Distrib Parallel Databases (2017) 35:335–362
DOI 10.1007/s10619-017-7200-6

# Scalable parallel graph algorithms with matrix–vector multiplication evaluated with queries

**Wellington Cabrera**[1] · **Carlos Ordonez**[1]

Published online: 29 August 2017
© Springer Science+Business Media, LLC 2017

**Abstract** Graph problems are significantly harder to solve with large graphs residing on disk compared to main memory only. In this work, we study how to solve four important graph problems: reachability from a source vertex, single source shortest path, weakly connected components, and PageRank. It is well known that the aforementioned algorithms can be expressed as an iteration of matrix–vector multiplications under different semi-rings. Based on this mathematical foundation, we show how to express the computation with standard relational queries and then we study how to efficiently evaluate them in parallel in a shared-nothing architecture. We identify a common algorithmic pattern that unifies the four graph algorithms, considering a common mathematical foundation based on sparse matrix–vector multiplication. The net gain is that our SQL-based approach enables solving "big data" graph problems on parallel database systems, debunking common wisdom that they are cumbersome and slow. Using large social networks and hyper-link real data sets, we present performance comparisons between a columnar DBMS, an open-source array DBMS, and Spark's GraphX.

**Keywords** Graph · Parallel computation · Data distribution · Columnar DBMS · Array DBMS

✉ Wellington Cabrera
wcabrera@cs.uh.edu

Carlos Ordonez
ordonez@cs.uh.edu

1 University of Houston, Houston, USA

# 1 Introduction

Graph analytics is a field which is increasing its importance every day. Furthermore, as the world has become more interconnected than before, graph data sets are larger and more complex. In graph analytics, the goal is to obtain insight and understanding of complex relationships that are present in telecommunication, transportation and social networks. In general, real-life graphs are sparse. While there are more than one billion user accounts in Facebook, a typical user may be connected just to a few hundred contacts. Other examples are roads connecting cities and flights linking airports. In this work, we concentrate on algorithms for sparse graphs stored in parallel DBMSs.

Relational database systems 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 RDBMSs cannot handle demanding graph problems, because relational queries are not sufficient to express important graphs algorithms, and a poor performance of database engines in the context of graph analytics. Consequently, several graph databases and graphs analytics systems have emerged, targeting large data sets, especially under the Hadoop/MapReduce platform. In recent years, Pregel and its open-source successor Giraph have supported the "vertex-centric" approach. This approach is based on performing computations at the vertex scope and sending/receiving messages to/from its neighbors. On the other hand, the "algebraic approach" solves graph algorithms via a generalized matrix multiplication, generally with optimized programs running in clusters with large main memory.

## 1.1 Why in-database graph analytics

DBMSs are between the most widespread systems in the industry. It is hard to imagine how to run an organization without using DBMSs, in any industrial field. Moreover, most of the internet relies on DBMSs for social networking, dynamic content or electronic commerce. Therefore, a lot of data is stored in these systems. We believe that efficient graph algorithms for relational databases are a contribution that will support in-database graphs analytics in large data sets, avoiding wasting time exporting the data or setting up external systems.

## 1.2 Summary of contributions

In this work we show how to compute several graph algorithms in a parallel DBMS, by executing iteratively one elegant (yet efficient) relational query. In contrast to popular graph analytics systems, our algorithms are able to process data sets larger than the main memory, since they are conceived as external algorithms. We present a unified algorithm based on regular relational queries with **join-aggregation** to solve classic graph problems, such as Reachability from a source vertex, Single Source Shortest Paths, Connected Components, and PageRank. We explain a graph partitioning strategy which reduces the execution time of the parallel queries of our unified algorithm, by improving data locality and

avoiding redundant data. While a general optimization of graph queries is out the scope of this work, we show experimentally that by using optimized **join-aggregation** queries, parallel DBMSs can compute a family of fundamental graph algorithms with promising performance. Furthermore, we show that columnar and array DBMSs are competitive to a state-of-the art system, namely Spark-GraphX.

## 2 Related work

In the last years the problem of solving graph algorithms in parallel DBMS with relational queries has received limited attention. Recently, the authors of [14] studied Vertica as a platform for graph analytics, focusing in the conversion of vertex-centric programs to relational queries, and in the implementation of shared-memory graph processing via UDFs, in one node. Rudolf et al. [26] describes the enhancements to SAP HANA to support graph analytics, in a columnar in-memory database. In [31], the authors show that their SQL implementation of shortest path algorithm has better performance than Neo4j. Note that the later work runs in one node with a large RAM (1 TB). Running on top of Hadoop, Pegasus [16] 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. Pregelix [3] is another graph system, built on top of the Hyracks parallel dataflow engine; the authors report better performance than GraphX, claiming that this system brings "data-parallel query evaluation techniques from the database world". Our work is different from 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 DBMSs based on a careful data distribution; 3) the out-of-core graph computation allows us to analyze graphs with hundreds millions edges with minimum RAM requirements.

A sub-problem of sparse-matrix vector multiplication is analyzed in [25]. Specifically, the authors propose an array-relation dot-product join database operator, motivated by the computation of Stochastic Gradient Descent (SGD). Regarding the matrix–vector multiplication with relational operators, the authors argue about its applicability to SGD. In contrast, our work is focused on the optimization of graph algorithms. The algorithms of our concern do not require incremental updates; instead, the matrix vector multiplication updates completely the results. While our algorithms do not require gradient methods, it would be interesting to study if the dot product operator could be applicable to graph problems.

In our initial work [21], we proposed optimized recursive queries to solve two important graph problems using SQL: Transitive closure and All Pairs Shortest Path. Our recent work [22] shows that columnar DBMS technology performs much better than array or row DBMSs. More recently, we turned our attention to graph algorithms based on matrix–vector multiplication with relational queries in [4], where we explored a unified algorithm to solve several graph problems, based on relational operators. However, DBMS storage details, parallelism, and query optimizations were not studied.

