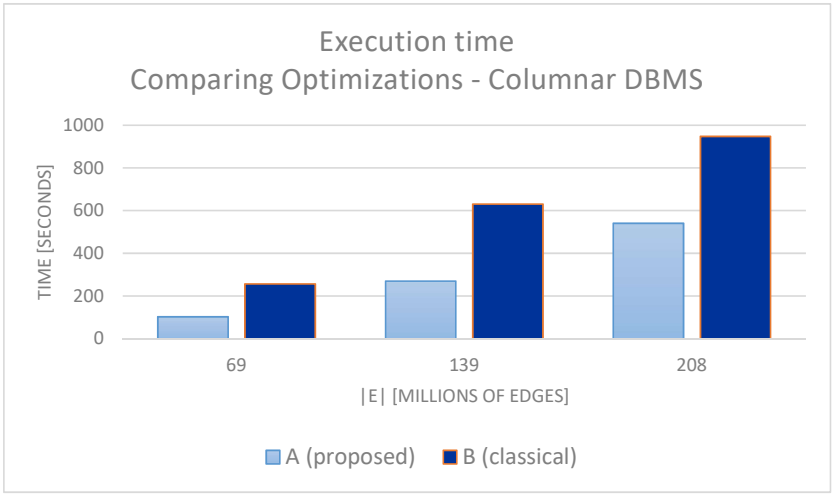


Figure 6.5: Comparing execution times in columnar DBMS: (A) Proposed partition strategy. (B) Classical parallel join optimization by replication of the smaller table

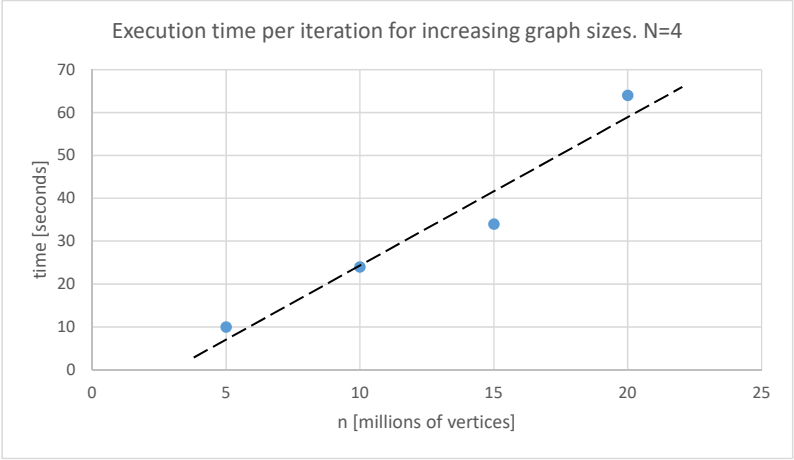


Figure 6.6: Execution time of the join-aggregation query for PageRank in columnar DBMS.

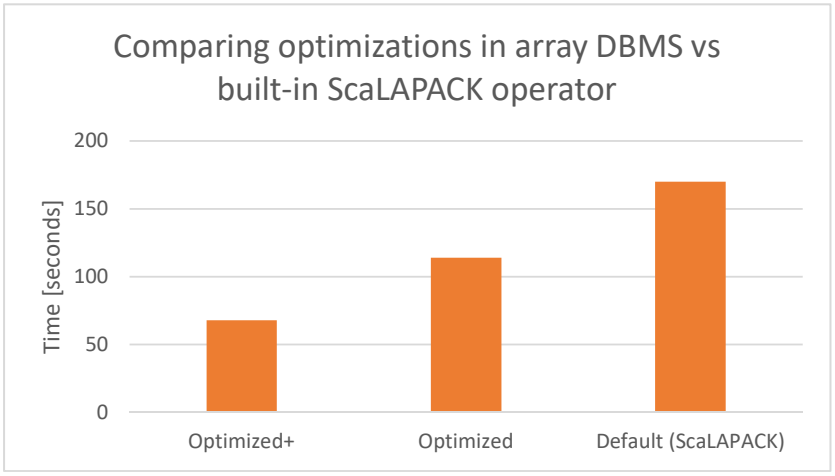


Figure 6.7: A comparison of Matrix Multiplication in SciDB. Slowest computation with the SciDB's built-in operator calling ScaLAPACK. Faster computation with join and aggregation plus repartitioning.

