



Givens rotations for QR decomposition, SVD and PCA over database joins

Dan Olteanu¹ · Nils Vortmeier² · Đorđe Živanović³

Received: 31 January 2023 / Revised: 24 August 2023 / Accepted: 13 September 2023 / Published online: 23 November 2023
© The Author(s) 2023

Abstract

This article introduces FiGARo, an algorithm for computing the upper-triangular matrix in the QR decomposition of the matrix defined by the natural join over relational data. FiGARo's main novelty is that it pushes the QR decomposition past the join. This leads to several desirable properties. For acyclic joins, it takes time linear in the database size and independent of the join size. Its execution is equivalent to the application of a sequence of Givens rotations proportional to the join size. Its number of rounding errors relative to the classical QR decomposition algorithms is on par with the database size relative to the join output size. The QR decomposition lies at the core of many linear algebra computations including the singular value decomposition (SVD) and the principal component analysis (PCA). We show how FiGARo can be used to compute the orthogonal matrix in the QR decomposition, the SVD and the PCA of the join output without the need to materialize the join output. A suite of experiments validate that FiGARo can outperform both in runtime performance and numerical accuracy the LAPACK library Intel MKL by a factor proportional to the gap between the sizes of the join output and input.

Keywords Givens rotations · QR decomposition · SVD · PCA · Relational databases · Joins

1 Introduction

This paper revisits the fundamental problem of computing the QR decomposition: Given a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, its (thin) QR decomposition is the multiplication of the orthogonal matrix $\mathbf{Q} \in \mathbb{R}^{m \times n}$ and the upper-triangular matrix $\mathbf{R} \in \mathbb{R}^{n \times n}$ [30]. This decomposition originated seven decades ago in works by Rutishauser [48] and Francis [21]. There are three classical algorithms for QR decomposition: Gram-Schmidt and its modified version [28, 52], Householder [31], and Givens rotations [24].

QR decomposition lies at the core of many linear algebra techniques and their machine learning applications [26, 53, 55] such as the matrix (pseudo) inverse and the least

squares used in the closed-form solution for linear regression. In particular, the upper-triangular matrix \mathbf{R} shares the same singular values with \mathbf{A} ; its singular value decomposition can be used for the principal component analysis of \mathbf{A} ; it constitutes a Cholesky decomposition of $\mathbf{A}^T \mathbf{A}$, which is used for training (non)linear regression models using gradient descent [51]; the product of its diagonal entries equals (ignoring the sign) the determinant of \mathbf{A} , the product of the eigenvalues of \mathbf{A} , and the product of the singular values of \mathbf{A} .

In the classical linear algebra setting, the input matrix \mathbf{A} is fully materialized and the process that constructs \mathbf{A} is irrelevant. Our database setting is different: \mathbf{A} is not materialized and instead defined symbolically by the join of the relations in the input database. Our goal is to compute the QR decomposition of \mathbf{A} without explicitly constructing \mathbf{A} . This is desirable in case \mathbf{A} is larger than the input database. By pushing the decomposition past the join down to the input relations, the runtime improvement is proportional to the size gap between the materialized join output and the input database. This database setting has been used before for learning over relational data [40]. Joins are used to construct the training dataset from all available data sources and are unselective by design (the more labelled samples available for training, the better). This is not only the case for

✉ Dan Olteanu
olteanu@ifi.uzh.ch
Nils Vortmeier
nils.vortmeier@rub.de
Đorđe Živanović
dorde.zivanovic@cs.ox.ac.uk

¹ University of Zurich, Zurich, Switzerland

² Ruhr University Bochum, Bochum, Germany

³ University of Oxford, Oxford, UK

many-to-many joins; even key-fkey joins may lead to large (number of values in the) output, where data functionally determined by a key in a relation is duplicated in the join output as many times as this key appears in the join output. By avoiding the materialization of this large data matrix and by pushing the learning task past the joins, learning can be much faster than over the materialized matrix [40]. Prior instances of this setting include pushing sum aggregates past joins [34, 59] and the computation of query probability in probabilistic databases [41].

This article introduces FiGARo, an algorithm for computing the upper-triangular matrix \mathbf{R} in the QR decomposition of the matrix \mathbf{A} defined by the natural join of the relations in the input database. The acronym FiGARo stands for **F**actorized **G**ivens **R**otations with the letters **a** and **i** swapped.

FiGARo enjoys several desirable properties.

1. FiGARo's execution is equivalent to a sequence of Givens rotations over the join output. Yet instead of effecting the rotations individually as in the classical Givens QR decomposition, it performs fast block transformations whose effects are the same as for long sequences of rotations.
2. FiGARo takes time linear in the input data size and independently of the size of the (potentially much larger) join output for acyclic joins. It achieves this by pushing the computation past the joins.
3. Its transformations can be applied independently and in parallel to disjoint blocks of rows and also to different columns. This sets it apart from inherently sequential mainstream methods for QR decomposition of materialized matrices such as Gram-Schmidt.
4. FiGARo can outperform both in runtime performance and accuracy the LAPACK library Intel MKL by a factor proportional to the gap between the join output and input sizes, which is up to three orders of magnitude in our experiments (Sect. 9). We show this to be the case for both key - foreign key joins over two real-world databases and for many-to-many joins of one real-world and one synthetic database. We considered matrices with 2M-125M rows (84M-150M rows in the join matrix) and both narrow (up to 50 data columns) and wide (thousands of data columns). The choice of the join tree can significantly influence the performance (up to 47x). FiGARo is more accurate than MKL as it introduces far less rounding errors in case the join output is larger than the input database.

In Sect. 8, we further extended FiGARo to compute: the orthogonal matrix \mathbf{Q} in the QR decomposition of \mathbf{A} , the singular value decomposition (SVD) of \mathbf{A} , and the principal

component analysis (PCA) of \mathbf{A} . These computations rely essentially on the fast and accurate computation of the upper-triangular matrix \mathbf{R} , are done *without materializing \mathbf{A}* , can benefit from pushing past joins dot products that are computed once and reused several times, and are amenable to partial computation in case only some output vectors are needed, such as the top- k principal components. We show experimentally that these optimizations yield runtime and accuracy advantages of FiGARo over Intel MKL.

For QR decomposition, we designed an accuracy experiment of independent interest (Appendix A). The accuracy of an algorithm for QR decomposition is commonly based on how close the computed matrix \mathbf{Q} is to an orthogonal matrix. We introduce an alternative approach that allows for a fragment $\mathbf{R}_{\text{fixed}}$ of the upper-triangular matrix \mathbf{R} to be chosen arbitrarily and to serve as ground truth. Two relations are defined based on $\mathbf{R}_{\text{fixed}}$. We then compute the QR decomposition of the Cartesian product of the two relations and check how close $\mathbf{R}_{\text{fixed}}$ is to the corresponding fragment from the computed upper-triangular matrix.

Our work complements prior work on linear algebra computation powered by database engines [7, 18, 37, 49, 61] and on languages that unify linear algebra and relational algebra [16, 22, 32]. No prior work considered the interaction of QR decomposition with database joins. This requires a redesign of the decomposition algorithm from first principles. FiGARo is the first approach to take advantage of the structure and sparsity of relational data to improve the performance and accuracy of QR decomposition. The sparsity typically accounts for blocks of zeroes in the data. In our work, sparsity is more general as it accounts for blocks of arbitrary values that are repeated many times in the join output. FiGARo avoids such repetitions: It computes once over a block of values and reuses the computed result instead of recomputing it at every repetition.

We introduce FiGARo in several steps. We first explain how it works on the materialized join (Sect. 4) and then on the unmaterialized join modelled on any of its join trees (Sect. 6). FiGARo requires the computation of a batch of group-by counts over the join. This can be done in only two passes over the input data (Sect. 5). To obtain the desired upper-triangular matrix, FiGARo's output is further post-processed (Sect. 7).

This article extends a conference paper [43] with three new contributions. The extension of FiGARo to also compute: the orthogonal matrix \mathbf{Q} in the QR decomposition of \mathbf{A} , an SVD of \mathbf{A} , and the PCA of \mathbf{A} (Sect. 8). This extension was implemented and benchmarked against mainstream approaches based on Intel MKL for both runtime performance and accuracy (Sect. 9). The article also extends the introductory example (Sect. 1.1) and the related work (Sect. 10).

1.1 Givens rotations on the Cartesian product

We next showcase the main ideas behind FIGARO and start with introducing a (special case of) Givens rotation. A 2×2 Givens rotation matrix is a matrix $\mathbf{G} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$ for some angle θ (see Def. 1 for the definition of the general $d \times d$ case). If θ is selected appropriately, applying a Givens rotation introduces zeros in matrices. Consider a matrix $\mathbf{B} = \begin{bmatrix} a \\ b \end{bmatrix}$, where a, b are real numbers with $b \neq 0$. We can visualize \mathbf{B} as a vector in the two-dimensional Cartesian coordinate system. The matrix multiplication \mathbf{GB} represents the application of the rotation \mathbf{G} to \mathbf{B} : Its effect is to rotate the vector \mathbf{B} counter-clockwise through the angle θ about the origin. We can choose θ such that the rotated vector lies on the x-axis, so, having 0 as second component: If we choose θ such that $\sin \theta = -\frac{\text{sign}(a)b}{\sqrt{a^2+b^2}}$ and $\cos \theta = \frac{|a|}{\sqrt{a^2+b^2}}$ [4], then $\mathbf{GB} = \begin{bmatrix} r \\ 0 \end{bmatrix}$, where $r = \text{sign}(a)\sqrt{a^2+b^2}$.

When applied to matrices that represent the output of joins, Givens rotations can compute the QR decomposition more efficiently than for arbitrary matrices. We sketch this next for the Cartesian product of two unary relations. Consider the matrices $\mathbf{S} = \begin{bmatrix} s_1 \\ \vdots \\ s_p \end{bmatrix}$, $\mathbf{T} = \begin{bmatrix} t_1 \\ \vdots \\ t_q \end{bmatrix}$, representing unary

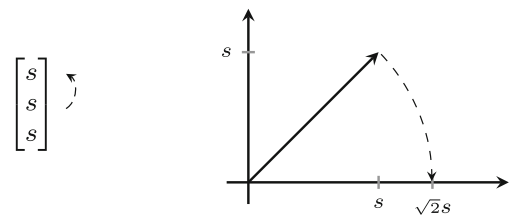
relations $S(Y_1)$ and $T(Y_2)$. The matrix $\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_p \end{bmatrix}$, where

\mathbf{A}_i is the $q \times 2$ block matrix $\begin{bmatrix} s_i & t_1 \\ \vdots & \vdots \\ s_i & t_q \end{bmatrix}$ for $i \in [p]$, represents

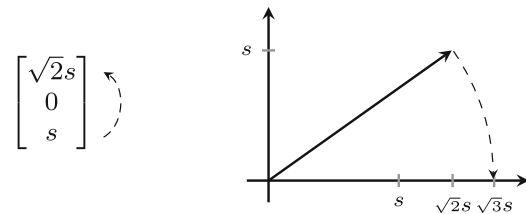
the Cartesian product of these two unary relations, so, their natural join. We would like to compute the upper-triangular matrix \mathbf{R} in the QR decomposition of \mathbf{A} : $\mathbf{A} = \mathbf{QR}$.¹

The classical Givens rotations algorithm constructs the upper-triangular matrix \mathbf{R} from \mathbf{A} by using Givens rotations to zero each cell below the diagonal in \mathbf{A} . A sequence of Givens rotations can be used to set all entries below the diagonal of any matrix \mathbf{A} to 0, thus obtaining an upper triangular matrix. The left multiplication of these rotation matrices yields the orthogonal matrix \mathbf{Q} in the QR decomposition of \mathbf{A} . This approach needs time quadratic in the input \mathbf{S} and \mathbf{T} : It involves applying $2pq - 3$ rotations, one rotation for zeroing each cell below the diagonal in \mathbf{A} . This takes $13(2pq - 3)$ multiplication, division, square, and square root operations.

¹ The matrix \mathbf{R} does not depend on the order of rows in \mathbf{A} respectively \mathbf{S} and \mathbf{T} : Consider any permutation \mathbf{P} of the rows in \mathbf{A} . Then, $\mathbf{PA} = (\mathbf{PQ})\mathbf{R}$ so the permutation only affects the orthogonal matrix \mathbf{Q} and not the upper-triangular matrix \mathbf{R} . When we go from relations to matrices, we can therefore order rows arbitrarily.



(a) Setting the entry of the second row to 0.



(b) Setting the entry of the third row to 0.

Fig. 1 Applying Givens rotations to a 1×3 matrix with all entries having the value s

In contrast, FIGARO constructs \mathbf{R} from \mathbf{S} and \mathbf{T} in linear time using $30(p + q)$ such operations.

Suppose we use Givens rotations to introduce zeros in the first column of the block $\mathbf{A}_1 = \begin{bmatrix} s_1 & t_1 \\ \vdots & \vdots \\ s_1 & t_q \end{bmatrix}$. So, we want to apply Givens rotations to a matrix column that consists of a single value and our goal is to set every occurrence of this value to 0, apart of the first one.

This setting is sketched in Fig. 1: we have a matrix that consists of a single column and all entries are equal to the same value s . Using Givens rotations, all but the first entry need to be set to 0. The first rotation is applied to the first and the second occurrence of s , so to a vector that has the value s in both components. This vector therefore has an angle of 45 degrees to the x -axis, independent of the value of s . So, the rotation angle θ is independent of s . We see this also when we calculate $\sin \theta = -\frac{\text{sign}(s)s}{\sqrt{s^2+s^2}} = -\frac{1}{\sqrt{2}}$ and $\cos \theta = \frac{|s|}{\sqrt{s^2+s^2}} = \frac{1}{\sqrt{2}}$ as explained above. The result of the rotation is that the second occurrence of s is set to 0, the first occurrence becomes $\text{sign}(s)\sqrt{s^2+s^2} = \sqrt{2}s$.

For the second rotation, we choose θ based on the value $\sqrt{2}s$ of the first entry and the value s of the third entry of the first column, with the aim of setting the latter to 0. The corresponding rotation with $\sin \theta = -\frac{1}{\sqrt{3}}$ and $\cos \theta = \frac{\sqrt{2}}{3}$ sets the first entry to $\sqrt{3}s$. This argument can be continued for larger columns: The rotation, which is used to set the k -th occurrence of s to 0, has $\sin \theta = -\frac{1}{\sqrt{k}}$ and $\cos \theta = \frac{\sqrt{k-1}}{k}$. We emphasize again that these rotations do not depend on the value s but only on the number of such values.

So, to set for the block $\mathbf{A}_1 = \begin{bmatrix} s_1 & t_1 \\ \dots & \dots \\ s_1 & t_q \end{bmatrix}$ all occurrences of the value s_1 , apart the first, to 0, we can use a series of Givens rotations with $\sin \theta = -\sqrt{\frac{1}{2}}, \dots, -\sqrt{\frac{1}{q}}$ and $\cos \theta = \sqrt{\frac{1}{2}}, \dots, \sqrt{\frac{q-1}{q}}$, respectively. The same applies to all other blocks \mathbf{A}_i as well, so the same sequence of rotation angles can also be used to introduce zeros in the first column of every matrix \mathbf{A}_i .