# 3 Definitions and background

## 3.1 Graph dataset

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 $i$th entry of $S$ is a value corresponding to vertex $i$, and it is denoted as $S[i]$.

### *3.1.1 Graph storage*

In sparse graphs processing, it is reasonable to represent the adjacency matrix of $G$ in a sparse form, which saves storage and computing resources. There exist several mechanisms to represent sparse matrices, and the interested reader may check [2]. In our work, sparse matrices are represented as a set of tuples $(i, j, v)$ such that $v \neq 0$, where $i$ and $j$ represent row/column, and $v$ represents the value of entry $E_{ij}$. Since cells where $v = 0$ are not stored, the space complexity is $m = |E|$. In sparse matrices, we assume $m = O(n)$.

## 3.2 Parallel systems overview

The algorithms in this work are conceived for parallel DBMSs under a shared-nothing architecture. While our optimized algorithms can work in any DBMS, in our previous work [22], we showed experimentally that columnar and array DBMSs present performance substantially better than row DBMSs for graphs analysis. For this reason we concentrate the study in columnar and array DBMSs. These systems are architected for fast query processing, rather than transaction processing. In this work, we consider parallel DBMSs running in a cluster with $N$ nodes, where each node has separate RAM and disk.

### *3.2.1 Row DBMS*

The pioneer parallel database management systems stored data by rows. This systems were aimed to exploit the I/O bandwidth of multiple disks [7], 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.

*3.2.2 Columnar DBMSs*

Columnar DBMSs emerged in the previous decade presenting outstanding performance for OLAP. C-Store [29] and Monet-DB [12] 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. Columnar DBMS indexing is significantly different than traditional row stores. For instance, in Vertica there is no row-level index defined by the DBA. Instead, additional projections can be defined to improve query execution. An in-depth study of columnar DBMS architectures is given in [1].

*3.2.3 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 multidimensional arrays. Array stores organize array data by data blocks called chunks [27,28,30], distributed across the cluster. In bi-dimensional arrays, chunks are square or rectangular blocks. The chunk map is a main memory data structure which 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. An important difference between array and relations is that user-defined indexes are unnecessary: The subscripts of the array are used to locate the corresponding chunk on disk, and to find the specific position in the chunk. Parallel array DBMSs distribute data through the cluster's disk storage on a chunk basis using diverse strategies. A detailed review of array DBMS architectures is found in [27].

*3.2.4 Spark*

Spark [32] is a system for interactive data analytics built on top of HDFS. The main abstraction of Spark is the Resilient Distributed Dataset (RDD), an immutable collection of objects which can be distributed by partitions across the cluster. This objects may improve the computation of iterative algorithms, by caching them in main memory. RDDs can be reconstructed from data in reliable storage when a partition is lost. When there is not enough memory in the cluster to cache all the partitions of an RDD, Spark can recompute it as soon as needed. However, this re-computation impacts negatively on the system performance.

### 3.3 Background on graph algorithms

In this section we provide background on four well-known graph algorithms. We describe the standard implementation, as well as the algebraic approach based on matrix operations. Based on this background, in Sect. 4 we will identify a common algorithmic pattern, based on certain computational similarities.

#### *3.3.1 Reachability from a source vertex*

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 [17], 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. 1

$$S_k = (E^T)^k \cdot S_0 = E^T \cdot \dots \cdot (E^T \cdot (E^T \cdot S_0)) \text{ ($k$ vector–matrix products)} \quad (1)$$

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

$$S_0[i] = 1 \text{ when } i = s, \text{ and 0 otherwise} \quad (2)$$

#### *3.3.2 Bellman–Ford: a single source shortest path algorithm*

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 [6]. A way to express Bellman–Ford with matrix–vector multiplication under the min-plus semi-ring is explained in [9]. The shortest path of length $k$ from a source vertex $s$ to every reachable $v \in E$ can be computed as:

$$S_k = (E^T)^k \cdot S_0 = E^T \cdot \dots \cdot (E^T \cdot (E^T \cdot S_0)) \text{ ($k$ vector-matrix products)} \quad (3)$$

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

$$S_0[i] = 1 \text{ when } i = s, \text{ and } \infty \text{ otherwise} \quad (4)$$

Notice that the expression to compute SSSP looks similar to the computation of reachability, but the initialization and the multiplication (min,+), are different.

#### *3.3.3 Weakly connected components (WCC)*

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 [16]. 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.

#### *3.3.4 PageRank*

PageRank [23] 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 [15]. 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 $j$th 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 Eq. 5.

$$S_k = (T'')^k \cdot S_0 \tag{5}$$

Although Eq. 5 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 5 can be expressed in terms of the sparse matrix $T$ as follows:

$$S_d = (1 - p)T \cdot S_{d-1} + (1 - p)D \cdot S_{d-1} + (p/n)A \cdot S_{d-1} \tag{6}$$

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:

$$S_d = (1 - p)T \cdot S_{d-1} + P \tag{7}$$

where every entry of the vector $P$ is equal to $p/n$. It is recommended to set $p = 0.15$ [23].

## 4 Solving graph problems with relational queries: a unified algorithm

In the previous section we introduced four graph algorithms and we showed how to express them as an iteration of matrix–vector multiplications. 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. See Table 1 for a comparison of the four algorithms.

### 4.1 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 (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 [6]. Table 2 shows examples of relational queries to compute matrix–vector multiplication under different semirings.

**Table 1** 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 $s$ | $\forall i \in V$ | $\forall i \in V$ |

**Table 2** Matrix–vector multiplication with relational queries under common semirings

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

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

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

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

### 4.2 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.

**Algorithm 1:** Graph Algorithms Evaluated with Relational Queries

**Data**: Table $E$, Table $S_0$, $\epsilon$, 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 1 is a pattern to solve several graph problems with an iteration of relational queries. We base Algorithm 1 in our previous work [4], where we expressed some graph algorithms with relational algebra. This algorithmic pattern can be applied in relational databases and array databases. 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 one column
3. The output of the query is inserted into 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 $\Delta$ (the value returned by $f$) is less than some small value $\epsilon$, or when it reaches the max number of iterations.