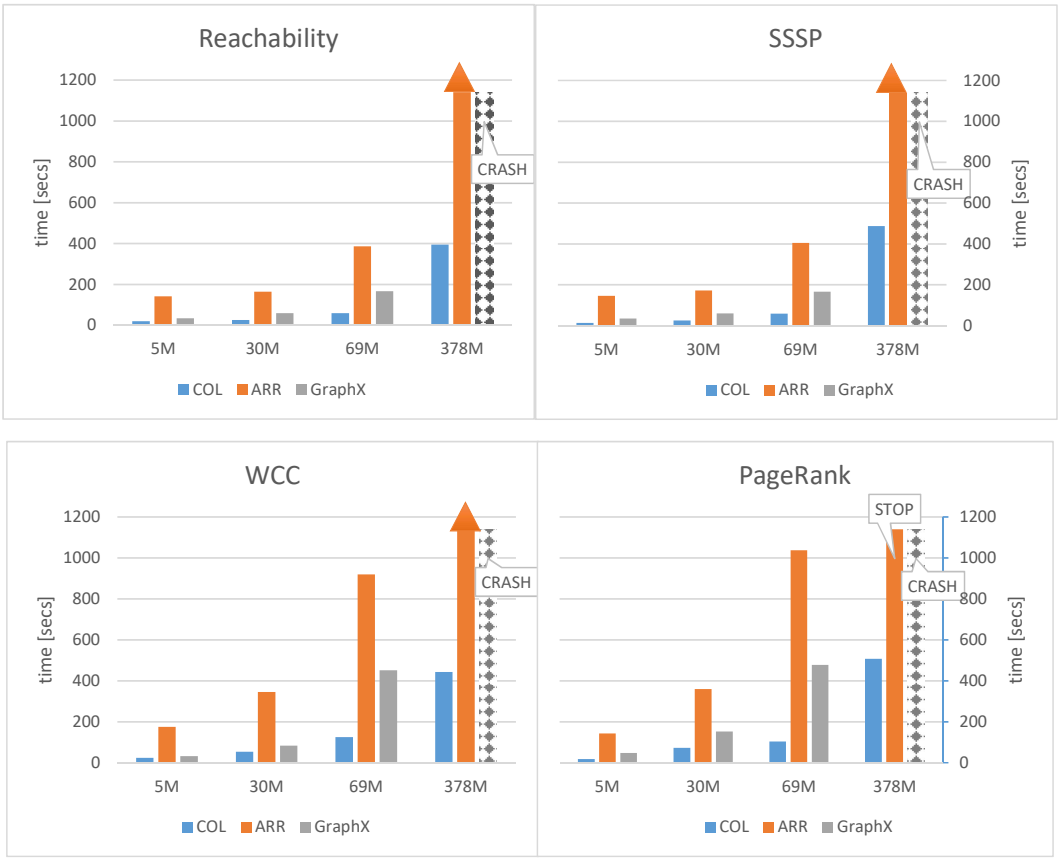


Figure 6.8: Performance Comparisons: Columnar DBMS, Array DBMS and Spark-GraphX

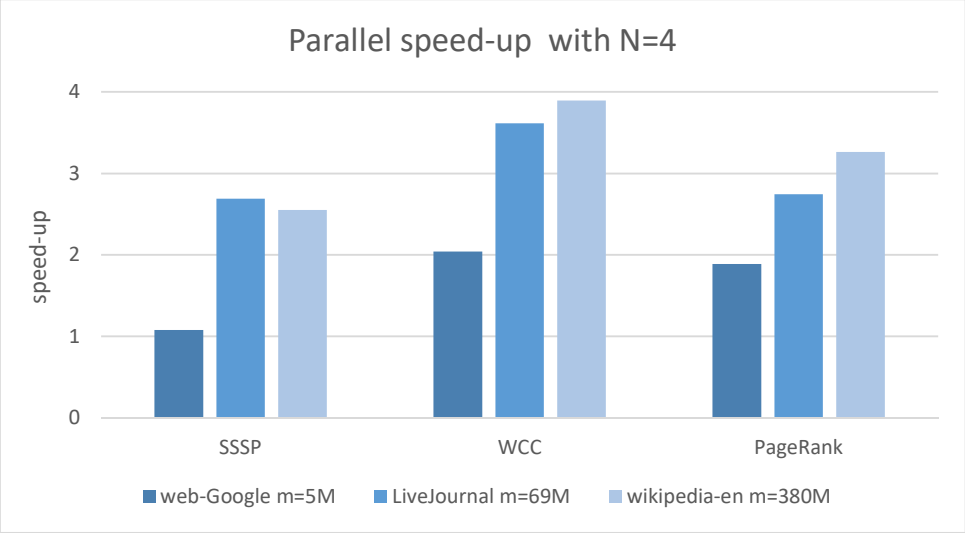


Figure 6.9: Parallel speed-up for SSSP, Connected Components and PageRank for 3 data sets (4 Nodes)

Chapter 7

Conclusions

In this work we study the parallel in-database computation of three special classes of matrix products that are frequently computed as part as machine learning and graphs algorithms: the product of a matrix by its transposed, the kth power of a matrix, and the product of a sparse matrix by a vector.