Insight 1 For the matrix \mathbf{A}_i , there is a sequence \mathbf{G}_s of Givens rotations that sets the first occurrence of s_i to $s_i \sqrt{q}$ and all other occurrences to 0. \mathbf{G}_s is independent of the values in \mathbf{A}_i .

The blocks \mathbf{A}_i have a second column, each consisting of the same values t_1, \dots, t_q . As the same rotations are applied

to each \mathbf{A}_i , the corresponding results $\mathbf{A}'_i = \begin{bmatrix} s_i \sqrt{q} & t'_1 \\ 0 & t'_2 \\ \dots & \dots \\ 0 & t'_q \end{bmatrix}$ have

the same values t'_j throughout all blocks. They do not depend on the values s_i , so they can be computed from \mathbf{T} alone. In Sect. 3, we show that this is possible in time $\mathcal{O}(q)$.

Insight 2 \mathbf{G}_s yields the same values t'_1, \dots, t'_q in each matrix \mathbf{A}'_i . These values can be computed once.

Among the blocks \mathbf{A}'_i , there are p occurrences of the row $[0 \ t'_j]$, for each $2 \leq j \leq q$, and p rows of the form $[s'_i \sqrt{q} \ t'_1]$. These rows can be grouped to have the same structure as the blocks \mathbf{A}_i above: the second column consists of the same value t'_j . So, rotations analogous to \mathbf{G}_s can be applied to these blocks to yield one row $[0 \ t'_j \sqrt{p}]$ and $p-1$ rows $[0 \ 0]$ for each $j \geq 2$, as well as one row $[s'_i \sqrt{q} \ t'_1 \sqrt{p}]$ and $p-1$ rows $[s'_i \sqrt{q} \ 0]$. The values s'_i can be computed from \mathbf{S} in time $\mathcal{O}(p)$ the same way as mentioned above and detailed in Sect. 3.

When all mentioned rotations are applied, we obtain the matrix \mathbf{A}'' that consists of the blocks $\mathbf{A}''_1 = \begin{bmatrix} s'_1 \sqrt{q} & t'_1 \sqrt{p} \\ 0 & t'_2 \sqrt{p} \\ \dots & \dots \\ 0 & t'_q \sqrt{p} \end{bmatrix}$

and $\mathbf{A}''_i = \begin{bmatrix} s'_i \sqrt{q} & 0 \\ 0 & 0 \\ \dots & \dots \\ 0 & 0 \end{bmatrix}$ for $2 \leq i \leq p$.

We observe the following four key points:

(1) The matrix \mathbf{A}'' has only $p+q$ nonzero values. We do not need to represent all zero rows as they are not part of the result \mathbf{R} .

- (2) The values in \mathbf{A}'' can be computed in one pass over \mathbf{S} and \mathbf{T} . We need to compute: \sqrt{p} , \sqrt{q} , and the values $s'_1, \dots, s'_p, t'_1, \dots, t'_q$.
- (3) Our linear-time computation of the nonzero entries in \mathbf{A}'' yields the same result as quadratically many Givens rotations.
- (4) We see in Sect. 3 that the computation of the values in \mathbf{A}'' does not require taking the squares of the input values, as done by the Givens QR decomposition. This means less rounding errors that would otherwise appear when squaring very large or very small input values.

So far, \mathbf{A}'' is not upper-triangular. We obtain \mathbf{R} from \mathbf{A}'' using a post-processing step that applies a sequence of $p+q-3$ rotations to zero all but the top three remaining nonzero values.

In summary, FIGARO needs $\mathcal{O}(p+q)$ time to compute \mathbf{R} in our example. Including post-processing, the number of needed squares, square roots, multiplications and divisions is at most $4(p+q)$, $3(p+q)$, $17(p+q)$ and $6(p+q)$, respectively. In contrast, the Givens QR decomposition works on the materialized Cartesian product and needs $\mathcal{O}(pq)$ time and computes $4pq$ squares, $2pq$ square roots, $16pq$ multiplications and $4pq$ divisions.

The linear-time behaviour of FIGARO holds not only for the example of a Cartesian product, but for matrices defined by any acyclic join over arbitrary relations. In contrast, the runtime of any algorithm, which works on the materialized join output, as well as its square, square root, multiplication, and division computations are proportional to the size of the join output.

2 The FIGARO algorithm: setup

Relations and Joins. A database \mathcal{D} consists of a set S_1, \dots, S_r of relations. Each relation has a schema (Z_1, \dots, Z_k) , which is a tuple of attribute names, and contains a set of tuples over this schema. We denote tuples (Z_1, \dots, Z_ℓ) of attribute names by \bar{Z} and tuples (z_1, \dots, z_ℓ) of values by \bar{z} . For every relation S_i , we denote by \bar{X}_i the tuple of its *key* (join) attributes and by \bar{Y}_i the tuple of the *data* (non-join) attributes. We allow key values of any type, e.g., categorical, while the data values are real numbers. We denote by \bar{X}_{ij} the join attributes common to both relations S_i and S_j . We consider the natural join of all relations in a database and write \bar{X} and \bar{Y} for the tuple of all key and data attributes in the join output. A database is *fully reduced* if there is no dangling input tuple that does not contribute to the join output. A join is (α) -acyclic if and only if it admits a *join tree* [1]: In a join tree, each relation is a node and there is a path between the nodes for two relations S_i and S_j if all nodes along that path correspond to relations whose keys include \bar{X}_{ij} .

From Relations to Matrices. For a natural number n , we write $[n]$ for the set $\{1, \dots, n\}$. Matrices are denoted by bold upper-case letters, (column) vectors by bold lower-case letters. Let a matrix \mathbf{A} with m rows and n columns, a row index $i \in [m]$ and a column index $j \in [n]$. Then $\mathbf{A}[i : j]$ is the entry in row i and column j , $\mathbf{A}[i :]$ is the i -th row, $\mathbf{A}[: j]$ is the j -th column of \mathbf{A} , and $|\mathbf{A}|$ is the number of rows in \mathbf{A} . For sets $I \subseteq [m]$ and $J \subseteq [n]$, $\mathbf{A}[I : J]$ is the matrix consisting of the rows and columns of \mathbf{A} indexed by I and J , respectively.

Each relation S_i is encoded by a matrix \mathbf{S}_i that consists of all rows $(\bar{x}_i, \bar{y}_i) \in S_i$. The relation representing the output of the natural join of the input relations is encoded by the matrix \mathbf{A} whose rows (\bar{x}, \bar{y}) are over the key attributes \bar{X} and data attributes \bar{Y} . We keep the keys and the column names as *contextual information* [16] to facilitate convenient reference to the data in the matrix. *For ease of presentation, in this paper we use relations and matrices interchangeably following our above mapping. We use relational algebra over matrices to mean over the relations they encode.*

We use an indexing scheme for matrices that exploits their relational nature. We refer to the matrix columns by their corresponding attribute names. We refer to the matrix rows and blocks by tuples of key and data values. Consider for example a ternary relation S with attribute names X, Y_1, Y_2 . We represent this relation as a matrix \mathbf{S} such that for every tuple $(x, y_1, y_2) \in S$ there is a matrix row $\mathbf{S}[x, y_1, y_2 :] = [x \ y_1 \ y_2]$ with entries $\mathbf{S}[x, y_1, y_2 : X] = x$, $\mathbf{S}[x, y_1, y_2 : Y_1] = y_1$, and $\mathbf{S}[x, y_1, y_2 : Y_2] = y_2$. We also use row indices that are different from the content of the row. We use $*$ to denote sets of indices. For example, $\mathbf{S}[x, * : X, Y_1]$ denotes the block that contains all rows of \mathbf{S} whose identifier starts with x , restricted to the columns X and Y_1 .

Input. The input to FiGARO consists of (1) the set of matrices $\mathbf{S}_1, \dots, \mathbf{S}_r$, one per relation in the input fully-reduced database \mathcal{D} , and (2) a join tree τ of the acyclic natural join of these matrices (or equivalently of the underlying relations).²

Output. FiGARO computes the upper-triangular matrix \mathbf{R} in the QR decomposition of the matrix block $\mathbf{A}[: \bar{Y}]$, which consists of the data columns \bar{Y} in the join of the database relations. It first computes an almost upper-triangular matrix \mathbf{R}_0 such that $\mathbf{A}[: \bar{Y}] = \mathbf{Q}_0 \mathbf{R}_0$ for an orthogonal matrix \mathbf{Q}_0 . A post-processing step (Sect. 7) further decomposes \mathbf{R}_0 : $\mathbf{R}_0 = \mathbf{Q}_1 \mathbf{R}$, where \mathbf{Q}_1 is orthogonal. Thus, $\mathbf{A}[: \bar{Y}] = (\mathbf{Q}_0 \mathbf{Q}_1) \mathbf{R}$, where the final orthogonal matrix is $\mathbf{Q} = \mathbf{Q}_0 \mathbf{Q}_1$. FiGARO does not compute \mathbf{Q} explicitly, it only computes \mathbf{R} .

² Arbitrary joins (including cyclic ones) can be supported by partial evaluation to acyclic joins: We construct a tree decomposition of the join and materialise its bags (sub-queries), cf. prior works on learning over joins, e.g., [42, 50, 51]. This partial evaluation incurs nonlinear time. This is unavoidable under widely held conjectures.

The QR decomposition always exists; whenever \mathbf{A} has full rank, and \mathbf{R} has positive diagonal values, the decomposition is unique [55, Chapter 4, Theorem 1.1].

3 Heads and tails

Given a matrix, the classical Givens QR decomposition algorithm repeatedly zeroes the values below its diagonal. Each zero is obtained by one Givens rotation. In case the matrix encodes a join output, a pattern emerges: The matrix consists of blocks representing the Cartesian products of arbitrary matrices and one-row matrices (see Fig. 2a). The effects of applying Givens rotations to such blocks are captured using so-called *heads* and *tails*. Before we introduce these notions, we recall the notion of Givens rotations.

Definition 1 (Givens rotation) A $d \times d$ Givens rotation matrix is obtained from the d -dimensional identity matrix by changing four entries: $\mathbf{G}[i : i] = \mathbf{G}[j : j] = \cos \theta$, $\mathbf{G}[i : j] = -\sin \theta$ and $\mathbf{G}[j : i] = \sin \theta$, for some pair i, j of indices and an angle θ .

$$\mathbf{G} = \begin{matrix} & & i & & j & & \\ \begin{matrix} i \\ j \end{matrix} & \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & \cos \theta & \cdots & -\sin \theta & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & \sin \theta & \cdots & \cos \theta & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \end{matrix}$$

We denote such a d -dimensional rotation matrix for given i, j and θ by $\mathbf{Giv}_d(i, j, \sin \theta, \cos \theta)$. \square

A rotation matrix is orthogonal. The result $\mathbf{B} = \mathbf{G}\mathbf{A}$ of the product of a rotation \mathbf{G} and a matrix $\mathbf{A} \in \mathbb{R}^{d \times d}$ is \mathbf{A} except for the i -th and the j -th rows, which are subject to the counterclockwise rotation through the angle θ about the origin of the 2-dimensional Cartesian coordinate system: for each column c , $\mathbf{B}[i : c] = \mathbf{A}[i : c] \cos \theta - \mathbf{A}[j : c] \sin \theta$ and $\mathbf{B}[j : c] = \mathbf{A}[i : c] \sin \theta + \mathbf{A}[j : c] \cos \theta$.

The following notions are used throughout the paper. Given matrices $\mathbf{S} \in \mathbb{R}^{m_1 \times n_1}$ and $\mathbf{T} \in \mathbb{R}^{m_2 \times n_2}$, their *Cartesian product* $\mathbf{S} \times \mathbf{T}$ is the matrix

$$\begin{bmatrix} \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_{m_1} \end{bmatrix} \in \mathbb{R}^{m_1 m_2 \times (n_1 + n_2)}$$

$$\text{with } \mathbf{A}_i = \begin{bmatrix} \mathbf{S}[i :] & \mathbf{T}[1 :] \\ \vdots & \vdots \\ \mathbf{S}[i :] & \mathbf{T}[m_2 :] \end{bmatrix}. \text{ For } \mathbf{S} \in \mathbb{R}^{1 \times n} \text{ and } \mathbf{v} \in \mathbb{R}^m, \text{ the}$$

Kronecker product $\mathbf{S} \otimes \mathbf{v}$ is $\begin{bmatrix} v[1]\mathbf{S} \\ \vdots \\ v[m]\mathbf{S} \end{bmatrix}$. We denote by $\|\mathbf{v}\|_2$

the ℓ_2 norm $\sqrt{\sum_{i=1}^m v[i]^2}$ of a vector $\mathbf{v} \in \mathbb{R}^m$.

We now define the head and tail of a matrix, which we later use to express the Givens rotations over Cartesian products of matrices.

Definition 2 (Head and Tail) Let \mathbf{A} be a matrix from $\mathbb{R}^{m \times n}$.

The *head* $\mathcal{H}(\mathbf{A}) \in \mathbb{R}^{1 \times n}$ of \mathbf{A} is the one-row matrix

$$\mathcal{H}(\mathbf{A}) = \frac{1}{\sqrt{m}} \sum_{i=1}^m \mathbf{A}[i :].$$

The *tail* $\mathcal{T}(\mathbf{A}) \in \mathbb{R}^{(m-1) \times n}$ of \mathbf{A} is the matrix (for $j \in [m-1]$)

$$\mathcal{T}(\mathbf{A})[j :] = \frac{1}{\sqrt{j+1}} \left(\sqrt{j} \mathbf{A}[j+1 :] - \frac{\sum_{i=1}^j \mathbf{A}[i :]}{\sqrt{j}} \right) \quad \square$$

Let a matrix \mathbf{A} that represents the Cartesian product of a one-row matrix \mathbf{S} and an arbitrary matrix \mathbf{T} . Then \mathbf{A} is obtained by extending each row in \mathbf{T} with the one-row \mathbf{S} , as in Fig. 2a. As exemplified in Sect. 1.1, if all but the first occurrence of \mathbf{S} in \mathbf{A} is to be replaced by zeros using Givens rotations, the specific sequence of rotations only depends on the number of rows of \mathbf{T} and not on the entries of \mathbf{S} . The result of these rotations can be described by the head and tail of \mathbf{T} .