# 5 Graph data storage and partitioning for the unified algorithm

## 5.1 Physical storage

The graph storage in the two parallel systems (Columnar DBMS/Array DBMS) has important differences.

In the columnar DBMS, graphs are stored in a relational table containing the edge information: source vertex, destination vertex, and edge's attributes. An optional table may include vertex attributes. The computation of the algorithms relays on a projection of the edges table, table $E(i, j, v)$, with primary key $(i, j)$ and a numeric edge attribute $v$. Note that table $E$ stores the adjacency matrix of G in sparse representation.

In the array DBMS, the graph can be stored as a $n \times n$ disk array. The graph may have one or several numeric attributes. In contrast to the columnar DBMS, $i$ and $j$ are not stored, because these values are implicit, as it occurs in main memory arrays. SciDB—an open source array DBMS—does not assign physical space for empty cells, only cells with a value different than zero are stored.

As explained in Sect. 3, our algorithms require a vector which is recomputed at each iteration. In the case of the columnar DBMS, this vector is stored in table $S(i, v)$, and in the case of the array DBMS, it is stored as a uni-dimensional array.

## 5.2 Graph data partitioning for the unified algorithm

Parallel graph processing systems devise several partitioning strategies to minimize the communication between nodes and to keep the processing balanced in the parallel system. An ideally balanced processing would have an even data distribution with $m/N$ edges per node (recall that $N$ is the number of computing nodes). It is challenging to reach this objective, due to the skewed degree distribution of real graphs: vertices have high probability of a low degree, but very low probability of a high degree. Researchers [8,20] have observed that degree distribution in real-world complex networks follows the *power law*: the number of vertices of degree $k$ is approximately proportional to $k^{-\beta}$ for some $\beta > 0$. Graphs following the power law are challenging because of their strong skewness.

State-of-the-art graph systems propose two main approaches of graph partitioning:

- edge-cut: vertices are distributed evenly across the cluster; the number of edges crossing machines needs to be minimized.
- vertex-cut: edges are not broken, they do not span across machines. In this case, vertices might need to be replicated.

Gonzalez et al. [10] showed weaknesses of the edge-cut model, especially for graphs following the power law, as is the case of natural graphs. Spark-GraphX and Power-Graph apply vertex-cut partition.

In this section, we present a graph partition strategy to improve data locality in the processing of parallel join which computes matrix–vector multiplication, and to balance the data in the cluster. Note this strategy is not intended to accelerate arbitrary queries on graphs or the entire spectrum of graph algorithms, but to reduce the execution time of the algorithms of our concern: a group of graph algorithms based

**Table 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($S.i$) | $E.i$, $S.i$ |

on iterative matrix–vector multiplication. In Sect. 7 we will present an experimental evaluation of our strategy.

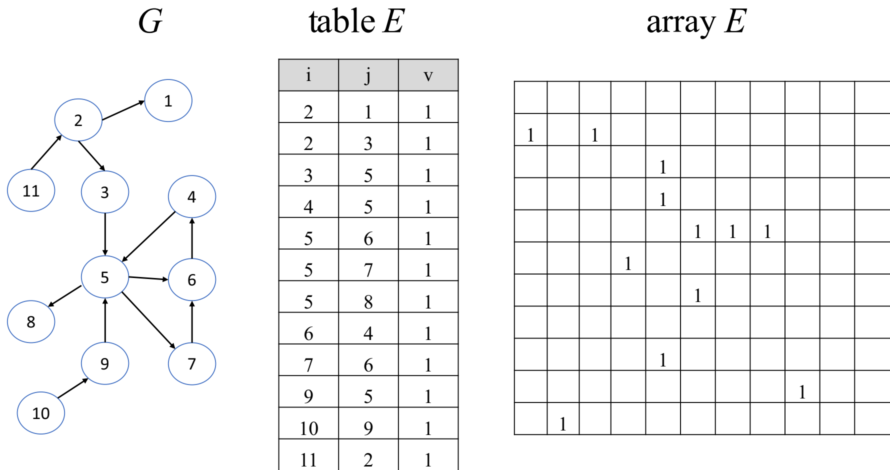
In the case of the family of algorithms studied in this work, keeping a low data transfer between nodes is critical to achieve good performance. The core computation of our algorithms is the query that computes the matrix–vector multiplication, comprised of a join and an aggregation. Moreover, because of the iterative nature of 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 3).

### *5.2.1 Partitioning in a columnar DBMS*

The illustration in Fig. 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 $i$ (source vertex) or $j$ (destination vertex). Table $S$ is clearly partitioned by its primary key, the vertex id.

Specifically, if the join condition is $E_i = S_i$ (Fig. 2a), the edges having the same **source** vertex are stored in only one computing node, along with the corresponding vertices in $S$. When the join condition is $E_j = S_i$ (Fig. 2b), the edges having the same **destination** 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. Note for the actual data distribution to the cluster we incorporate a built-in hash function, which is useful to avoid unbalanced data.