We present optimizations for three methods widely used in machine learning and data analytics: linear regression, variable selection and PCA. We show that pre-computation of the summarization matrix Γ is an effective strategy to accelerate the aforementioned algorithms, by avoiding redundant computations. Furthermore, in virtue of the scalability of the computation of Γ, the computation of the models is able to deal with very large data sets, even larger than the system RAM. Our algorithms are up to two order of magnitude faster than standard approaches. The RAM requirements are small, as the size of the summarization matrix Γ is $d \times d$, with neglectable extra space required for the computation.

We demonstrate that relational queries are suitable to solve fundamental graph problems: (1) transitive closure computation, (2) all pairs shortest paths, (3) triangle counting, (4) reachability from a source vertex, (5) single source shortest path, (6) weakly connected components, and (7) PageRank. We present algorithms to compute (1),(2) and (3) on the foundation of matrix powers, incorporating several query optimizations and a data partition strategy which promotes data locality on the node level, on columnar and array DBMSs. Our experimental results show our algorithms have promising performance and parallel speed up, due to several query optimizations and our data partitioning strategy. Besides, our results shows that columnar DBMS presents a superior performance computing sparse matrix-matrix multiplication with relational queries, compared with the array system calling ScaLAPACK.

We show a common algorithmic pattern to solve graph problems (4), (5), (6) and (7) , based on an iteration of matrix-vector multiplications, evaluated equivalently with an iteration of relational queries. The unified computation can solve different problems by computing the matrix-vector multiplication under different semirings. Based on this framework we studied query optimization on columnar and array database systems, paying close attention to their storage and query plans. Furthermore, we propose a graph partitioning approach that promotes join locality as well as even data partitioning through the parallel cluster. We remark that our algorithms are based only regular queries, avoiding UDFs or internal modifications to the database. Therefore, our optimizations are easily portable to other systems. In the experimental section we used real graph data sets to demonstrate that the

join, the most challenging operation in parallel, runs with a performance close to linear. Our experiments show a promising parallel speed-up, specially when the vector S is dense. By comparing a columnar DBMS, an array DBMS and Spark-GraphX, we observed that the columnar DBMS shows superior performance and scalability, being able to handle the largest graphs. The performance of the columnar DBMS is better than Spark-GraphX even when the data set fits in the cluster's main memory. Our results show that the array DBMS performs 2-3 times slower than Spark, and up to 10 times slower than the columnar DBMS. We believe this results are related to problems in SciDB's operators handling sparse matrices. Even though, the array DBMS is more reliable than Spark-GraphX when the graph data set is larger than the cluster RAM.

Future Work

We plan to devise more in-database machine learning algorithms, specially exploiting parallel matrix-vector multiplication. In the graph analytics field, Our work sheds light on a family of algorithms that can be optimized as a single one, which opens many opportunities for future work. We want to understand if there exist other graph algorithms which can be also unified with similar ideas. We plan to study further optimizations that may that advantage not only of the sparsity of the matrix, but also of the sparsity of the vector, even considering different degrees of sparsity. Moreover, we will look for opportunities to improve algorithms beyond graph analytics, exploiting our optimized sparse matrix-vector multiplication with relational

queries.

Bibliography

[1] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, S. Madden, et al. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases : The Logical Level*. Pearson Education POD, facsimile edition, 1994.

[3] R. Agrawal, S. Dar, and H. Jagadish. Direct and transitive closure algorithms: Design and performance evaluation. *ACM TODS*, 15(3):427–458, 1990.

[4] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. ACM SIGMOD Conference*, pages 16–52, 1986.

[5] W. Cabrera, D. Benhaddou, and C. Ordonez. Solar power prediction for smart community microgrid. In *2016 IEEE International Conference on Smart Computing, SMARTCOMP 2016, St Louis, MO, USA, May 18-20, 2016*, pages 1–6, 2016.

[6] W. Cabrera and C. Ordonez. Unified algorithm to solve several graph problems with relational queries. In *Proceedings of the 10th Alberto Mendelzon International Workshop on Foundations of Data Management, Panama City, Panama, May 8-10, 2016*, 2016.

[7] W. Cabrera and C. Ordonez. Parallel graph algorithms with in-database matrix-vector multiplication. *Distributed and Parallel Databases (DAPD)*, ?(?), 2017.

[8] W. Cabrera, C. Ordonez, D. S. Matusevich, and V. Baladandayuthapani. Bayesian variable selection for linear regression in high dimensional microarray data. In *Proceedings of the 7th international workshop on Data and text mining in biomedical informatics*, pages 17–18. ACM, 2013.