Lemma 1 For matrices $\mathbf{S} \in \mathbb{R}^{1 \times n_1}$ and $\mathbf{T} \in \mathbb{R}^{m \times n_2}$, let $\mathbf{A} = \mathbf{S} \times \mathbf{T} \in \mathbb{R}^{m \times (n_1+n_2)}$ be their Cartesian product. Let $\mathbf{R}_i = \mathbf{Giv}_m(1, i, -\frac{1}{\sqrt{i}}, \frac{\sqrt{i-1}}{\sqrt{i}})$, for all $i \in \{2, \dots, m\}$, be a Givens rotation matrix and let $\mathbf{G} = \mathbf{R}_m \cdots \mathbf{R}_2$ be the orthogonal matrix that is the product of the rotations \mathbf{R}_m to \mathbf{R}_2 .

The matrix $\mathbf{U} = \mathbf{GA}$ obtained by applying the rotations \mathbf{R}_i to \mathbf{A} is:

$$\mathbf{U} = \begin{bmatrix} \mathcal{H}(\mathbf{A}) \\ \mathcal{T}(\mathbf{A}) \end{bmatrix} = \begin{bmatrix} \sqrt{m}\mathbf{S} & \mathcal{H}(\mathbf{T}) \\ \mathbf{0}^{(m-1) \times n_1} & \mathcal{T}(\mathbf{T}) \end{bmatrix}.$$

In other words, the application of the $m-1$ rotations to \mathbf{A} yields a matrix with four blocks: the head and tail of \mathbf{T} , \mathbf{S} scaled by the square root of the number m of rows in \mathbf{T} , and zeroes over $m-1$ rows and n_1 columns. So instead of applying the rotations, we may compute these blocks directly from \mathbf{S} and \mathbf{T} .

We also encounter cases where blocks of matrices do not have the simple form depicted in Fig. 2a, but where the multiple copies of the one-row matrix \mathbf{S} are scaled by different real numbers v_1 to v_m , as in Fig. 2b. In these cases, we cannot compute heads and tails as in Lemma 1 to capture the

effect of a sequence of Givens rotations. However, a generalized version of heads and tails can describe these effects, as defined next.

Definition 3 (Generalized Head and Tail) For any vector $\mathbf{v} \in \mathbb{R}_{>0}^m$ and matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, the *generalized head* $\mathcal{H}(\mathbf{A}, \mathbf{v}) \in \mathbb{R}^{1 \times n}$ and *generalized tail* $\mathcal{T}(\mathbf{A}, \mathbf{v}) \in \mathbb{R}^{(m-1) \times n}$ of \mathbf{A} weighted by \mathbf{v} are:

$$\mathcal{H}(\mathbf{A}, \mathbf{v}) = \frac{1}{\|\mathbf{v}\|_2} \sum_{i=1}^m v[i] \mathbf{A}[i :]$$

$$\mathcal{T}(\mathbf{A}, \mathbf{v})[j :] = \frac{1}{\|\mathbf{v}[1, \dots, j+1]\|_2} \left(\|\mathbf{v}[1, \dots, j]\|_2 \mathbf{A}[j+1 :] - \frac{\mathbf{v}[j+1] \sum_{i=1}^j v[i] \mathbf{A}[i :]}{\|\mathbf{v}[1, \dots, j]\|_2} \right) \quad \square$$

If \mathbf{v} is the vector of ones, then $\|\mathbf{v}\|_2 = \sqrt{m}$ and the generalized head and generalized tail become the head and tail, respectively.

We next generalize Lemma 1, where we consider each copy of \mathbf{S} in \mathbf{A} weighted by a positive scalar value from a weight vector \mathbf{v} : the i -th row of \mathbf{T} is appended by $v_i \mathbf{S}$, for some positive real v_i , see Fig. 2b. This scaling is expressed using the Kronecker product \otimes . Here again, to set all but the first (scaled) occurrences of \mathbf{S} to zero using Givens rotations, we use that these rotations are independent of \mathbf{S} and only depend on the scaling factors \mathbf{v} and the number of rows in \mathbf{T} . We use $\mathcal{H}(\mathbf{A}, \mathbf{v})$ and $\mathcal{T}(\mathbf{A}, \mathbf{v})$ to construct the result of applying the rotations to \mathbf{A} .

Lemma 2 Let $\mathbf{v} \in \mathbb{R}_{>0}^m$, $\mathbf{S} \in \mathbb{R}^{1 \times n_1}$, and $\mathbf{T} \in \mathbb{R}^{m \times n_2}$ be given and let $\mathbf{A} = [\mathbf{S} \otimes \mathbf{v} \mathbf{T}] \in \mathbb{R}^{m \times (n_1+n_2)}$. Let $\mathbf{R}_i = \mathbf{Giv}_m(1, i, -\frac{v[i]}{\|\mathbf{v}[1, \dots, i]\|_2}, \frac{\|\mathbf{v}[1, \dots, i-1]\|_2}{\|\mathbf{v}[1, \dots, i]\|_2})$, for all $i \in \{2, \dots, m\}$, be a Givens rotation matrix and let \mathbf{G} be the orthogonal matrix $\mathbf{G} = \mathbf{R}_m \cdots \mathbf{R}_2$.

$$\mathbf{A} = \begin{bmatrix} \mathbf{S} & \mathbf{T} \\ \vdots & \vdots \\ \mathbf{S} & \vdots \end{bmatrix} \rightsquigarrow \begin{matrix} \mathcal{H}(\mathbf{A}) = \begin{bmatrix} \sqrt{m}\mathbf{S} & \mathcal{H}(\mathbf{T}) \\ 0 & \vdots \\ 0 & \vdots \\ 0 & \vdots \end{bmatrix} \\ \mathcal{T}(\mathbf{A}) = \begin{bmatrix} \mathbf{T} \\ \vdots \\ \mathbf{T} \end{bmatrix} \end{matrix}$$

(a) Transformation of the Cartesian product of a one-row matrix \mathbf{S} and an m -row matrix \mathbf{T} .

$$\mathbf{A} = \begin{bmatrix} v_1 \mathbf{S} & \mathbf{T} \\ \vdots & \vdots \\ v_m \mathbf{S} & \vdots \end{bmatrix} \rightsquigarrow \begin{matrix} \mathcal{H}(\mathbf{A}, \mathbf{v}) = \begin{bmatrix} \|\mathbf{v}\|_2 \mathbf{S} & \mathcal{H}(\mathbf{T}, \mathbf{v}) \\ 0 & \vdots \\ 0 & \vdots \\ 0 & \vdots \end{bmatrix} \\ \mathcal{T}(\mathbf{A}, \mathbf{v}) = \begin{bmatrix} \mathbf{T} \\ \vdots \\ \mathbf{T} \end{bmatrix} \end{matrix}$$

(b) Generalization of the transformation at (a), where \mathbf{S} is scaled by a vector $\mathbf{v} = (v_1, \dots, v_m)$ of positive real numbers.

Fig. 2 Visualization of Lemmas 1 and 2: Matrices and results of applying the (generalized) head and tail

Algorithm 1: Computing generalized tails in linear time.

```

1 GENTAIL( $\mathbf{A}, \mathbf{v}$ )
2  $\mathcal{T} \leftarrow \mathbf{0}^{(m-1) \times n}$ 
3  $\text{SUM\_A} \leftarrow \mathbf{0}^{1 \times n}$ 
4  $\text{SSUM\_V} \leftarrow 0$ 
5 for  $j = 1, \dots, m-1$  do
6    $\text{SSUM\_V} \leftarrow \text{SSUM\_V} + \mathbf{v}[j]^2$ 
7    $\text{SUM\_A} \leftarrow \text{SUM\_A} + \mathbf{v}[j]\mathbf{A}[j :]$ 
8    $\mathcal{T}[j :] \leftarrow \sqrt{\text{SSUM\_V}} \mathbf{A}[j+1 :] - \frac{\mathbf{v}[j+1]\text{SUM\_A}}{\sqrt{\text{SSUM\_V}}}$ 
9    $\mathcal{T}[j :] \leftarrow \frac{\mathcal{T}[j :]}{\sqrt{\text{SSUM\_V} + \mathbf{v}[j+1]^2}}$ 
10 return  $\mathcal{T}$ 

```

The matrix $\mathbf{U} = \mathbf{GA}$ obtained by applying the rotations \mathbf{R}_i to \mathbf{A} is:

$$\mathbf{U} = \begin{bmatrix} \mathcal{H}(\mathbf{A}, \mathbf{v}) \\ \mathcal{T}(\mathbf{A}, \mathbf{v}) \end{bmatrix} = \begin{bmatrix} \|\mathbf{v}\|_2 \mathbf{S} & \mathcal{H}(\mathbf{T}, \mathbf{v}) \\ \mathbf{0}^{(m-1) \times n_1} & \mathcal{T}(\mathbf{T}, \mathbf{v}) \end{bmatrix}.$$

The generalized head and tail can be computed in linear time in the size of the input matrix \mathbf{A} .

Lemma 3 Given any matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ and vector $\mathbf{v} \in \mathbb{R}_{>0}^m$, the generalized head $\mathcal{H}(\mathbf{A}, \mathbf{v})$ and the generalized tail $\mathcal{T}(\mathbf{A}, \mathbf{v})$ can be computed in time $\mathcal{O}(mn)$.

The statement is obvious for generalized heads $\mathcal{H}(\mathbf{A}, \mathbf{v})$. We give the explanation for the tail.

Given the ℓ_2 -norm $\|\mathbf{v}[1, \dots, j]\|_2$ of the vector \mathbf{v} restricted to the first j entries, we can compute the same norm of \mathbf{v} , now restricted to the first $j+1$ entries, in constant time. Likewise, the sum $\sum_{i=1}^j \mathbf{v}[i]\mathbf{A}[i :]$ of the rows $\mathbf{v}[i]\mathbf{A}[i :]$, where i goes up to j , can be reused to compute the sum of these rows up to $j+1$ in time proportional to the row length n . See Alg. 1 for details.

The following lemma follows immediately from Def. 3.

Lemma 4 For any matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, any vector $\mathbf{v} \in \mathbb{R}_{>0}^m$, and any numbers $k \in \mathbb{R}$ and $\ell \in \mathbb{R}_{>0}$, the following holds:

$$\mathcal{H}(k\mathbf{A}, \ell\mathbf{v}) = k \mathcal{H}(\mathbf{A}, \mathbf{v})$$

$$\mathcal{T}(k\mathbf{A}, \ell\mathbf{v}) = k \mathcal{T}(\mathbf{A}, \mathbf{v})$$

4 Example: rotations on a join output

Section 1.1 shows how to transform a Cartesian product of two matrices into an almost upper-triangular matrix by applying Givens rotations. Section 3 subsequently shows how the same result can be computed in linear time from the input matrices using the new operations head and tail, whose effects on matrices of a special form are summarized in Lemmas 1 and 2.

Towards an algorithm for the general case, this section gives an example of applying Givens rotations to a matrix

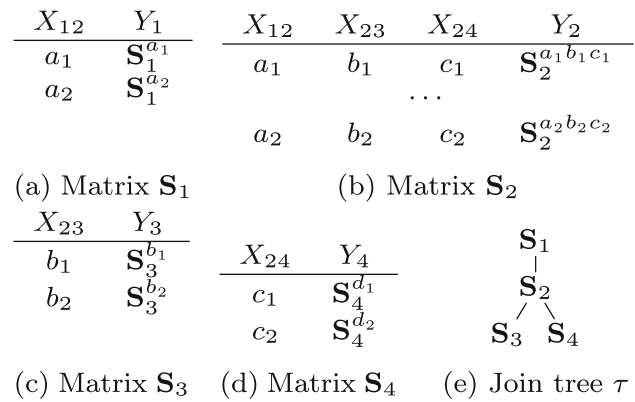


Fig. 3 Matrices and join tree used in Sect. 4

that represents the natural join of four input relations. The insights gained from this example lead to the formulation of the FIGARO algorithm.

Our approach is as follows. The result of joining two relations is a union of Cartesian products: for each join key \bar{x} , all rows from the first relation that have \bar{x} as their value for the join attributes are paired with all rows from the second relation with \bar{x} as their join value. Section 1.1 and 3 explain how to process Cartesian products. Then we take the union of their results, so, accumulate the resulting rows in a single matrix.

Figure 3 sketches the input matrices $\mathbf{S}_1, \dots, \mathbf{S}_4$ and the join tree τ used in this section. We explain \mathbf{S}_1 , the other matrices are similar. The matrix \mathbf{S}_1 in Fig. 3a has a column X_{12} that represents the key attribute common with the matrix \mathbf{S}_2 . It contains two distinct values a_1 and a_2 for its join attribute X_{12} . For each a_j , there is one vector $\mathbf{S}_1^{a_j}$ of values for the data column Y_1 in \mathbf{S}_1 . The rows of \mathbf{S}_1 are the union of the Cartesian products $[a_j] \times \mathbf{S}_1^{a_j}$, for all $j \in \{1, 2\}$.

The output \mathbf{A} of the natural join of $\mathbf{S}_1, \dots, \mathbf{S}_4$ contains the Cartesian product

$$[a_j \ b_k \ c_\ell] \times \mathbf{S}_1^{a_j} \times \mathbf{S}_2^{a_j b_k c_\ell} \times \mathbf{S}_3^{b_k} \times \mathbf{S}_4^{c_\ell}$$

for every tuple (a_j, b_k, c_ℓ) of join keys, with $j, k, \ell \in \{1, 2\}$. As mentioned in Sect. 2, we want to transform the matrix $\mathbf{A}[\bar{Y}]$, i.e., the projection of \mathbf{A} to the data columns \bar{Y} , into a matrix \mathbf{R}_0 that is almost upper-triangular. In particular, the number of rows with nonzero entries in \mathbf{R}_0 needs to be linear in the size of the input matrices. We do this by repeatedly identifying parts of the matrix $\mathbf{A}[\bar{Y}]$ that have the form depicted in Fig. 2 and by applying Lemmas 1 and 2 to these parts. Recall that each of these applications corresponds to the application of a sequence of Givens rotations.