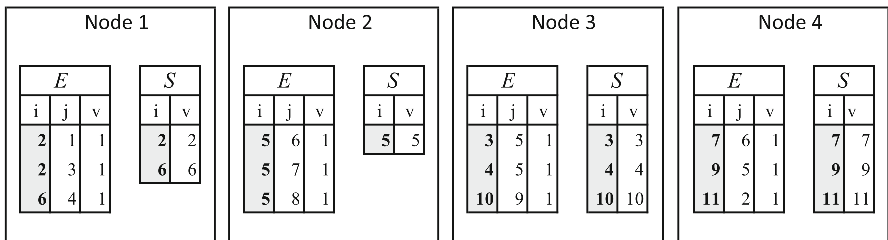
On the other hand, skewness may cause unbalanced processing. To achieve balanced data distribution we take advantage of hash function partitioning, a functionality available in some parallel DBMSs. Specifically, the partitioning $E$ is done by a hash function on the join column (either $i$ or $j$).




| 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 |

**Fig. 1** A sample graph $G$ is stored a table $E$ in the columnar DBMS or as a bi-dimensional array $E$ in the array DBMS

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




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

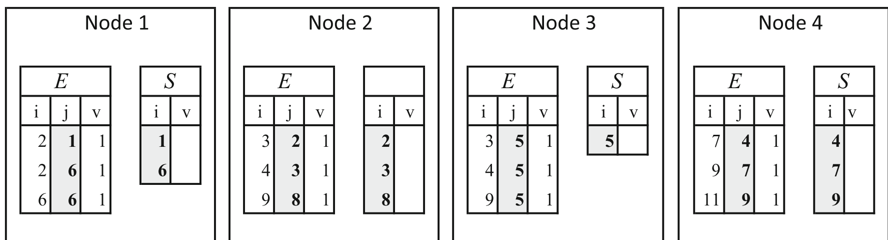



**Fig. 2** Partitioning of table $E$ in four nodes

### *5.2.2 Partitioning in an array DBMS*

In general, big-data graphs are characterized by a sparse adjacency matrix. The adjacency matrix of $E$ is stored as a bi-dimensional array, and $S$ as a unidimensional array. Arrays are blocked by chunks of homogeneous size. SciDB assigns chunks

in a rigid way: considering an array whose chunks are numbered as $1, 2, \ldots$ and stored in $N$ worker nodes, chunks are assigned to the workers just by the formula $(chunknumber \bmod N) + 1$. The strategy is the same as columnar DBMS: the parallel join $E \bowtie S$ finds matching data in the same node. To obtain this objective, we divide $S$ in $K$ chunks and $E$ in $K \times K$ chunks. $K$ is determined based on $N$; considering that SciDB distributes chunks to nodes in a round robin manner, local join is possible when $K$ is a multiple of $N$.

#### *5.2.3 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 the computing nodes 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 Algorithms expressed with queries

*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 join key matching: Rows that satisfy the join condition are always in the same computing node. This is crucial 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 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 cost 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 matrix multiplication operator; this operator internally calls the high performance linear algebra library SCALAPACK [5]. In the array DBMS, a carefully data partition let us to compute the join minimizing data transfer: cells satisfying 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 in an alternative manner, as we elaborate on Sect. 7. Like the columnar DBMS, queries in the array DBMS can be optimized to exploit:

(1) Local join key matching for parallel joins; (2) Presorted data, which is inherent of the array-based data organization.

*Spark-GraphX* stores graphs using two main data structures, namely EdgeRDD and VertexRDD, that 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 [11], 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" .

### 6.1 Computing PageRank

PageRank is simple, but it is necessary to consider carefully the relational query to avoid mistakes. Recalling Eq. 7, 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 $E \cdot j = v$. Thus a row $T \cdot i = v$ 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*. Recall that the parameter $p$ was defined in Sect. 3.3.4

```
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 AS i, E.i AS j, 1/C.cnt AS v
     FROM E,
          (SELECT i, COUNT(*) cnt
            FROM E
           GROUP BY i) C
    WHERE E.i = C.i;
```

**Algorithm 2:** PageRank

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

*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 $\epsilon$. 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;
```

*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.

### 6.2 Connected components

Our Connected Components algorithm is an improvement of HCC, an iterative algorithm proposed in [16]. The algorithm in [16] 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:

$$S_d = assign(E \cdot S_{d-1}) \tag{8}$$

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 [13] applied HCC in a RDBMS. As showed in [13] 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 [16], 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 [13], 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.

**Algorithm 3:** Connected Components

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

*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;
```

### 6.2.1 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 Pregel's message-passing abstraction.

## 6.3 Bellman ford (SSSP)

Recalling Sect 3.3.2, SSSP is computed by an iteration of matrix–vector multiplication: the transposed of the adjacency matrix multiplied by a vector which holds the "current" minimum value. From a relational point of view, the vector $S_d$ 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 $\infty$ 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 $d$th 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 the 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 $\infty$. In the database systems, those values are not stored.