[9] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. Whaley. Scalapack: a portable linear algebra library for distributed memory computers–design issues and performance. *Computer Physics Communications*, 97(1-2):1–15, 1996.

[10] J. Choi, J. Dongarra, R. Pozo, and D. Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, pages 120–127. IEEE, 1992.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[12] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[13] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database: An architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[14] J. T. Fineman and E. Robinson. *5. Fundamental Graph Algorithms*, chapter 5, pages 45–58.

[15] E. I. George and R. E. McCulloch. Variable selection via Gibbs sampling. *Journal of the American Statistical Association*, 88(423):881–889, 1993.

[16] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, 2014.

[17] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182, 2003.

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

[19] 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. *Proc. of VLDB*, 5(12):1700–1711, 2012.

[20] R. R. Hocking. A biometrics invited paper. the analysis and selection of variables in linear regression. *Biometrics*, 32(1):1–49, 1976.

[21] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. Mullender, and M. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[22] J. L. Jianqing Fan. Sure independence screening for ultrahigh dimensional feature space. *Journal of the Royal Statistical Society*, 70:849–911, 2008.

[23] A. Jindal, S. Madden, M. Castellanos, and M. Hsu. Graph analytics using vertica relational database. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 1191–1200. IEEE, 2015.

[24] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker. Vertexica: your relational friend for graph analytics! *Proceedings of the VLDB Endowment*, 7(13):1669–1672, 2014.

[25] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub. Extrapolation methods for accelerating pagerank computations. In *Proceedings of the 12th Int. Conf. on World Wide Web*, pages 261–270. ACM, 2003.

[26] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 229–238, Washington, DC, USA, 2009. IEEE Computer Society.

[27] J. Kepner and J. Gilbert. Graph algorithms in the language of linear algebra, 2011.

[28] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandier, L. Doshi, and C. Bear. The Vertica analytic database: C-store 7 years later. *PVLDB*, 5(12):1790–1801, 2012.

[29] O. Lehmberg, R. Meusel, and C. Bizer. Graph structure in the web: Aggregated by pay-level domain. In *Proceedings of the 2014 ACM Conference on Web Science*, WebSci '14, pages 119–128, New York, NY, USA, 2014. ACM.

[30] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

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

[32] D. Matusevich, W. Cabrera, and C. Ordonez. Accelerating a gibbs sampler for variable selection on genomics data with summarization and variable preselection combining an array DBMS and R. *Machine Learning*, 102(3):483–504, 2016.

[33] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

[34] T. J. Mitchell and J. J. Beauchamp. Bayesian variable selection in linear regression. *Journal of the American Statistical Association*, 83(404):1023–1032, 1988.

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

[36] C. Ordonez, W. Cabrera, and A. Gurram. Comparing columnar, row and array DBMSs to process recursive queries on graphs. *Information Systems*, 63:66–79, 2017.

[37] C. Ordonez, C. Garcia-Alvarado, and V. Baladandayuthapani. Bayesian variable selection in linear regression in one pass for large datasets. *ACM TKDD*, 9(1):3, 2014.

[38] C. Ordonez, S. Maabout, D. S. Matusevich, and W. Cabrera. Extending ER models to capture database transformations to build data sets for data mining. *Data & Knowledge Engineering*, 2013.

[39] C. Ordonez, M. Navas, and C. Garcia-Alvarado. Parallel multithreaded processing for data set summarization on multicore CPUs. *Journal of Computing Science and Engineering*, 5(2):111–120, 2011.

[40] 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.

[41] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[42] S. Puntanen and G. P. Styan. The equality of the ordinary least squares estimator and the best linear unbiased estimator. *The American Statistician*, 43(3):153–161, 1989.

[43] E. Soroush, M. Balazinska, and D. Wang. ArrayStore: A storage manager for complex parallel array processing. In *Proc. ACM SIGMOD Conference*, pages 253–264, 2011.

[44] 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 *Proc. VLDB Conference*, pages 553–564, 2005.

[45] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The architecture of SciDB. In *Proceedings of SSDBM*, SSDBM'11, pages 1–16. Springer-Verlag, 2011.

[46] 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.

[47] J. Ullman. Implementation of logical query languages for databases. *ACM Trans. Database Syst.*, 10(3):289–321, 1985.

[48] A. Welc, R. Raman, Z. Wu, S. Hong, H. Chafi, and J. Banerjee. Graph analysis: do we have to reinvent the wheel? In *First International Workshop on Graph Data Management Experiences and Systems*, page 7. ACM, 2013.

[49] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud USENIX Workshop*, 2010.