The matrix $\mathbf{A}[\bar{Y}]$ contains the rows with the following Cartesian product associated with the join keys (a_1, b_1, c_1) :

$$\begin{array}{c} Y_1 \quad Y_2 \quad Y_3 \quad Y_4 \\ \hline \mathbf{S}_1^{a_1} \times \mathbf{S}_2^{a_1 b_1 c_1} \times \mathbf{S}_3^{b_1} \times \mathbf{S}_4^{c_1} \end{array}$$

In turn, this Cartesian product is the union of Cartesian products $[t_1 \ t_2 \ t_3] \times \mathbf{S}_4^{c_1}$, for each choice t_1, t_2, t_3 of rows of $\mathbf{S}_1^{a_1}, \mathbf{S}_2^{a_1 b_1 c_1}$ and respectively $\mathbf{S}_3^{b_1}$. The latter products have the form depicted in Fig. 2a: the Cartesian product of a one-row matrix $[t_1 \ t_2 \ t_3]$ and an arbitrary matrix $\mathbf{S}_4^{c_1}$. By applying Lemma 1 to each of them, we obtain the following rows, where we use $\alpha = \sqrt{|\mathbf{S}_4^{c_1}|}$:

$$\begin{array}{c} Y_1 \quad Y_2 \quad Y_3 \quad Y_4 \\ \hline \alpha \mathbf{S}_1^{a_1} \times \alpha \mathbf{S}_2^{a_1 b_1 c_1} \times \alpha \mathbf{S}_3^{b_1} \times \mathcal{H}(\mathbf{S}_4^{c_1}) \\ 0 \quad 0 \quad 0 \quad \mathcal{T}(\mathbf{S}_4^{c_1}) \\ \vdots \quad \vdots \quad \vdots \quad \vdots \\ 0 \quad 0 \quad 0 \quad \mathcal{T}(\mathbf{S}_4^{c_1}) \end{array}$$

This matrix contains one copy of $\mathcal{T}(\mathbf{S}_4^{c_1})$ for every choice of t_1, t_2, t_3 . More copies of $\mathcal{T}(\mathbf{S}_4^{c_1})$ arise when we analogously apply Lemma 1 to the rows of $\mathbf{A}[\cdot \tilde{Y}]$ associated with join keys (a_j, b_k, c_1) that are different from (a_1, b_1, c_1) . In total, the number of copies of $\mathcal{T}(\mathbf{S}_4^{c_1})$ we obtain is

$$\sum_{j,k \in \{1,2\}} |\mathbf{S}_1^{a_j}| \cdot |\mathbf{S}_2^{a_j b_k c_1}| \cdot |\mathbf{S}_3^{b_k}| = |\sigma_{X_{24}=c_1}(\mathbf{S}_1 \bowtie \mathbf{S}_2 \bowtie \mathbf{S}_3)|,$$

so, the size of the join of all matrices except \mathbf{S}_4 , where the attribute X_{24} is fixed to c_1 . We denote this number by $\Phi_4^\circ(c_1)$.

We apply more Givens rotations to the rows of the form $[0 \ 0 \ 0] \times \mathcal{T}(\mathbf{S}_4^{c_1})$. These are again Cartesian products, namely the Cartesian products of the zero matrix of dimension $\Phi_4^\circ(c_1) \times 3$ and each row t of $\mathcal{T}(\mathbf{S}_4^{c_1})$. We can apply again Lemma 1 to these Cartesian products and obtain the rows

$$\begin{array}{c} Y_1 \quad Y_2 \quad Y_3 \quad Y_4 \\ \hline 0 \quad 0 \quad 0 \quad \sqrt{\Phi_4^\circ(c_1)} \mathcal{T}(\mathbf{S}_4^{c_1}) \\ 0 \quad 0 \quad 0 \quad 0 \\ \vdots \quad \vdots \quad \vdots \quad \vdots \\ 0 \quad 0 \quad 0 \quad 0. \end{array}$$

The rows that only consist of zeros can be discarded, rows with nonzero entries become part of the (almost upper-triangular) result matrix \mathbf{R}_0 .

We now turn back to the remaining rows, which are of the form

$$\begin{array}{c} Y_1 \quad Y_2 \quad Y_3 \quad Y_4 \\ \hline \alpha \mathbf{S}_1^{a_1} \times \alpha \mathbf{S}_2^{a_1 b_1 c_1} \times \alpha \mathbf{S}_3^{b_1} \times \mathcal{H}(\mathbf{S}_4^{c_1}). \end{array}$$

As above, we view these rows as unions of (column-permutations of) Cartesian products $[\alpha t_1 \ \alpha t_2 \ \mathcal{H}(\mathbf{S}_4^{c_1})] \times \alpha \mathbf{S}_3^{b_1}$, for each choice t_1, t_2 of rows of $\mathbf{S}_1^{a_1}$ and $\mathbf{S}_2^{a_1 b_1 c_1}$, respectively. The corresponding applications of Lemma 1 yield the rows

$$\begin{array}{c} Y_1 \quad Y_2 \quad Y_3 \quad Y_4 \\ \hline \gamma \mathbf{S}_1^{a_1} \times \gamma \mathbf{S}_2^{a_1 b_1 c_1} \times \alpha \mathcal{H}(\mathbf{S}_3^{b_1}) \times \beta \mathcal{H}(\mathbf{S}_4^{c_1}) \\ 0 \quad 0 \quad \alpha \mathcal{T}(\mathbf{S}_3^{b_1}) \quad 0 \\ \vdots \quad \vdots \quad \vdots \quad \vdots \\ 0 \quad 0 \quad \alpha \mathcal{T}(\mathbf{S}_3^{b_1}) \quad 0 \end{array}$$

where $\beta = \sqrt{|\mathbf{S}_3^{b_1}|}$ and $\gamma = \alpha\beta$. Considering also the rows that we analogously obtain for join keys (a_j, b_1, c_ℓ) different from (a_1, b_1, c_1) , we get $\sum_{j \in \{1,2\}} |\mathbf{S}_1^{a_j}| \cdot |\mathbf{S}_2^{a_j b_1 c_1}|$ copies of $\sqrt{|\mathbf{S}_4^{c_1}|} \mathcal{T}(\mathbf{S}_3^{b_1})$ in total, and $\sum_{j \in \{1,2\}} |\mathbf{S}_1^{a_j}| \cdot |\mathbf{S}_2^{a_j b_1 c_2}|$ copies of $\sqrt{|\mathbf{S}_4^{c_2}|} \mathcal{T}(\mathbf{S}_3^{b_1})$.

We apply further Givens rotations to the rows that consist of zeros and the scaled copies of $\mathcal{T}(\mathbf{S}_3^{b_1})$. For each row t of $\mathcal{T}(\mathbf{S}_3^{b_1})$, we apply this time Lemma 2 to the matrix of all scaled copies of t and three columns of zeros. The result contains some all-zero rows and one row with entry t scaled by the ℓ_2 norm of the vector of the original scaling factors. This factor is the square root of

$$\begin{aligned} & \sum_{j \in \{1,2\}} |\mathbf{S}_1^{a_j}| \cdot |\mathbf{S}_2^{a_j b_1 c_1}| \cdot \sqrt{|\mathbf{S}_4^{c_1}|^2} \\ & + \sum_{j \in \{1,2\}} |\mathbf{S}_1^{a_j}| \cdot |\mathbf{S}_2^{a_j b_1 c_2}| \cdot \sqrt{|\mathbf{S}_4^{c_2}|^2} \\ & = \sum_{j, \ell \in \{1,2\}} |\mathbf{S}_1^{a_j}| \cdot |\mathbf{S}_2^{a_j b_1 c_\ell}| \cdot |\mathbf{S}_4^{c_\ell}| \\ & = |\sigma_{X_{23}=b_1}(\mathbf{S}_1 \bowtie \mathbf{S}_2 \bowtie \mathbf{S}_4)|, \end{aligned}$$

which, similarly as above, is the size of the join of all matrices except \mathbf{S}_3 , where the attribute X_{23} is fixed to b_1 . We denote this number by $\Phi_3^\circ(b_1)$. In summary, the result of applying Lemma 2 contains, besides some all-zero rows, the rows

$$\begin{array}{c} Y_1 \quad Y_2 \quad Y_3 \quad Y_4 \\ \hline 0 \quad 0 \quad \sqrt{\Phi_3^\circ(b_1)} \mathcal{T}(\mathbf{S}_3^{b_1}) \quad 0. \end{array}$$

The remaining columns are processed analogously. Up to now, the transformations result in all-zero rows, rows that contain the scaled tail of a part of an input matrix with other columns being 0, and one row for every join key (a_j, b_k, c_ℓ) , for $j, k, \ell \in \{1, 2\}$. We give the resulting rows for $a_j = a_1$, where for $k, \ell \in \{1, 2\}$ we use

$$\alpha_{k\ell} = \sqrt{|\mathbf{S}_1^{a_1}| \cdot |\mathbf{S}_2^{a_1 b_k c_\ell}| \cdot |\mathbf{S}_3^{b_k}| \cdot |\mathbf{S}_4^{c_\ell}|}.$$

The first column Y_1 has entries

$$\begin{array}{c} Y_1 \\ \hline \frac{\alpha_{11}}{\sqrt{|\mathbf{S}_1^{a_1}|}} \mathcal{H}(\mathbf{S}_1^{a_1}) \\ \vdots \\ \frac{\alpha_{22}}{\sqrt{|\mathbf{S}_1^{a_1}|}} \mathcal{H}(\mathbf{S}_1^{a_1}), \end{array}$$

the remaining columns are

$$\begin{array}{ccc} Y_2 & Y_3 & Y_4 \\ \frac{\alpha_{11}}{\sqrt{|S_2^{a_1 b_1 c_1}|}} \mathcal{H}(S_2^{a_1 b_1 c_1}) & \frac{\alpha_{11}}{\sqrt{|S_3^{b_1}|}} \mathcal{H}(S_3^{b_1}) & \frac{\alpha_{11}}{\sqrt{|S_4^{c_1}|}} \mathcal{H}(S_4^{c_1}) \\ \vdots & \vdots & \vdots \\ \frac{\alpha_{22}}{\sqrt{|S_2^{a_1 b_2 c_2}|}} \mathcal{H}(S_2^{a_1 b_2 c_2}) & \frac{\alpha_{22}}{\sqrt{|S_3^{b_2}|}} \mathcal{H}(S_3^{b_2}) & \frac{\alpha_{22}}{\sqrt{|S_4^{c_2}|}} \mathcal{H}(S_4^{c_2}). \end{array}$$

We observe that the entries in the column Y_1 only differ by the scaling factors $\alpha_{k\ell}$. It follows that we can apply Lemma 2, with $\mathbf{S} = \mathcal{H}(S_1^{a_1})$, \mathbf{T} consisting of the columns Y_2, Y_3, Y_4

$$\text{above, and } \mathbf{v} = \begin{bmatrix} \frac{\alpha_{11}}{\sqrt{|S_1^{a_1}|}} \\ \vdots \\ \frac{\alpha_{22}}{\sqrt{|S_1^{a_1}|}} \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{|S_1^{a_1}| \cdot |S_2^{a_1 b_1 c_1}| \cdot |S_3^{b_1}| \cdot |S_4^{c_1}|}}{\sqrt{|S_1^{a_1}|}} \\ \vdots \\ \frac{\sqrt{|S_1^{a_1}| \cdot |S_2^{a_1 b_2 c_2}| \cdot |S_3^{b_2}| \cdot |S_4^{c_2}|}}{\sqrt{|S_1^{a_1}|}} \end{bmatrix} = \begin{bmatrix} \sqrt{|S_2^{a_1 b_1 c_1}| \cdot |S_3^{b_1}| \cdot |S_4^{c_1}|} \\ \vdots \\ \sqrt{|S_2^{a_1 b_2 c_2}| \cdot |S_3^{b_2}| \cdot |S_4^{c_2}|} \end{bmatrix}.$$

The first row of the result is the generalized head of the above rows. For column Y_1 , this is $\|\mathbf{v}\|_2 \mathcal{H}(S_1^{a_1})$, where $\|\mathbf{v}\|_2$ equals $\sqrt{\sum_{k,\ell \in \{1,2\}} |S_2^{a_1 b_k c_\ell}| \cdot |S_3^{b_k}| \cdot |S_4^{c_\ell}|} = \sqrt{|\sigma_{X_{12}=a_1} S_2 \bowtie S_3 \bowtie S_4|}$. The term under the square root is the size of the join of all matrices in the subtree of S_3 in τ , where X_{12} is fixed to a_1 . We denote this number by $\Phi_3^\downarrow(a_1)$. We skip the discussion for the other columns.

The remaining rows are formed by the generalized tail. In column Y_1 , these rows only have zeros. For column Y_4 , the result is $\sqrt{|S_1^{a_1}|} \mathbf{T}_{4,a_1}$, where $\mathbf{T}_{4,a_1} = \mathcal{T} \left(\begin{bmatrix} \sqrt{|S_2^{a_1 b_1 c_1}| \cdot |S_3^{b_1}|} \mathcal{H}(S_4^{c_1}) \\ \vdots \\ \sqrt{|S_2^{a_1 b_2 c_2}| \cdot |S_3^{b_2}|} \mathcal{H}(S_4^{c_2}) \end{bmatrix}, \mathbf{v} \right)$, and analogously for the columns Y_2 and Y_3 .

Remark 1 Suppose S_1 would have another child S_5 in τ , so, S_1 would also have the join attributes X_{15} . Then we would get one scaled copy of \mathbf{T}_{4,a_1} for every value of X_{15} that occurs together with the value a_1 for X_{12} . We would then apply Lemma 2 again. The scaling factor for \mathbf{T}_{4,a_1} we obtain in the general case is the square root of the size of the join of all relations that are *not* in the subtree of S_3 in τ , where X_{12} is fixed to a_1 . In the simple case considered here, this value is $\sqrt{|S_1^{a_1}|}$.

To sum up, the resulting nonzero rows are of the following form:

1. $\sqrt{\Phi_i^\circ(\bar{x}_i)} \mathcal{T}(S_i^{\bar{x}_i})$, for all $1 \leq i \leq 4$ and values \bar{x}_i for the join attributes of S_i ,

2. $\sqrt{|S_1^{a_j}|} [\mathbf{T}_{2,a_j} \mathbf{T}_{3,a_j} \mathbf{T}_{4,a_j}]$, for $j \in \{1, 2\}$, and
3. one row of scaled generalized heads for a_1 and a_2 .

The number of rows of type (1) is bounded by the overall number of rows in the input matrices minus the number of different join keys, as the tail of a matrix has one row less than the input matrix. The number of rows of type (2) is bounded by the number of different values for the join attributes (X_{12}, X_{23}, X_{24}) minus the number of different values for X_{12} . The number of rows of type (3) is bounded by the number of different values for the join attribute X_{12} . Together, the number of nonzero rows is bounded by the overall number of rows in the input matrices, as desired.

5 Scaling factors as count queries

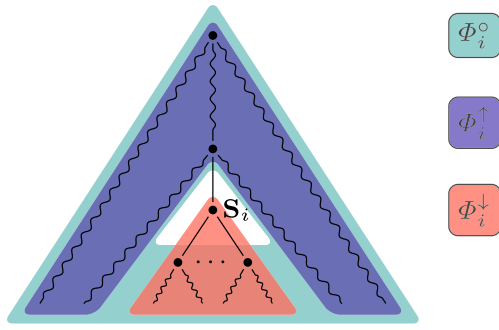
The example in Sect. 1.1 applies Givens rotations to a Cartesian product. The resulting matrix uses scaling factors that are square roots of the numbers of rows of the input matrices. These numbers can be trivially computed directly from these matrices. In case of a matrix defined by joins, as seen in the preceding Sect. 4, these scaling factors are defined by group-by count queries over the joins. FIGARO needs three types of such count queries at each node in the join tree. They can be computed together in linear time in the size of the input matrices. This section defines these queries and explains how to compute them efficiently.