*Iterations* Following our algorithmic pattern, the iterative process stops when $\Delta$ is equal or less a value $\epsilon$. The value $\Delta$ 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
  FROM E JOIN S0 ON S0.i = E.i
      GROUP BY E.j;
```

**Algorithm 4:** Single Source Shortest Path

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

*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.

### 6.4 Reachability from a source vertex

*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.

**Algorithm 5:** Reachability from a Source Vertex

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

*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.

## 7 Experimental evaluation

Our experiments have two main objectives: First, we want to evaluate our proposed optimizations in columnar and array DBMS, comparing them with some techniques for performance improvement already available in these systems. Second, we conduct experiments to compare performance and results of three graph algorithms, under three different systems: An industrial columnar database (Vertica), an open source array database (SciDb) and the well known system for Hadoop, Spark/GraphX. Considering our previous research [22], we expect that columnar and array DBMS will largely surpass the performance of row DBMS in graph problems. Moreover, superior performance of parallel columnar DBMS in analytical workloads was previously demonstrated in [24]. For this reason, we focus our experiments in columnar and array DBMSs. Vertica, SciDB and Spark-GraphX were installed in the same hardware: a four node cluster, each node with 4GB RAM and a Quad core Intel CPU 2.6 GHz. The cluster has in total 16 GB RAM and 4TB of disk storage, running Linux Ubuntu.

### 7.1 Data sets

In graph analysis, social networks and hyperlink networks are considered challenging, not only because their size, but also because the skewed distribution of the degree of the vertices of these graphs. 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 [19], one dataset from Wikipedia, and a very large web graph data set from Web Data Commons with m = 620 M [18]. 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 (Table 4).

### 7.2 Evaluation of query optimizations

As explained in Sects. 4 and 5, we introduced several optimizations: (1) exploit data locality of the join $E \bowtie S$; (2) presorted data to compute the join with merge-join

**Table 4** Datasets

| Data set | Description | $n$ | $m = \|E\|$ | Avg degree | Max degree | Max WCC |
|---|---|---|---|---|---|---|
| web-Google | Hyperlink graph | 1M | 5M | 12 | 6,353 | 855,802 |
| soc-pokec | Social network | 2M | 30M | 38 | 20,518 | 1,632,803 |
| LiveJournal | Social network | 5 M | 69M | 28 | 22,887 | 4,843,953 |
| wikipedia-en | Hyperlink graph | 13M | 378M | 62 | 963,032 | 11,191,454 |
| Web Data Commons | Hyperlink graph | 43M | 623M | 29 | 4M | 39M |

algorithm; (3) In each iteration $S$ is computed strictly with one matrix multiplication evaluated with relational queries, always joining only two tables. Our experimental validation includes both columnar and array DBMSs.

### *7.2.1 Evaluating optimizations in a columnar DBMS*

We compare the benefits of: (A) Our proposed partitioning strategy i.e., partitioning by the join key applying a built-in hash function versus (B) A classical parallel join optimization, the replication of the smaller table of the join to every node in the parallel cluster, which ensures that data of this small table is local to any node. Replication of the smaller table in a parallel join is a well known and very efficient optimization, but we show that it is not the winner in our case. To make the aforementioned comparison, we run experiments with the LiveJournal social network data set [19], and with augmented versions of this data set, increasing its size by a factor of two and three. The graph data set is *partitioned in the same way* both in A and in B. In case A, the table $S$ is partitioned by its primary key. In case B, the table $S$ (smaller than $E$) is replicated through the cluster. Note in both A and B we have defined the same ordered projections, for the best possible performance.

Figure 3 compares the execution time of the two strategies for an iterative computation of PageRank in various data set sizes. Our proposed strategy (labeled in the figure as "a") is superior than the one which replicates the smaller table to every node in the cluster (labeled as "B"). How can our optimization be better than a data replication through the cluster? Recall that the algorithms in this work require several iterations, and that the vector $S$ is recomputed at each iteration. Therefore, keeping a local copy has the disadvantage that the result table needs to be replicated to the complete cluster in every iteration. On the other hand, we run a second set of experiments to understand time complexity of algorithms, using the same LiveJournal data set. In this experiment, we increased the data set up to four times. We measured the average time to run one iteration of PageRank in the columnar database. Figure 4 shows our measurements. Every iteration runs with a time complexity close to linear.