We define the count queries Φ_i° , Φ_i^\uparrow and Φ_i^\downarrow for each input matrix S_i in a given join tree τ ($i \in [r]$). Figure 4 depicts graphically the meaning of these count queries: They give the sizes of joins of subsets of the input matrices, grouped by various join attributes of S_i : Φ_i^\uparrow and Φ_i^\downarrow join all matrices above and respectively below S_i in the join tree τ , while Φ_i° joins all matrices except S_i .

Each of these aggregates can be computed individually in time linear in the size of the input matrices using standard techniques for group-by aggregates over acyclic joins [42, 59]. This section shows how to further speed up their evaluation by identifying and executing common computation across all these aggregates. This reduces the necessary number of scans of each of the input matrices from $\mathcal{O}(r)$ scans to two scans.

We recall our notation: τ be the join tree; \bar{X}_i are the join attributes of S_i and \bar{X}_{ij} are the common join attributes of S_i and its child S_j in τ . We assume that: each matrix S_i is sorted on the attributes \bar{X}_i ; and a child S_j of S_i in τ is first sorted on the attributes \bar{X}_{ij} .

We first initialize the counts $\text{ROWS_PER_KEY}_i(\bar{x}_i)$ for each matrix S_i and value \bar{x}_i for its join attributes \bar{X}_i . These counts give the number of those rows in S_i that contain the join key \bar{x}_i for the columns \bar{X}_i . This step is done in one pass



$$\begin{aligned}\Phi_i^o(\bar{x}_i) &= \left| \sigma_{\bar{X}_i = \bar{x}_i} \left(\bigotimes_{j \in [r], j \neq i} S_j \right) \right| \\ \Phi_i^{\uparrow}(\bar{x}_p) &= \left| \sigma_{\bar{X}_p = \bar{x}_p} \left(\bigotimes_{\substack{j \in [r] \\ S_j \text{ is not in the subtree of } S_i}} S_j \right) \right| \\ \Phi_i^{\downarrow}(\bar{x}_p) &= \left| \sigma_{\bar{X}_p = \bar{x}_p} \left(\bigotimes_{\substack{j \in [r] \\ S_j \text{ is in the subtree of } S_i}} S_j \right) \right|\end{aligned}$$

Fig. 4 Count queries used by FIGARO, defined over the join tree τ of the r input matrices S_1, \dots, S_r . \bar{X}_i are the join attributes of S_i and \bar{X}_p are the join attributes that S_i has in common with its parent in τ . Φ_i^o gives the number of tuples in the join of all relations except S_i , grouped by the attributes \bar{X}_i . Φ_i^{\uparrow} and Φ_i^{\downarrow} give the number of tuples in the join of the relations that are in the subtree of S_i in τ respectively not in this subtree, grouped by the attributes \bar{X}_p

over the sorted matrices and takes linear time in their size. The aggregates Φ_i^{\downarrow} are computed in the same data pass. The computation of the aggregates Φ_i^o and Φ_i^{\uparrow} is done in a second pass over the matrices. We utilize the following relationships between the aggregates.

Computing Φ_i^{\downarrow} : This aggregate gives the size of the join of the matrices in the subtree of S_i in τ grouped by the join attributes \bar{X}_p that are common to S_i and its parent in τ . If S_i is a leaf in τ , then $\bar{X}_p = \bar{X}_i$ and $\Phi_i^{\downarrow}(\bar{x}_i) = \text{ROWS_PER_KEY}_i(\bar{x}_i)$. If S_i is not a leaf in τ , then we compute the intermediate aggregate

$$\Theta_i^{\downarrow}(\bar{x}_i) = \left| \sigma_{\bar{X}_i = \bar{x}_i} \left(\bigotimes_{\substack{j \in [r] \\ S_j \text{ is in the subtree of } S_i}} S_j \right) \right|$$

which gives the size of the join of the same matrices as Φ_i^{\downarrow} , but grouped by \bar{X}_i instead of \bar{X}_p . The reason for computing Θ_i^{\downarrow} is that it is useful for other aggregates as well, as discussed later. This aggregate can also be expressed as

$$\Theta_i^{\downarrow}(\bar{x}_i) = \text{ROWS_PER_KEY}_i(\bar{x}_i) \cdot \prod_{\text{child } S_j \text{ of } S_i} \Phi_j^{\downarrow}(\bar{x}_{ij}),$$

where \bar{x}_{ij} is the projection of \bar{x}_i to the join attributes \bar{X}_{ij} . The aggregate $\Phi_i^{\downarrow}(\bar{x}_p)$ is obtained by summing all values $\Theta_i^{\downarrow}(\bar{x}_i)$ such that \bar{x}_i projected to \bar{X}_p equals \bar{x}_p .

All aggregates $\Phi_1^{\downarrow}, \dots, \Phi_r^{\downarrow}$ can thus be computed in a bottom-up traversal of the join tree τ and in one pass over the sorted matrices.

Computing Φ_i^{\uparrow} : This aggregate is defined similarly to Φ_i^{\downarrow} , where the join is now over all matrices that are *not* in the subtree of S_i in the join tree τ . It is computed in a top-down traversal of τ .

Assume $\Phi_i^{\uparrow}(\bar{x}_p)$ is already computed, where \bar{x}_p is a value for the join attributes \bar{X}_p that S_i has in common with its parent in τ . The intermediate aggregate $\text{FULL_JOIN_SIZE}_i(\bar{x}_i) = \Phi_i^{\uparrow}(\bar{x}_p) \cdot \Theta_i^{\downarrow}(\bar{x}_i)$ gives the size of the join over all matrices, grouped by \bar{X}_i . For the root S_1 of τ , we set $\text{FULL_JOIN_SIZE}_1(\bar{x}_1) = \Theta_1^{\downarrow}(\bar{x}_1)$.

We next define aggregates $\text{FULL_JOIN_SIZE}_{ij}(\bar{x}_{ij})$ that give the size of the join of all matrices for each value of the join attributes \bar{X}_{ij} that are common to S_i and its child S_j . They can be computed by summing up the values $\text{FULL_JOIN_SIZE}_i(\bar{x}_i)$ for the keys \bar{x}_i that agree with \bar{x}_{ij} on \bar{X}_{ij} . Since this size is the product of $\Phi_j^{\downarrow}(\bar{x}_{ij})$ and $\Phi_j^{\uparrow}(\bar{x}_{ij})$, we have $\Phi_j^{\uparrow}(\bar{x}_{ij}) = \text{FULL_JOIN_SIZE}_{ij}(\bar{x}_{ij}) / \Phi_j^{\downarrow}(\bar{x}_{ij})$, where Φ_j^{\downarrow} is already computed. For all children S_j of S_i , we can compute all sums $\text{FULL_JOIN_SIZE}_{ij}$ together in one pass over S_i .

Computing Φ_i^o : This aggregate gives the size of the join of all matrices except S_i , grouped by the attributes \bar{X}_i . If S_i is a leaf in τ , then $\Phi_i^o = \Phi_i^{\uparrow}$ by definition. For a non-leaf S_i , we re-use the values $\text{FULL_JOIN_SIZE}_i(\bar{x}_i)$ defined above. This value is very similar to $\Phi_i^o(\bar{x}_i)$, but also depends on the size $\text{ROWS_PER_KEY}_i(\bar{x}_i)$. More precisely, we need to divide $\text{FULL_JOIN_SIZE}_i(\bar{x}_i)$ by $\text{ROWS_PER_KEY}_i(\bar{x}_i)$ to obtain $\Phi_i^o(\bar{x}_i)$.

Alg. 2 gives the procedure COMPUTE-COUNTS for shared computation of the three aggregate types. First, in a bottom-up pass of the join tree τ , it computes the aggregates Θ_i^{\downarrow} and Φ_i^{\downarrow} . Then, it computes Φ_i^{\uparrow} and Φ_i^o in a top-down pass over τ . The view FULL_JOIN_SIZE_i is defined depending on whether S_i is the root. For every child S_j , the value FULL_JOIN_SIZE_i is added to $\Phi_j^{\uparrow}(\bar{x}_{ij})$. The division by $\Phi_j^{\downarrow}(\bar{x}_{ij})$ is done just before the recursive call. Before that call, Φ_i^o is computed.

Lemma 5 *Given the matrices S_1, \dots, S_r and the join tree τ , the aggregates Φ_i^{\downarrow} , Φ_i^{\uparrow} and Φ_i^o from Fig. 4 can be computed in time $\mathcal{O}(|S_1| + \dots + |S_r|)$ and with two passes over these matrices.*

We parallelize Alg. 2 by executing the loops starting in lines 7 and 15 in parallel for different values \bar{x}_i . Atomic operations are used for the assignments in lines 13 and 22 to handle concurrent write operations of different threads.

Algorithm 2: Computing the count aggregates for FiGARo.

```

1 COMPUTE-COUNTS(matrices  $S_1, \dots, S_r$ , join tree  $\tau$ )
  ▷  $\Phi_i^\downarrow, \Phi_i^\uparrow, \Phi_i^\circ$ , ROWS_PER_KEY $_i$  and  $\Theta_i^\downarrow$  are initially empty ordered
  maps, for every matrix  $S_i$  (for the root  $S_r$  of  $\tau$ ,  $\Phi_r^\downarrow, \Phi_r^\uparrow$  do not
  exist).
2 PASS1(1)
3 PASS2(1)
4 PASS1(index  $i$ )
  ▷ Computes  $\Phi_i^\downarrow$  and intermediate maps ROWS_PER_KEY $_i$ ,  $\Theta_i^\downarrow$ .
5 foreach child  $S_j$  of  $S_i$  in  $\tau$  do
6   PASS1( $j$ )
7 foreach value  $\bar{x}_i$  of the join attributes  $\bar{X}_i$  of  $S_i$  do
8   ROWS_PER_KEY $_i(\bar{x}_i) \leftarrow |\sigma_{\bar{X}_i=\bar{x}_i} S_i|$ 
9    $\Theta_i^\downarrow(\bar{x}_i) \leftarrow \text{ROWS\_PER\_KEY}_i(\bar{x}_i)$ 
10  foreach child  $S_j$  of  $S_i$  in  $\tau$  do
11    ▷ let  $\bar{x}_{ij}$  be the projection of  $\bar{x}_i$  onto  $\bar{X}_{ij}$ , the join attributes
    shared between  $S_i$  and  $S_j$ 
12     $\Theta_i^\downarrow(\bar{x}_i) \leftarrow \Theta_i^\downarrow(\bar{x}_i) \cdot \Phi_j^\downarrow(\bar{x}_{ij})$ 
13    if  $S_i$  is not the root of  $\tau$  then
14      ▷ let  $\bar{x}_p$  be the projection of  $\bar{x}_i$  onto the join attributes shared
      between  $S_i$  and its parent in  $\tau$ 
15       $\Phi_i^\downarrow(\bar{x}_p) \leftarrow \Phi_i^\downarrow(\bar{x}_p) + \Theta_i^\downarrow(\bar{x}_i)$ 
16  PASS2(index  $i$ )
  ▷ Computes  $\Phi_i^\uparrow$  and  $\Phi_i^\circ$ .
17 foreach value  $\bar{x}_i$  of the join attributes  $\bar{X}_i$  of  $S_i$  do
18 if  $S_i$  is not the root of  $\tau$  then
19   ▷ let  $\bar{x}_p$  be the projection of  $\bar{x}_i$  onto the join attributes shared by
    $S_i$  and its parent in  $\tau$ 
20   UP_COUNT  $\leftarrow \Phi_i^\uparrow(\bar{x}_p)$ 
21 else
22   UP_COUNT  $\leftarrow 1$ 
23 FULL_JOIN_SIZE $_i(\bar{x}_i) \leftarrow \Theta_i^\downarrow(\bar{x}_i) \cdot \text{UP\_COUNT}$ 
24 foreach child  $S_j$  of  $S_i$  in  $\tau$  do
25   ▷ let  $\bar{x}_{ij}$  be the projection of  $\bar{x}_i$  onto  $\bar{X}_{ij}$ 
26    $\Phi_j^\uparrow(\bar{x}_{ij}) \leftarrow \Phi_j^\uparrow(\bar{x}_{ij}) + \text{FULL\_JOIN\_SIZE}_i(\bar{x}_i)$ 
27    $\Phi_i^\circ(\bar{x}_i) \leftarrow \frac{\text{FULL\_JOIN\_SIZE}_i(\bar{x}_i)}{\text{ROWS\_PER\_KEY}_i(\bar{x}_i)}$ 
28 foreach child  $S_j$  of  $S_i$  in  $\tau$  do
29   foreach value  $\bar{x}_{ij}$  of join attributes  $\bar{X}_{ij}$  of  $S_j$  do
30      $\Phi_j^\uparrow(\bar{x}_{ij}) \leftarrow \frac{\Phi_j^\uparrow(\bar{x}_{ij})}{\Phi_j^\downarrow(\bar{x}_{ij})}$ 
31   PASS2( $j$ )

```

6 FiGARo: pushing rotations past joins

We are now ready to introduce FiGARo. Algorithm 3 gives its pseudocode. It takes as input the matrices S_1, \dots, S_r and a join tree τ of the natural join of the input matrices. It computes an almost upper-triangular matrix \mathbf{R}_0 with at most $|S_1| + \dots + |S_r|$ nonzero rows such that \mathbf{R}_0 can be alternatively obtained from the natural join of the input matrices by a sequence of Givens rotations. It does so in linear time in the size of the input matrices. In contrast to the example in Sect. 4, FiGARo does not require the join output \mathbf{A} to be materialized. Instead, it traverses the input join tree bottom-up and computes heads

and tails of the join of the matrices under each node in the join tree.

FiGARo uses the following two high-level ideas.

First, the join output typically contains multiple copies of an input row, where the precise number of copies depends on the number of rows from other matrices that share the same join key. When using Givens rotations to set all but the first occurrence of a value to zero, this first occurrence will be scaled by some factor, which is given by a count query (Sect. 5). FiGARo applies the scaling factor directly to the value, without the detour of constructing the multiple copies and getting rid of them again.

Second, all matrix head and tail computations can be done independently on the different columns of the input matrix. FiGARo performs them on the input matrix, where these columns originate, and combines the results, including the application of scaling factors, in a bottom-up pass over the join tree τ .

The algorithm. FiGARo manages two matrices **Data** and **Out**. The matrix **Out** holds the result of FiGARo, the matrix **Data** holds (generalized) heads that will be processed further in later steps. The algorithm proceeds recursively along the join tree τ , starting from the root. It maintains the invariant that after the computation finished for a non-root node S_i of the join tree, **Data** contains exactly one row for every value of the join attributes that are shared between S_i and its parent in τ .

For a matrix S_i , FiGARo first iterates over all values \bar{x}_i of its join attributes \bar{X}_i . For each \bar{x}_i it computes head and tail of the matrix $S_i^{\bar{x}_i}$ that is obtained by selecting all rows of S_i that have the join key \bar{x}_i and then by projecting these rows onto the data columns \bar{Y}_i . The tail is multiplied with the scaling factor $\Phi_i^\circ(\bar{x}_i)$ and written to the output **Out**, adding zeros to other columns. The head is stored in the matrix **Data**. The scaling factor $\sqrt{|S_i^{\bar{x}_i}|}$ that, as per Lemma 1, is applied to other columns, is stored in the vector **scales**. If S_i is a leaf in the join tree τ , then the invariant is satisfied.

If S_i is not a leaf in the join tree τ , then the algorithm is recursively called for its children. Any output of the recursive calls is written to **Out**. The recursive call for the child S_j also returns a matrix **Data** $_j$ that contains columns \bar{Y}_k with computed heads corresponding to this column, for all k such that S_k is in the subtree of S_j in τ , as well as scaling factors **scales** $_j$. As to the maintained invariant, **Data** $_j$ contains exactly one row for every value of the join attributes \bar{X}_{ij} that are shared between S_i and S_j .

The contents of the matrices **Data** and **Data** $_j$ are joined. More precisely, for each join value \bar{x}_i of the attributes \bar{X}_i of S_i , the entry **Data** $[\bar{x}_i : \bar{Y}_k]$ is set to the entry **Data** $_j[\bar{x}_{ij} : \bar{Y}_k]$, where \bar{x}_{ij} is the projection of \bar{x}_i onto \bar{X}_{ij} . Also, scaling factors are applied to the different columns of **Data**. For the columns \bar{Y}_k , all scaling factors from **scales** and from the

Algorithm 3: FiGARO computes the almost upper-triangular matrix \mathbf{R}_0 from the input matrices $\mathbf{S}_1, \dots, \mathbf{S}_r$ with join tree τ .

```

1 FiGARO (input matrices  $\mathbf{S}_1, \dots, \mathbf{S}_r$ , join tree  $\tau$ , index  $i$ )
  ▷ Out, Data are matrices, scales is a vector. All are initially empty.
2 HEADS_AND_TAILS
3 if  $\mathbf{S}_i$  is not a leaf in  $\tau$  then
4   PROCESS_AND_JOIN_CHILDREN
5   if  $\mathbf{S}_i$  is not the root of  $\tau$  then
6     PROJECT_AWAY_JOIN_ATTRIBUTES
7   if  $\mathbf{S}_i$  is the root of  $\tau$  then
8     Out.APPEND(Data)
9     return Out
10  return (Out, Data, scales)
11 HEADS_AND_TAILS:
  ▷ Compute heads and tails of  $\mathbf{S}_i$ , grouped by the columns  $\bar{X}_i$ . Heads
  are written to Data, tails are scaled and written to Out.
12 foreach value  $\bar{x}_i$  of the join attributes  $\bar{X}_i$  in  $\mathbf{S}_i$  do
  ▷ Let  $\mathbf{S}_i^{\bar{x}_i}$  consist of all rows of  $\mathbf{S}_i$  with value  $\bar{x}_i$  for columns  $\bar{X}_i$ ,
  projected onto the data attributes  $\bar{Y}_i$ 
13   Out.APPEND( $[\mathbf{T}_1 \dots \mathbf{T}_r]$ )
14   where  $\mathbf{T}_k = \mathbf{0}$  for  $k \neq i$  and  $\mathbf{T}_i = \mathcal{T}(\mathbf{S}_i^{\bar{x}_i}) \cdot \sqrt{\Phi_i^{\circ}(\bar{x}_i)}$ 
15   Data $[\bar{x}_i : \bar{Y}_i] \leftarrow \mathcal{H}(\mathbf{S}_i^{\bar{x}_i})$ 
16   scales $[\bar{x}_i] \leftarrow \sqrt{|\mathbf{S}_i^{\bar{x}_i}|}$ 
17 PROCESS_AND_JOIN_CHILDREN:
  ▷ Recursively applies FiGARO to all children. Concatenates the
  results Out, joins the matrices Data and applies the factors scales
18 foreach child  $\mathbf{S}_j$  of  $\mathbf{S}_i$  in  $\tau$  do
19   (Out $_j$ , Data $_j$ , scales $_j$ )  $\leftarrow$  FiGARO( $\mathbf{S}_1, \dots, \mathbf{S}_r, \tau, j$ )
20   Out.APPEND(Out $_j$ )
21 foreach value  $\bar{x}_i$  of the join attributes  $\bar{X}_i$  in  $\mathbf{S}_i$  do
  ▷ let  $\bar{X}_{ij}$  be the join attributes shared between  $\mathbf{S}_i$  and  $\mathbf{S}_j$  and let  $\bar{x}_{ij}$ 
  be the projection of  $\bar{x}_i$  onto  $\bar{X}_{ij}$ .
22   foreach child  $\mathbf{S}_j$  of  $\mathbf{S}_i$  in  $\tau$  do
23     foreach  $\mathbf{S}_k$  in the subtree of  $\mathbf{S}_j$  in  $\tau$  do
24       Data $[\bar{x}_i : \bar{Y}_k] \leftarrow \mathbf{Data}_j[\bar{x}_{ij} : \bar{Y}_k] \cdot \mathbf{scales}[\bar{x}_i] \cdot$ 
        $\prod_{j' \in [r] \setminus \{j\}} \mathbf{scales}_{j'}[\bar{x}_{ij}']$ 
25       Data $[\bar{x}_i : \bar{Y}_i] \leftarrow \mathbf{Data}[\bar{x}_i : \bar{Y}_i] \cdot \prod_{j \in [r]} \mathbf{scales}_j[\bar{x}_{ij}]$ 
26       scales $[\bar{x}_i] \leftarrow \mathbf{scales}[\bar{x}_i] \cdot \prod_{j \in [r]} \mathbf{scales}_j[\bar{x}_{ij}]$ 
27 PROJECT_AWAY_JOIN_ATTRIBUTES:
  ▷ Let  $\bar{X}_p$  be the join attributes shared between  $\mathbf{S}_i$  and its parent in  $\tau$ .
  So far, Data has one row for every value of  $\bar{X}_i$ . Reduce this to one
  row for every value of  $\bar{X}_p$ .
28 foreach value  $\bar{x}_p$  of the join attributes  $\bar{X}_p$  in  $\mathbf{S}_i$  do
29   Out.APPEND( $[\mathbf{T}_1 \dots \mathbf{T}_r]$ )
30   where  $\mathbf{T}_k = \mathbf{0}$  if  $\mathbf{S}_k$  is not in the subtree of  $\mathbf{S}_i$  in  $\tau$ ; else
31    $\mathbf{T}_k = \mathcal{T}(\mathbf{Data}[\bar{x}_p, * : \bar{Y}_k], \mathbf{scales}[\bar{x}_p, *]) \cdot \sqrt{\Phi_i^{\dagger}(\bar{x}_p)}$ 
32   Data $'[\bar{x}_p :] \leftarrow \mathcal{H}(\mathbf{Data}[\bar{x}_p, * :], \mathbf{scales}[\bar{x}_p, *])$ 
33   scales $'[\bar{x}_p] \leftarrow \sqrt{\Phi_i^{\dagger}(\bar{x}_p)}$ 
34   (Data, scales)  $\leftarrow$  (Data $'$ , scales $'$ )

```

vectors **scales** $_j$ need to be applied, for all j such that \mathbf{S}_k is not in the subtree of \mathbf{S}_j . In the case of \bar{Y}_i , all factors from the vectors **scales** $_j$ are applied, but not from **scales**.

If \mathbf{S}_i is not the root of the join tree, rows of **Data** are aggregated in PROJECT_AWAY_JOIN_ATTRIBUTES to satisfy the invariant. Before entering the loop of Line 28, **Data** contains one row for every value \bar{x}_i of the join attributes \bar{X}_i of \bar{S}_i . The number of rows is reduced in PROJECT_AWAY_JOIN_ATTRIBUTES such that there is only one row for every value \bar{x}_p of the join attributes \bar{X}_p that are shared between \mathbf{S}_i and its parent in τ . To do so, FiGARO iterates over all values \bar{x}_p of these attributes. For each \bar{x}_p it takes the rows of **Data** for the keys \bar{x}_i that agree with \bar{x}_p on \bar{X}_p , and computes their generalized head and tail. The generalized tail is scaled by the factor $\sqrt{\Phi_i^{\dagger}(\bar{x}_p)}$ and written to **Out**, adding zeros for other columns. The matrix **Data** is overwritten with the collected generalized heads. So, the invariant is satisfied now. The scaling factor $\sqrt{\Phi_i^{\dagger}(\bar{x}_p)}$ equals the scaling factor that is applied to other columns as per Lemma 2, and is stored.

The next theorem states that indeed the result of FiGARO is an almost upper-triangular matrix with a linear number of nonzero rows. It also states the runtime guarantees of FiGARO.

Theorem 1 *Let the matrices $\mathbf{S}_1, \dots, \mathbf{S}_r$ represent fully reduced relations and let τ be any join tree of the natural join over these relations. Let \mathbf{A} be the matrix representing this natural join. Let M be the overall number of rows and N be the overall number of data columns in the input matrices \mathbf{S}_i .*

On input $(\mathbf{S}_1, \dots, \mathbf{S}_r, \tau, 1)$, FiGARO returns a matrix \mathbf{R}_0 in $\mathcal{O}(MN)$ time such that:

- (1) \mathbf{R}_0 has at most M rows,
- (2) there is an orthogonal matrix \mathbf{Q} such that

$$\mathbf{A}[:, \bar{Y}] = \mathbf{Q} \begin{bmatrix} \mathbf{R}_0 \\ \mathbf{0} \end{bmatrix}.$$

We can parallelize FiGARO by executing the loop starting in line 12 of Alg. 3 in parallel for different values \bar{x}_i , similarly for the loops starting in lines 28 and 21. As the executions are independent, no synchronization between the threads is necessary.

Comparison with Sect. 4. We exemplify the execution of FiGARO using the same input as in Sect. 4 and depicted in Fig. 3. We will obtain the same result \mathbf{R}_0 as in that section, modulo permutations of rows.

After the initial call of $\text{FiGARo}(\mathbf{S}_1, \dots, \mathbf{S}_4, \tau, 1)$, at first heads and tails of \mathbf{S}_1 are computed. We skip the explanation of this part and focus on the following recursive call of $\text{FiGARo}(\mathbf{S}_1, \dots, \mathbf{S}_4, \tau, 2)$. When computing the heads and tails of \mathbf{S}_2 , at first the matrices $\sqrt{\Phi_2^\circ(a_j b_k c_\ell)} \mathcal{T}(\mathbf{S}_2^{a_j b_k c_\ell})$ are padded with zeros and written to **Out**, for all $j, k, \ell \in \{1, 2\}$. Then, the entry $\mathbf{Data}[a_j b_k c_\ell : Y_3]$ is set to $\mathcal{H}(\mathbf{S}_2^{a_j b_k c_\ell})$, $\mathbf{scales}[a_j b_k c_\ell]$ becomes $\sqrt{|\mathbf{S}_2^{a_j b_k c_\ell}|}$.

Next, the children \mathbf{S}_3 and \mathbf{S}_4 of \mathbf{S}_2 in τ are processed and the intermediate results are joined. The recursive calls for the children return $\mathbf{Data}_3[b_k : Y_3] = \mathcal{H}(\mathbf{S}_3^{b_k})$ and $\mathbf{scales}_3[b_k] = \sqrt{|\mathbf{S}_3^{b_k}|}$ as well as $\mathbf{Data}_4[c_\ell : Y_4] = \mathcal{H}(\mathbf{S}_4^{c_\ell})$ and $\mathbf{scales}_4[c_\ell] = \sqrt{|\mathbf{S}_4^{c_\ell}|}$; the matrices **Out**₃, **Out**₄ contain as nonzero entries $\sqrt{\Phi_3^\circ(b_k)} \mathcal{T}(\mathbf{S}_3^{b_k})$ and $\sqrt{\Phi_4^\circ(c_\ell)} \mathcal{T}(\mathbf{S}_4^{c_\ell})$, respectively, for every $k, \ell \in \{1, 2\}$.

After joining the results and applying the scaling factors, the rows $\mathbf{Data}[a_j b_k c_\ell :]$ are as follows for every $j, k, \ell \in \{1, 2\}$, with $\beta_{j k \ell} = \sqrt{|\mathbf{S}_2^{a_j b_k c_\ell}| \cdot |\mathbf{S}_3^{b_k}| \cdot |\mathbf{S}_4^{c_\ell}|}$:

$$\frac{Y_2}{\frac{\beta_{j k \ell}}{\sqrt{|\mathbf{S}_2^{a_j b_k c_\ell}|}} \mathcal{H}(\mathbf{S}_2^{a_j b_k c_\ell})} \quad \frac{Y_3}{\frac{\beta_{j k \ell}}{\sqrt{|\mathbf{S}_3^{b_k}|}} \mathcal{H}(\mathbf{S}_3^{b_k})} \quad \frac{Y_4}{\frac{\beta_{j k \ell}}{\sqrt{|\mathbf{S}_4^{c_\ell}|}} \mathcal{H}(\mathbf{S}_4^{c_\ell})}$$

It holds $\mathbf{scales}[a_j b_k c_\ell] = \beta_{j k \ell}$.

Notice the similarity to the corresponding rows from Sect. 4. There we obtained, for $j = 1$ and with $\alpha_{k \ell} = \sqrt{|\mathbf{S}_1^{a_1}| \cdot |\mathbf{S}_2^{a_1 b_k c_\ell}| \cdot |\mathbf{S}_3^{b_k}| \cdot |\mathbf{S}_4^{c_\ell}|}$, rows that have a column Y_1 consisting of $\frac{\alpha_{k \ell}}{\sqrt{|\mathbf{S}_1^{a_1}|}} \mathcal{H}(\mathbf{S}_1^{a_1})$, and the columns

$$\frac{Y_2}{\frac{\alpha_{k \ell}}{\sqrt{|\mathbf{S}_2^{a_1 b_k c_\ell}|}} \mathcal{H}(\mathbf{S}_2^{a_1 b_k c_\ell})} \quad \frac{Y_3}{\frac{\alpha_{k \ell}}{\sqrt{|\mathbf{S}_3^{b_k}|}} \mathcal{H}(\mathbf{S}_3^{b_k})} \quad \frac{Y_4}{\frac{\alpha_{k \ell}}{\sqrt{|\mathbf{S}_4^{c_\ell}|}} \mathcal{H}(\mathbf{S}_4^{c_\ell})}.$$

For $j = 1$, the only difference in the columns Y_2 to Y_4 is in the scaling factors $\alpha_{k \ell}$ and $\beta_{1 k \ell}$, which only differ by the factor $\sqrt{|\mathbf{S}_1^{a_1}|}$.