### *7.2.2 Evaluating optimizations in an array DBMS*

In the array DBMS we apply the same strategy as in the columnar DBMS: to increase the join data locality by partitioning the data properly, as explained in Sect. 5.2.2. The

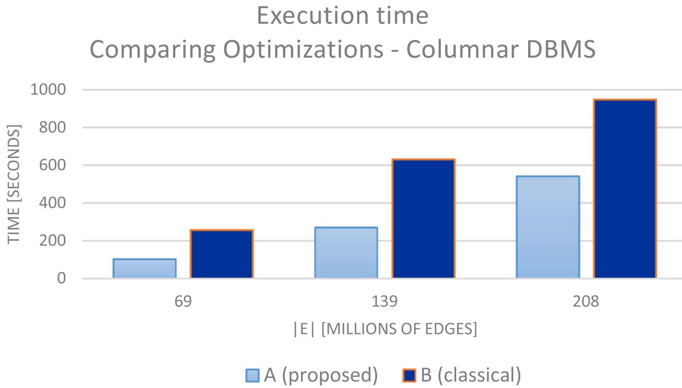



**Fig. 3** Comparing execution times in columnar DBMS: (*A*) proposed partition strategy. (*B*) Classical parallel join optimization by replication of the smaller table

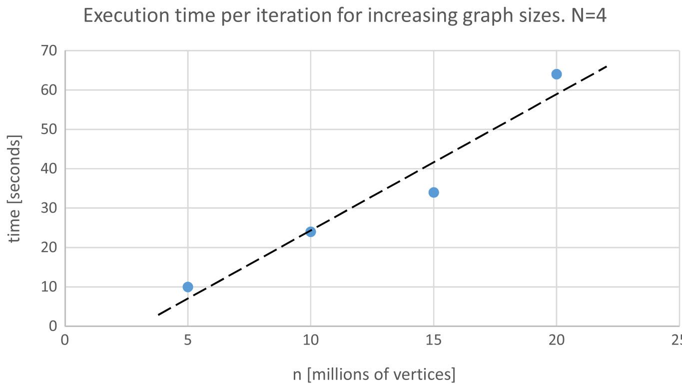



**Fig. 4** Execution time of the join-aggregation query for PageRank in columnar DBMS. Optimizations bring a performance close to linear

default way to load the array data may be sensible to skewness, as it is shown on the right side of Fig. 5. As a result, a few blocks of the matrix concentrates a large amount of data. To alleviate the problem of skewed data distribution, we load the array data in an alternative way : redefining array subscripts of the adjacency matrix. Considering that $E$ is divided in uniform squared-shaped chunks of size $h \times h$, the array subscripts are redefined as follows:

$$i' \rightarrow h * (i \bmod N) + i/N; \tag{9}$$

$$j' \rightarrow h * (j \bmod N) + j/N; \tag{10}$$

Figure 5 shows a plot of the density of the adjacency matrix using original array subscripts and redefined array subscripts. Furthermore, Fig. 6 shows the data distribution among the workers. Clearly, the redefinition of array subscripts helps to alleviate the problem of uneven partitions. To understand if the proposed strategy has a significant cost, keep in mind that in SciDB an array is loaded in two steps: Firstly, data

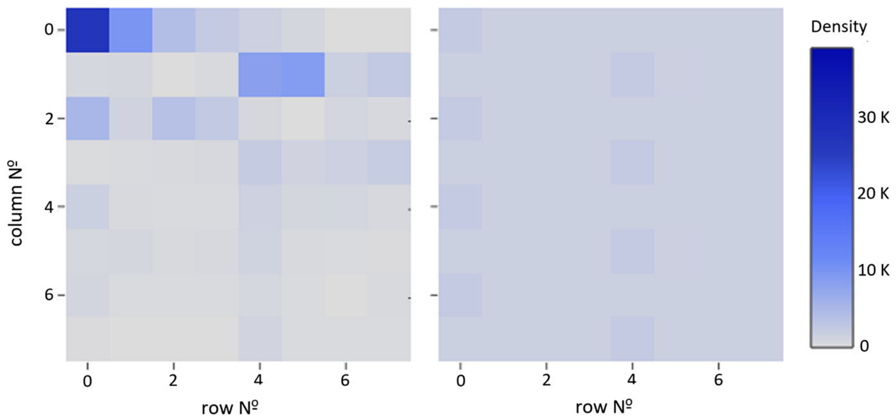



**Fig. 5** Array DBMS: Living Journal data set. The *left* heat map shows the density of the adjacency matrix with the original array subscripts. The *right* heat map shows the density of the adjacency matrix with the redefined array subscripts. The number on the axes are coordinates identifying chunks in the disk array storage. Chunk (0,0) on the *left* presents the maximum density

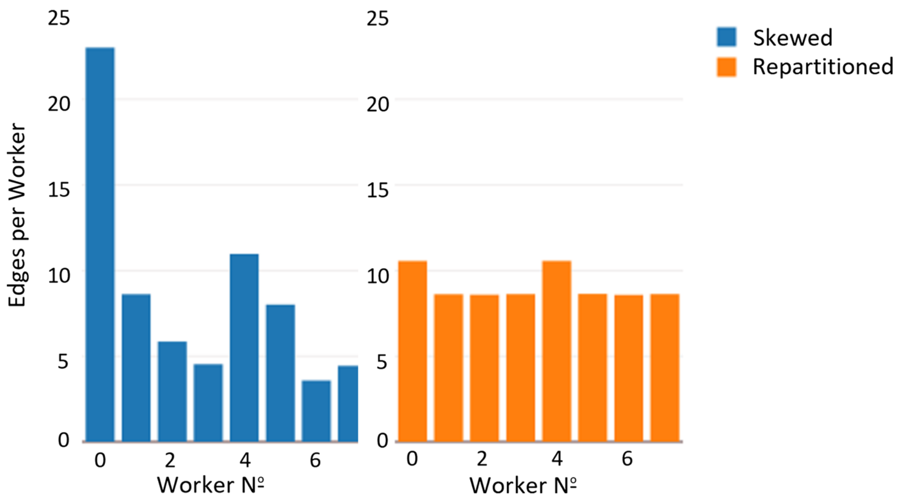



**Fig. 6** Array DBMS: Living Journal data set. Another perspective of the data density. Data distribution in cluster with original (*left*) and redefined (*right*) array subscripts. *Vertical axis* shows edges per worker. *Horizontal axis* identifies workers

is loaded to a unidimensional array, whose subscript is artificial. Second, the data is partitioned by chunk based on the actual subscripts, an operation known in SciDB as redimensioning. The proposed alternative is to perform the second step (redimensioning) based on redefined indexes instead of the original subscripts. The overhead of this alternative, compared to the default redimensioning, is just the cost of computing a projection on the original subscripts.

We want to understand experimentally the efficiency of our query-based matrix multiplication, and we contrast it to the execution time of matrix multiplication with

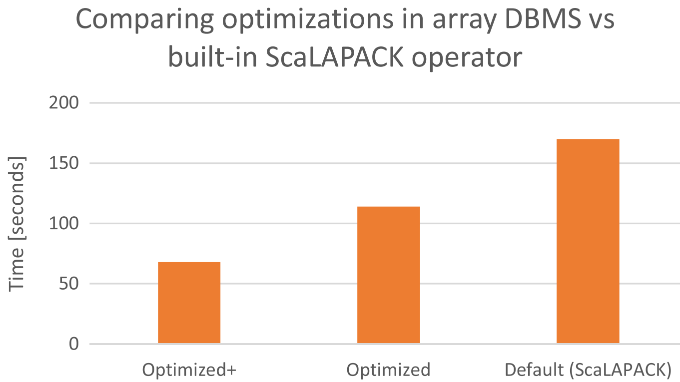



**Fig. 7** A comparison of Matrix Multiplication in SciDB. Slowest computation with the SciDB's built-in operator SpGemm. Faster computation with join and aggregation plus repartitioning

ScaLAPACK, available as a built-int SciDB operator. We partition the LiveJournal data set according to our proposed strategy (Sect. 5.2.2), and then we compute one iteration of PageRank, in three different ways:

1. Default: SciDB's built-in matrix multiplication operator SpGemm, calling ScaLAPACK;
2. Optimized: Our proposed join-aggregation query exploiting join locality.
3. Optimized+: Our join-aggregation query, plus redefinition of array subscripts to ensure even distribution of the data across the cluster.

In Fig. 7 we present the average execution time per iteration of a PageRank query, comparing (1), (2) and (3) Even though the data is partitioned to ensure join data locality, ScaLAPACK (right bar) takes the longest 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, where the execution time is improved by redefining the array subscripts when the data is loaded, which balances the computation through the cluster nodes.

### 7.3 Comparing performance in columnar DBMS, array DBMS and Spark-GraphX

Results of our experiments are presented in Table 5, as well as in Fig. 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