The join attributes X_{23} and X_{24} are then projected away, as they do not appear in the parent \mathbf{S}_1 of \mathbf{S}_2 . Observe that the vectors that are used for defining the generalized heads and tails are the same here and in Sect. 4. For example, the part of **scales** that corresponds to $a_j = a_1$

$$\text{equals } \mathbf{v} = \begin{bmatrix} \sqrt{|\mathbf{S}_2^{a_1 b_1 c_1}| \cdot |\mathbf{S}_3^{b_1}| \cdot |\mathbf{S}_4^{c_1}|} \\ \vdots \\ \sqrt{|\mathbf{S}_2^{a_1 b_2 c_2}| \cdot |\mathbf{S}_3^{b_2}| \cdot |\mathbf{S}_4^{c_2}|} \end{bmatrix}, \text{ the vector used in}$$

Sect. 4. So, the generalized heads and tails for the rows of **Data** associated with join keys a_1 and a_2 , respectively, have the same results as in Sect. 4 for the columns Y_2 to Y_4 , modulo the factor $\sqrt{|\mathbf{S}_1^{a_1}|}$. This factor equals $\sqrt{\Phi_2^\uparrow(a_1)}$, which FiGARo applies to the generalized tails. That factor

is also applied to the generalized heads in **Data**, namely by the procedure `PROCESS_AND_JOIN_CHILDREN` in the call $\text{FiGARo}(\mathbf{S}_1, \dots, \mathbf{S}_4, \tau, 1)$, after the call of FiGARo for \mathbf{S}_2 terminates. There, also the column Y_1 is added. The rows in the final result of $\text{FiGARo}(\mathbf{S}_1, \dots, \mathbf{S}_4, \tau, 1)$ are then the same as in Sect. 4.

7 Post-processing

The output of FiGARo is so far not an upper-triangular matrix, but a matrix \mathbf{R}_0 consisting of linearly many rows in the size of the input matrices. The transformation of \mathbf{R}_0 into the final upper-triangular matrix \mathbf{R} is the task of a post-processing step that we describe in this section. This step can be achieved by general techniques for QR factorization, such as Householder transformations and textbook Givens rotations approaches. The approach we present here exploits the structure of \mathbf{R}_0 , which has large blocks of zeros.

The nonzero blocks of \mathbf{R}_0 are either tails of matrices (see line 13 of Alg. 3), generalized tails of sets of matrices (line 29), or the content of the matrix **Data** at the end of the execution of FiGARo (line 8). In a first post-processing step, these blocks are upper-triangularized individually and potentially arising all-zero rows are discarded. In a second post-processing step, the resulting block of rows is transformed into the upper-triangular matrix \mathbf{R} .

In both steps we need to make blocks of rows upper-triangular, i.e., we need to compute their QR decomposition. This can be done using off-the-shelf Householder transformations. We implemented an alternative method dubbed THIN [12, 27]. It first divides the rows of a block among the available threads, each thread then applies a sequence of Givens rotations to bring its share of rows into upper-triangular form. Then THIN uses a classical parallel Givens rotations approach [26, p. 257] on the collected intermediate results to obtain the final upper-triangular matrix.

All approaches need time $\mathcal{O}(MN^2)$, where M is the number of rows and N is the number of columns of the input \mathbf{R}_0 to post-processing. While the main part of FiGARo works in linear time in the overall number of rows and columns of the input matrices (Theorem 1), post-processing is only linear in the number of rows.

8 From R to Q, SVD and PCA

The previous sections focus on computing the upper-triangular matrix $\mathbf{R} \in \mathbb{R}^{n \times n}$ in the QR decomposition of the matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ representing the join output with m tuples and n data attributes. Using \mathbf{R} and the *non-materialized* join matrix \mathbf{A} , this section shows how to compute the orthogonal matrix \mathbf{Q} in the QR decomposition of \mathbf{A} , the singular

value decomposition of \mathbf{A} , and the principal component analysis of \mathbf{A} .

8.1 The orthogonal matrix \mathbf{Q}

Using $\mathbf{A} = \mathbf{Q}\mathbf{R}$, we can compute the orthogonal matrix $\mathbf{Q} \in \mathbb{R}^{m \times n}$ as follows: $\mathbf{Q} = \mathbf{A}\mathbf{R}^{-1}$. The upper triangular matrix \mathbf{R} admits an inverse, which is also upper triangular, in case the values along the diagonal are nonzero. To compute \mathbf{Q} , we do not need to materialize \mathbf{A} ! Each row in \mathbf{A} represents one tuple in the join result. We can enumerate the rows in \mathbf{A} without materializing \mathbf{A} using factorization techniques from prior work [42, 44]. This enumeration has the delay constant in the size of the input database and linear in the number n of the data attributes. The delay is the maximum of three times: the time to output the first tuple, the time between outputting one tuple and outputting the next tuple, and the time to finish the enumeration after the last tuple was outputted. Before the enumeration starts, we calibrate the input relations by removing the dangling tuples, i.e., tuples that do not contribute to any output tuple. For acyclic joins (as considered in this article), this calibration takes time linear in the size of the input database [1]. To enumerate, we keep an iterator over the tuples of each relation. We construct an output tuple in a top-down traversal of the join tree, where for each tuple at the iterator of a relation P we consider the possible matching tuples at the iterators of the relations that are children of P in the join tree.

Let $\mathbf{R}^{-1} = [\mathbf{r}_1 \cdots \mathbf{r}_n]$, where $\mathbf{r}_1, \dots, \mathbf{r}_n$ are column vectors. Let the rows in \mathbf{A} be $\mathbf{t}_1, \dots, \mathbf{t}_m$. The cell $\mathbf{Q}(i, j)$ is the dot product $\langle \mathbf{t}_i, \mathbf{r}_j \rangle$. Once \mathbf{R}^{-1} is computed in $O(n^3)$ time, we can enumerate the cells in \mathbf{Q} with delay constant in the size of the input database and linear in the number n of data attributes. To construct the entire matrix \mathbf{Q} , it then takes $O(mn^2)$ time.

We further use an optimization that pushes the computation of \mathbf{Q} past the enumeration of the join output to reduce the complexity from $O(mn^2)$ to $O(mn\ell)$, where ℓ is the number of relations in the database. The idea is to compute dot products of each tuple in each relation with selected fragments of each column vector in \mathbf{R}^{-1} . Such dot products can then be reused in the computation of many cells in \mathbf{Q} , as explained next.

We assign a unique index $i \in [n]$ to each of the n data attributes. Let I be the set of indices of the data attributes in an input relation. For each tuple \mathbf{s} in that relation, we compute the dot products $\langle \mathbf{s}(I), \mathbf{r}_1(I) \rangle, \dots, \langle \mathbf{s}(I), \mathbf{r}_n(I) \rangle$, where $\mathbf{s}(I)$ is the vector of the values for data attributes in \mathbf{s} and $\mathbf{r}_i(I)$ is the vector of those values in \mathbf{r}_i at indices in I . Let us denote these dot products computed using \mathbf{s} as $s.d_1, \dots, s.d_n$. The cell (i, j) in \mathbf{Q} is then the sum of the j -th dot products for the input tuples $\mathbf{s}_1, \dots, \mathbf{s}_\ell$ that make up the i -th output tuple in the enumeration: $\mathbf{Q}(i, j) = \sum_{k \in [\ell]} \mathbf{s}_k.d_j$.

In case of categorical data attributes, such dot products can be performed even faster. Assume a data attribute X with k distinct categories, and a tuple \mathbf{s} that has at position i the ℓ -th value of X . If we were to one-hot encode X in \mathbf{s} , then \mathbf{s} would include a vector \mathbf{v} of k values in place of this category, where $\mathbf{v}(\ell) = 1$ and $\mathbf{v}(\ell') = 0$ for $\ell' \in [k], \ell' \neq \ell$. Given a k -dimensional vector \mathbf{r} , which is part of a vector in \mathbf{R}^{-1} , the dot product $\langle \mathbf{v}, \mathbf{r} \rangle$ is then $\mathbf{r}(\ell)$. This can be achieved even without an explicit one-hot encoding of X , the index ℓ suffices.

8.2 Singular value decomposition

The singular value decomposition of $\mathbf{A} \in \mathbb{R}^{m \times n}$ is given by $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$, where $\mathbf{U} \in \mathbb{R}^{m \times n}$ is the orthogonal matrix of left singular vectors, $\Sigma \in \mathbb{R}^{n \times n}$ is the diagonal matrix with the singular values along the diagonal, and $\mathbf{V} \in \mathbb{R}^{n \times n}$ is the orthogonal matrix of right singular vectors. We can compute the SVD of \mathbf{A} without materializing \mathbf{A} using FiGARo as follows.

The first step computes the upper triangular matrix $\mathbf{R} \in \mathbb{R}^{n \times n}$ in the QR decomposition of \mathbf{A} : $\mathbf{A} = \mathbf{Q}\mathbf{R}$.

The second step computes the SVD of \mathbf{R} as: $\mathbf{R} = \mathbf{U}_R \Sigma_R \mathbf{V}_R^T$, where $\mathbf{U}_R, \Sigma_R, \mathbf{V}_R \in \mathbb{R}^{n \times n}$. There are several off-the-shelf SVD algorithms [11], we used the divide-and-conquer algorithm [29] as this was numerically the most accurate in our experiments. Note that computing the SVD takes time $O(n^3)$ for $\mathbf{R} \in \mathbb{R}^{n \times n}$ and $O(mn^2)$ for $\mathbf{A} \in \mathbb{R}^{m \times n}$. The former computation time is much less than the latter, since n is the number of data columns in the database, whereas $m \gg n$ is the size of the join output.

The SVD of \mathbf{A} can then be computed using the SVD of \mathbf{R} as follows: $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T = \mathbf{Q}\mathbf{R} = \mathbf{Q}\mathbf{U}_R \Sigma_R \mathbf{V}_R^T$, where $\mathbf{U} = \mathbf{Q}\mathbf{U}_R$ is orthogonal as it is the multiplication of two orthogonal matrices, $\Sigma = \Sigma_R$, and $\mathbf{V} = \mathbf{V}_R$. This means that the singular values of \mathbf{R} , as given by Σ_R , are also the singular values of \mathbf{A} . The right singular vectors of \mathbf{A} are also those of \mathbf{R} , as given by \mathbf{V}_R .

The third and final step computes the left singular vectors of \mathbf{A} :

$$\begin{aligned} \mathbf{U} &= \mathbf{Q}\mathbf{U}_R = \mathbf{A}\mathbf{R}^{-1}\mathbf{U}_R = \mathbf{A}(\mathbf{U}_R \Sigma_R \mathbf{V}_R^T)^{-1}\mathbf{U}_R \\ &= \mathbf{A}\mathbf{V}_R \Sigma_R^{-1} \mathbf{U}_R^{-1} \mathbf{U}_R = \mathbf{A}\mathbf{V}_R \Sigma_R^{-1}, \end{aligned}$$

where we use that $\mathbf{U}_R^{-1}\mathbf{U}_R = \mathbf{I}_n$. The matrices \mathbf{V}_R and Σ_R are as computed in the previous step. The inverse Σ_R^{-1} of the diagonal matrix Σ_R is a diagonal matrix that has the diagonal values $\sigma_{i,i}^{-1}$ corresponding to the diagonal values $\sigma_{i,i}$ in Σ_R . The multiplication $\mathbf{S} = \Sigma_R^{-1}\mathbf{V}_R$ takes time $O(n^2)$. We are then left to perform the multiplication of the non-materialized matrix \mathbf{A} with the $n \times n$ matrix \mathbf{S} , for which we proceed as for computing \mathbf{Q} in Sect. 8.1.

8.3 Principal component analysis

Following standard derivations, we can compute the principal components (PCs) of \mathbf{A} using the computation of the SVD of \mathbf{A} explained in Sect. 8.2.

We compute the eigenvalues and eigenvectors of

$$\mathbf{A}^T \mathbf{A} = (\mathbf{U} \Sigma \mathbf{V}^T)^T \mathbf{U} \Sigma \mathbf{V}^T = \mathbf{V} \Sigma \mathbf{U}^T \mathbf{U} \Sigma \mathbf{V}^T = \mathbf{V} \Sigma^2 \mathbf{V}^T$$

The sought eigenvalues and their corresponding eigenvectors are the squares of the singular values and, respectively, the right singular vectors of \mathbf{A} , or equivalently of the upper-triangular matrix \mathbf{R} computed by FiGARo. The PCs of \mathbf{A} are these right singular vectors in $\mathbf{V} = \mathbf{V}_R$. It takes $O(n^3)$ to compute all these PCs from \mathbf{R} ; in contrast, it takes $O(mn^2)$ to compute them directly from \mathbf{A} .

A truncated SVD of \mathbf{A} is $\mathbf{U}_{:,1:k} \Sigma_{1:k,1:k} \mathbf{V}_{:,1:k}^T$, where we only keep the top- k largest singular values and their corresponding left and right singular vectors. This defines a k -dimensional linear projection of \mathbf{A} :

$$\mathbf{A} \mathbf{V}_{:,1:k} = \mathbf{U}_{:,1:k} \Sigma_{1:k,1:k} \mathbf{V}_{:,1:k}^T \mathbf{V}_{:,1:k} = \mathbf{U}_{:,1:k} \Sigma_{1:k,1:k}.$$

Among all possible sets of k n -dimensional vectors, the vectors in $\mathbf{V}_{:,1:k}$, which are the top- k PCs of \mathbf{A} , preserve the maximal variance in \mathbf{A} . Equivalently, they induce the lowest error $\|\mathbf{A} - \mathbf{A} \mathbf{V}_{:,1:k} \mathbf{V}_{:,1:k}^T\|_2$ for reconstructing \mathbf{A} as a k -dimensional linear projection (Eckart-Young-Mirsky theorem [17, 39]). Applications of PCA require the top- k PCs for some value of k and sometimes also the k -dimensional linear projection of \mathbf{A} .

9 Experiments

We evaluate the runtime performance and accuracy of FiGARo against Intel MKL 2021.2.0 using the MKL C++ API (MKL called from numpy is slower) and our custom implementations. We also benchmarked OpenBLAS 0.13.3 called from numpy 1.22.0 built from source, but it was consistently slower (1.5x) than MKL, so we do not report its performance further. Both MKL and OpenBLAS implement the Householder algorithm for the QR decomposition of dense matrices [31]. We also benchmarked THIN as a standalone algorithm to compute \mathbf{R} (Sect. 7).

We use the following naming conventions: FiGARo-THIN and FiGARo-MKL are FiGARo with THIN and MKL post-processing, respectively. In the plots, we also precede the system names by SVD or PCA to denote their versions that compute SVD or PCA respectively. FiGARo and THIN use row-major order for storing matrices, while MKL uses

Table 1 Characteristics of the datasets (original) and their one-hot encodings (OHE)

	Retailer (R)		Favorita (F)		Yelp (Y)	
	orig	OHE	orig	OHE	orig	OHE
<i>Input database</i>						
# rows (M)	84	0.84	125	1.28	2	0.013
size on disc (GB)	7.9	38.9	11.7	48.4	0.52	1
<i>Join output</i>						
# rows (M)	84	0.84	127	1.27	150	1.5
# data columns	43	2004	30	1645	50	5625
size on disc (GB)	84.2	39.2	89.4	50	175.7	197.5
PSQL time (s)	143.3	0.7	217	0.8	180.8	1.5

column-major order; we verified that they perform best for these orders.