**Table 5** Comparing columnar DBMS versus array DBMS versus Spark-GraphX

| Algorithm | Data set | $m = \|E\|$ | 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 | 623 | 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 | 623M | 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 | 623M | 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 | 623M | 2764 | stop | crash |

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 (Fig. 9).

### 7.4 Effect of partitioning by join key to common graph queries

While partitioning the graph data set by the join key improves the performance of the algorithms of our concern, the reader may wonder about its effect to common graph queries. We present experiments with interesting queries for graph exploration: (1) in-degree i.e., the counting of incoming edges; (2) out-degree i.e., the counting of outcoming edges; (3) getting the edges greater than/lesser than a constant; (4) sink vertices (5) neighbors of a vertex $v$. We run every query both under the default partitioning (by primary key) and under join key partitioning. Execution times are presented in Table 6, where times less than one tenth of second are represented by a

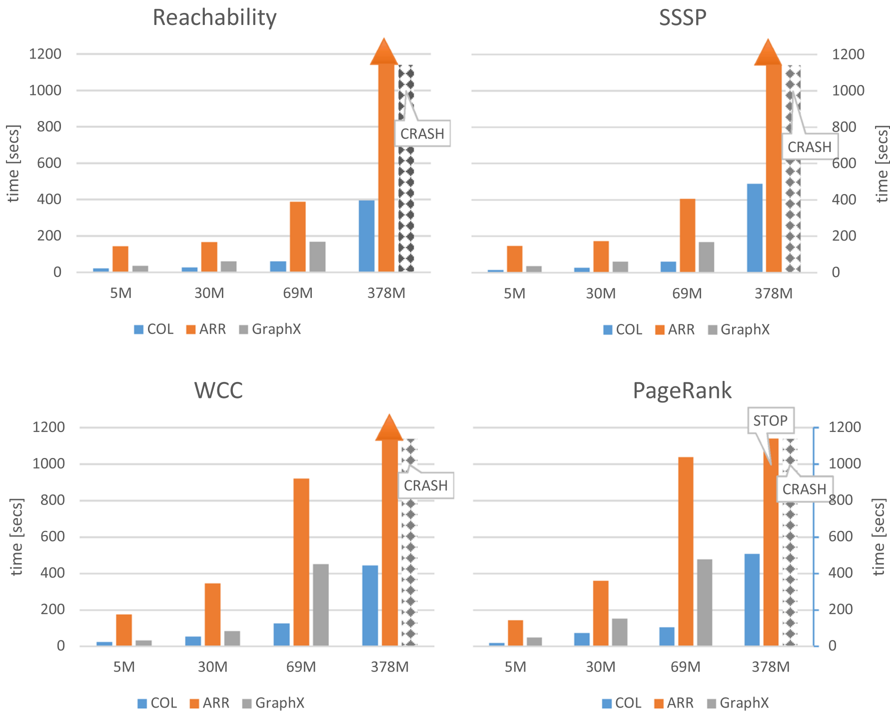



**Fig. 8** Performance comparisons: columnar DBMS, array DBMS and Spark-GraphX

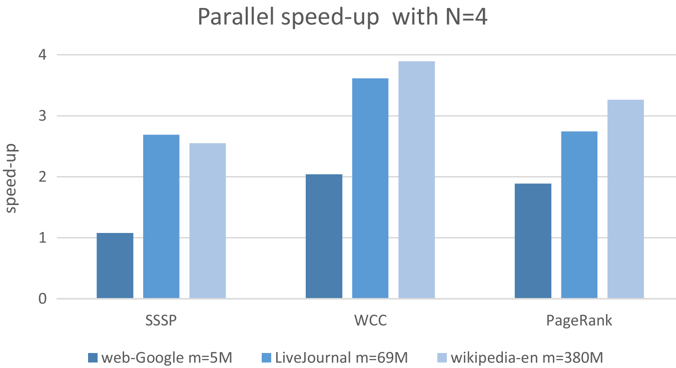



**Fig. 9** Parallel speedup for SSSP, connected components and PageRank for three data sets (4 Nodes)

small circle. We observe that our proposed partitioning does not hinder the performance of common exploratory queries.

### 7.5 Parallel speedup experimental evaluation

We present a parallel speedup 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 7. By definition, the parallel speedup

**Table 6** Comparison of execution time for exploratory queries under two different data partitioning methods: by primary key (default) and by joining key (proposed)

| Data set | $m$ | Indegree | | Outdegree | | $E \cdot v > c$ | | Sinks | | Neighbors of $v$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | PK | JK | PK | JK | PK | JK | PK | JK | PK | JK |
| web-Google | 5M | ∘ | ∘ | ∘ | ∘ | ∘ | ∘ | ∘ | ∘ | ∘ | ∘ |
| LiveJournal | 69M | ∘ | ∘ | ∘ | ∘ | 0.2 | ∘ | ∘ | ∘ | ∘ | ∘ |
| wikipedia-en | 378M | ∘ | ∘ | ∘ | ∘ | 3.6 | ∘ | ∘ | ∘ | 0.2 | ∘ |

Time in seconds. Times less than one tenth of second are represented by ∘

**Table 7** Serial versus parallel execution performance for three 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 |

is $S = t_1/t_N$. Our experiments shows that larger graph data sets benefit from parallel processing, obtaining a speedup from 2.50 up to 3.8 . In contrast, the experiments with the small data set (5 million edges) present a lower speedup. Recall that the concept behind of our algorithms is an iterative 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 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 at every iteration.

## 8 Conclusions

We demonstrated that relational queries are suitable to compute several graph algorithms, namely reachability from a source vertex, single source shortest path, weakly connected components, and PageRank. Moreover, we show a common algorithmic pattern to solve this important family of graph problems 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 on regular queries only,

avoiding UDFs or internal modifications to the database. Therefore, our optimizations are easily portable to other systems. Due to our data partitioning strategy, matching join keys are found in the same worker node, reducing data communication through the cluster. Moreover, by presorting the two joining tables, the join is solved with the merge algorithm. 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 speedup, 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. Although the array DBMS shows two/three times longer execution times than Spark-GraphX, it is more reliable when the graph data set is larger than the cluster RAM. The array DBMS built-in method to store sparse arrays—as a list of non-null values—seems to hinder performance.

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 take 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.

## References


1. Abadi, D., Boncz, P., Harizopoulos, S., Idreos, S., Madden, S., et al.: The design and implementation of modern column-oriented database systems. Found. Trends® Databases **5**(3), 197–280 (2013)
2. Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., van der Vorst, H.: Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide, vol. 11. Siam, Philadelphia (2000)
3. Bu, Y., Borkar, V., Jia, J., Carey, M.J., Condie, T.: Pregelix: big(ger) graph analytics on a dataflow engine. Proc. VLDB Endow. **8**(2), 161–172 (2014)
4. Cabrera, W., Ordonez, C.: 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, 8–10 May 2016 (2016)
5. Choi, J., Demmel, J., Dhillon, I., Dongarra, J., Ostrouchov, S., Petitet, A., Stanley, K., Walker, D., Whaley, R.: Scalapack: a portable linear algebra library for distributed memory computers-design issues and performance. Comput. Phys. Commun. **97**(1–2), 1–15 (1996)
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press, Cambridge (2009)
7. DeWitt, D., Gray, J.: Parallel database systems: the future of high performance database systems. Commun. ACM **35**(6), 85–98 (1992)
8. Faloutsos, M., Faloutsos, P., Faloutsos, C.: On power-law relationships of the internet topology. In: ACM SIGCOMM computer communication review, vol. 29, pp. 251–262. ACM (1999)
9. Fineman, J.T., Robinson, E.: Fundamental Graph Algorithms, chapter 5, pp. 45–58
10. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: Distributed graph-parallel computation on natural graphs. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, pp. 17–30, Berkeley, CA, USA, USENIX Association (2012)

11. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: Graphx: Graph processing in a distributed dataflow framework. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)
12. Idreos, S., Groffen, F., Nes, N., Manegold, S., Mullender, K., Kersten, M.: MonetDB: two decades of research in column-oriented database architectures. IEEE Data Eng. Bull. **35**(1), 40–45 (2012)
13. Jindal, A., Madden, S., Castellanos, M., Hsu, M.: Graph analytics using vertica relational database. In: 2015 IEEE International Conference on Big Data (Big Data), pp. 1191–1200. IEEE (2015)
14. Jindal, A., Rawlani, P., Wu, E., Madden, S., Deshpande, A., Stonebraker, M.: Vertexica: your relational friend for graph analytics!. Proc. VLDB Endow. **7**(13), 1669–1672 (2014)
15. Kamvar, S.D., Haveliwala, T.H., Manning, C.D., Golub, G.H.: Extrapolation methods for accelerating pagerank computations. In: Proceedings of the 12th Int. Conf. on World Wide Web, pp. 261–270. ACM (2003)
16. Kang, U., Tsourakakis, C.E., Faloutsos, C.: Pegasus: a peta-scale graph mining system implementation and observations. In: Proceedings of the 2009 Ninth IEEE International Conference on Data Mining
17. Kepner, J., Gilbert, J.: Graph algorithms in the language of linear algebra (2011)
18. Lehmberg, O., Meusel, R., Bizer, C.: Graph structure in the web: Aggregated by pay-level domain. In: Proceedings of the 2014 ACM Conference on Web Science, WebSci '14, pp. 119–128, New York, NY, USA, ACM (2014)
19. Leskovec, J., Krevl, A.: SNAP Datasets: stanford large network dataset collection (2014). http://snap.stanford.edu/data
20. Mahanti, A., Carlsson, N., Mahanti, A., Arlitt, M., Williamson, C.: A tale of the tails: power-laws in internet measurements. IEEE Netw. **27**(1), 59–64 (2013)
21. Ordonez, C.: Optimization of linear recursive queries in SQL. IEEE Trans. Knowl. Data Eng. (TKDE) **22**(2), 264–277 (2010)
22. Ordonez, C., Cabrera, W., Gurram, A.: Comparing columnar, row and array DBMSs to process recursive queries on graphs. Inf. Syst. **63**, 66–79 (2016)
23. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: bringing order to the web (1999)
24. Pavlo, A., Paulson, E., Rasin, A., Abadi, D., DeWitt, D., Madden, S., Stonebraker, M.: A comparison of approaches to large-scale data analysis. In: Proc. ACM SIGMOD Conference, pp. 165–178 (2009)
25. Qin, C., Rusu, F.: Dot-product join: an array-relation join operator for big model analytics. CoRR (2016). arXiv:1602.08845
26. Rudolf, M., Paradies, M., Bornhövd, C., Lehner, W.: Synopsys: large graph analytics in the SAP HANA database through summarization. In: First International Workshop on Graph Data Management Experiences and Systems, p. 16. ACM (2013)
27. Rusu, F., Cheng, Y.: A survey on array storage, query languages, and systems. CoRR (2013). arXiv:1302.0103
28. Soroush, E., Balazinska, M., Wang, D.: ArrayStore: a storage manager for complex parallel array processing. In: Proc. ACM SIGMOD Conference, pp. 253–264 (2011)
29. Stonebraker, M., Abadi, D., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., Zdonik, S.: C-Store: a column-oriented DBMS. In: Proc. VLDB Conference, pp. 553–564 (2005)
30. Stonebraker, M., Brown, P., Poliakov, A., Raman, S.: The architecture of SciDB. In: Proceedings of SSDBM, SSDBM'11, pp. 1–16. Springer (2011)
31. Welc, A., Raman, R., Wu, Z., Hong, S., Chafi, H., Banerjee, J.: Graph analysis: do we have to reinvent the wheel? In: First International Workshop on Graph Data Management Experiences and Systems, p. 7. ACM (2013)
32. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. HotCloud **10**(10–10), 95 (2010)