Experimental Setup. All experiments were performed on an Intel Xeon Silver 4214 (24 physical/48 logical cores, 1GHz, 188GiB) with Debian GNU/Linux 10. We use g++ 10.1 for compiling the C++ code using the Ofast optimization flag. The performance numbers are averages over 20 consecutive runs. We do not consider the time to load the database into RAM and assume that all relations and the join output are sorted by their join attributes. All systems use all cores in all experiments apart from Experiment 2.

Datasets. We use three datasets. Retailer (R) [51] and Favorita (F) [19] are used for forecasting user demands and sales. Yelp (Y) [60] has review ratings given by users to businesses. The characteristics of these datasets (Table 1) are common in retail and advertising, where data is generated by sales transactions or click streams. Retailer has a snowflake schema, Favorita has a star schema. Both have key-fkey joins, a large fact table, and several small dimension tables. Yelp has a star schema with many-to-many joins. We also consider a one-hot-encoding version of 1% of these datasets (OHE), where some keys are one-hot encoded in new data columns. They yield wider matrices. Some plots show performance for a percentage of the join output. The corresponding input dataset is computed by projecting the fragment of the join output onto the schemas of the relations. When taking a percentage of an OHE dataset, we map an attribute domain to that percentage of it using division hashing.

We also use synthetic datasets of relations $\mathbf{S}, \mathbf{T} \in \mathbb{R}^{m \times n}$, whose join is the Cartesian product. The data in each column follows a uniform distribution in the range $[-3, 3]$. For accuracy experiments, we fix (part of) the output \mathbf{R}_{fixed} and derive the input relation \mathbf{S} so that the QR decomposition of the Cartesian product agrees with \mathbf{R}_{fixed} . The advantage of this approach is that \mathbf{R}_{fixed} can be effectively used as ground truth for checking the accuracy of the algorithms.

9.1 Summary

The main takeaway of our experiments is that FiGARO significantly outperforms its competitors in runtime for computing QR, SVD, and PCA, as well as in accuracy for computing \mathbf{R} , by a factor proportional to the gap between the join output and input sizes. This factor is up to two orders of magnitude. This includes scenarios with key-fkey joins over real data with millions of rows and tens of data columns and with many-to-many joins over synthetic data with thousands of rows and columns. Whenever the join input and output sizes are very close, however, FiGARO's benefit is small; we verify this by reducing the number of rows and increasing the number of columns of our real data. When the number of data columns is at least the square root of the number of rows, e.g., 1K columns and 1M rows in our experiments, the performance gap almost vanishes. This is to be expected, since the complexity of computing the matrix \mathbf{R} increases linearly with the number of rows but quadratically with the number of columns.

We further show that FiGARO's performance depends on the join tree. Also, the accuracy of the orthogonal matrices \mathbf{Q} and \mathbf{U} computed by FiGARO depends on the condition number of the datasets and can be better or worse than MKL.

Yet there is not one flavour of FiGARO that performs best in all our experiments. In case of thin matrices, i.e., with less than 50 data columns in the join output, or for very sparse matrices, such as those obtained by one-hot encoding, FiGARO-THIN is the best as its QR post-processing phase can exploit effectively the sparsity in the result of FiGARO's Givens rotations. For dense and wide matrices, i.e., with hundreds or thousands of data columns, FiGARO-MKL is the best as its QR post-processing phase uses MKL that parallelizes better than THIN. Therefore, the following experiments primarily focus on the performance (in both runtime and accuracy) of the best of the two algorithms, with brief notes on the performance of the other algorithm. This means that we use FiGARO-MKL for the experiments with the synthetic datasets and FiGARO-THIN for the real-world datasets and their OHE versions.

FiGARO is the only algorithm that works directly on the input database, all others work on the materialized join output. For FiGARO, we report the time to compute both the matrix decompositions and the join *intertwined*, whereas for the others we only report the time to compute the matrix decompositions over the *precomputed* join matrix. Table 1 gives the times for materializing the join for the three datasets; these join times are typically larger than the MKL compute times.

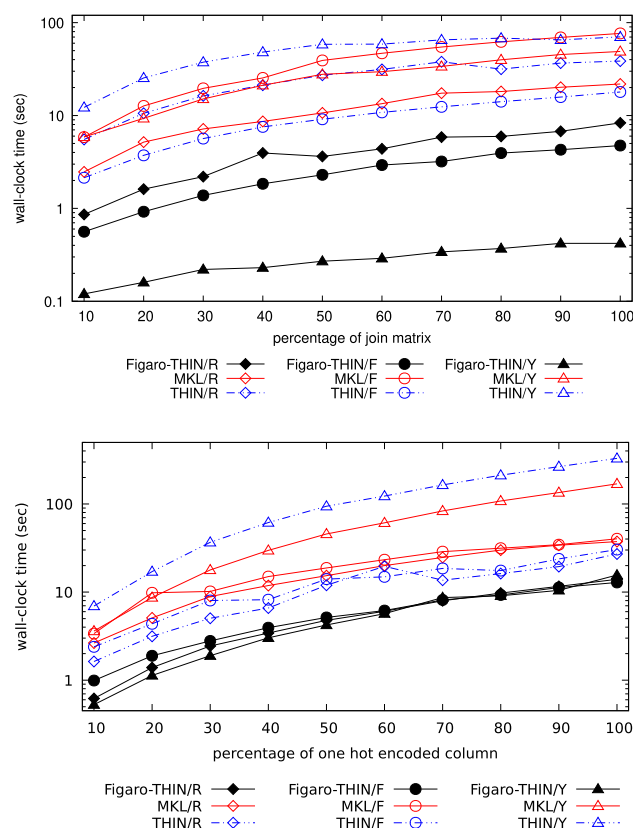


Fig. 5 Exp. 1: Runtime performance for computing \mathbf{R} in the QR decomposition over the original datasets R, F, Y (top) and their OHE versions (bottom)

9.2 QR decomposition

We first consider the task of computing the upper-triangular matrix \mathbf{R} only. After that, we investigate the computation of the orthogonal matrix \mathbf{Q} .

Experiment 1: Runtime performance for computing \mathbf{R} . When compared to its competitors, FiGARO performs very well for real data in case the join output is larger than its input. Figure 5 (top) gives the runtime performance of FiGARO-THIN, THIN, and MKL for the three real datasets as a function of the percentage of the dataset. The performance gap remains mostly unchanged as the data size is increased. Relative to MKL, FiGARO-THIN is 2.9x faster for Retailer, 16.1x for Favorita and 120.5x for Yelp. MKL outperforms THIN, except for Favorita where the latter is 4.3x faster than the former. This is because Favorita has the smallest number of columns amongst the three datasets and THIN works particularly well on thin matrices. We do not show FiGARO-MKL to avoid clutter; it consistently performs worse (up to 3x) than FiGARO-THIN.

#rows	#columns				#columns	#columns	#columns	#columns
	2 ⁶	2 ⁸	2 ¹⁰	2 ¹²				
2 ⁹	0.10	0.13	0.17	0.43	3	18	104	368
2 ¹⁰	0.10	0.13	0.30	0.81	16	76	246	609
2 ¹¹	0.12	0.15	0.31	1.77	53	278	785	
2 ¹²	0.12	0.14	0.38	4.65	201	1056		
2 ¹³	0.13	0.18	0.54	5.85	672			

Fig. 6 Exp. 1: Runtime performance of FiGARO-MKL and MKL for computing **R** in the QR decomposition of the Cartesian product of two relations. The numbers of rows and columns are per relation; for relations of 2¹³ rows columns and 2¹², MKL's input is a 2²⁶ × 2¹³ matrix. Left: Runtime performance of FiGARO-MKL (sec). Right: Speed-up of FiGARO-MKL over MKL (rounded to closest natural number). An empty cell means that MKL runs out of memory

We next consider the OHE versions of Retailer and Favorita, for which the join input and output sizes are close. Figure 5 (bottom) gives the runtime performance of FiGARO-THIN, THIN, and MKL for the one-hot encoded fragments of the three real datasets as a function of the percentage of one-hot encoded columns. The strategy of FiGARO to push past joins is less beneficial in this case, as it copies heads and tails along the join tree without a significant optimization benefit. Relative to MKL, it is 2.7x faster for Retailer and 3.1x faster for Favorita. The explanation for this speed-up is elsewhere: Whereas the speed-up in Fig. 5 (top) is due to structural sparsity, as enforced by the joins, here we have value sparsity due to the many zeros introduced by one-hot encoding. By sorting on the one-hot encoded attribute and allocating blocks with the same attribute value to each thread, we ensure that large blocks of zeros in the one-hot encoding will not be dirtied by the rotations performed by the thread. This effectively preserves many zeros from the input and reduces the number of rotations needed in post-processing to zero values in the output of FiGARO. This strategy needs however to be supported by a good join tree: A relation with one-hot encoded attributes is sorted on these attributes and this order is a prefix of the sorting order used by its join with its parent relation in the join tree.

We further verified that more input rows allowed FiGARO to scale better, but MKL ran out of memory. THIN and MKL have similar performance. For the Yelp OHE dataset, its size remains much less than its join result and FiGARO outperforms MKL by 11x.

Pronounced benefits are obtained for many-to-many joins, for which the join output is much larger than the input. We verified this claim for the Cartesian products of two relations of thousands of rows and columns. FiGARO outperforms MKL by up to three orders of magnitude on this synthetic data (Fig. 6). As expected, FiGARO scales linearly with the number of rows, while MKL scales quadratically as it works on the materialized Cartesian product. The speed-up of FiGARO over MKL increases as we increase the number

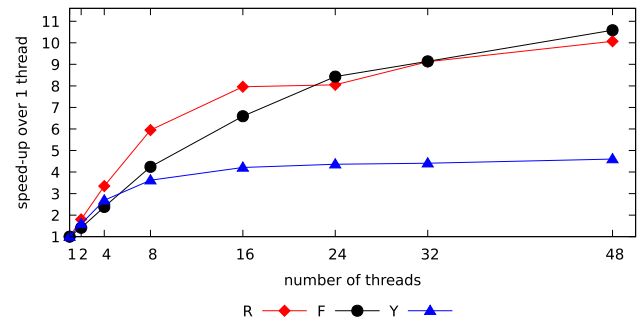


Fig. 7 Exp. 2: Speed-up of multi-threading over single-threading for FiGARO-THIN computation of **R** in the QR decomposition over the three datasets

Table 2 Exp. 3: FiGARO-THIN's runtime to compute **R** in the QR decomposition using a bad join tree and relative speed-up using a good join tree. The join trees use term notation over the abbreviated relation names: Retailer: Inventory(I), Item(T), Weather(W), Location(L), Census(C); Favorita: Sales(S), Stores(R), Oil(O), Holidays(H), Items(I), Transactions(T); Yelp: Business(B), Category(C), CheckIn(I), User(U), Hours(H), Review(R). The relation with OHE values is in bold

		Retailer	Favorita	Yelp
		Bad	Good	Good
Original	Bad	I(L(C),W,T) 67.10s	S(T,(R,O),H,I) 29.34s	R(B(C,I,H),U) 1.31s
	Good	L(C,I(W,T)) 8.06x	R(T(O,S(H,I))) 6.17x	B(C,I,H,R(U)) 3.13x
OHE	Bad	L(C,I(W,T)) 431.57s	H(S(T(R,O),I)) 802.62s	B(C,I,H, R (U)) 11.91s
	Good	T(I(L(C),W)) 21.25x	I(S(T(R,O),H)) 47.13x	H(B(R (U),C,I)) 1.32x

of rows and columns of the two relations. For wide relations (2¹² columns), FiGARO-MKL (shown in figure) is up to 10x faster than FiGARO-THIN (not shown).

Most of the time for FiGARO is taken by post-processing. Computing the batch of the group-by aggregates and the tails and heads of the input relations take under 10% of the overall time for OHE datasets, and about 50% for the original datasets.

Experiment 2: Multi-cores scalability. FiGARO uses domain parallelism: It splits each input relation into as many contiguous blocks as available threads and applies the same transformation to each block independently. Figure 7 shows the performance of FiGARO-THIN as we vary the number of threads up to the number of available logical cores (48). The fastest speed-up increase is achieved up to 8 threads for all datasets and up to 16 threads for Retailer and Favorita. The speed-up still increases up to 48 threads, yet its slope gets smaller. This is in particular the case for the smallest dataset Yelp, for which FiGARO-THIN only takes under 0.5 s using 8 threads.

Experiment 3: Effect of join trees. The runtime of FiGARO is influenced by the join tree. As in classical query optimization, FiGARO prefers join trees such that the over-

all size of the intermediate results is the smallest. For our datasets, this translates to having the large fact tables involved earlier in the joins. We explain for Retailer, whose large and narrow fact table Inventory uses key-fkey joins on composite keys (item id, date, location) to join with the small dimension tables Locations, Weather, Item and Census. FiGARO aggregates away join keys as it moves from leaves to the root of the join tree. By aggregating away join keys over the large table as soon as possible, it creates small intermediate results and reduces the number of copies of dimension-table tuples in the intermediate results to pair with tuples in the fact table. If the fact table would be the root, then FiGARO would first join it with its children in the join tree and then apply transformations, with no benefit over MKL as in both cases we would work on the materialized join output. Therefore, a bad join tree for Retailer has Inventory (I) as a root. In contrast, a good join tree first joins Inventory with Weather and Item, thereby aggregating away the item id and date keys and reducing the size of the intermediate result to be joined with Census and Location. As shown in Table 2, FiGARO-THIN performs 8.06x faster using a good join tree instead of a bad one.

For the OHE datasets, a key performance differentiator is the sorting order of the relations with one-hot encoded attributes, as explained in Experiment 1. For instance, the relation Inventory in the Retailer dataset (printed in bold in Table 2) has the one-hot encoded attribute product id. In a bad join tree, it is not sorted on this attribute: it joins with its parent relation Location on location id, so it is primarily sorted on location id. In a good join tree, Inventory is sorted on product id as it joins with its parent Item on product id.

Experiment 4: Runtime performance for computing \mathbf{Q} . FiGARO outperforms MKL for computing the orthogonal matrix \mathbf{Q} (Fig. 8). For this experiment, FiGARO takes as input the already computed \mathbf{R} , while MKL takes as input the precomputed Household vectors, which are also based on \mathbf{R} . The speed-up is 5.4x for Retailer, 3.7x for Favorita, and 13x for Yelp (Fig. 8 top). As we linearly increase the join matrix size, the runtime performance of FiGARO and MKL increases linearly. For the OHE datasets, the speed-up is 1.9–6.7x for Retailer, 1.9–10.5x for Favorita, and 29–47x for Yelp (Fig. 8 bottom).

Experiment 5: Runtime performance for end-to-end QR decomposition. To complete the picture, Fig. 9 reports the performance for computing both \mathbf{R} and \mathbf{Q} . The input to FiGARO-THIN is the database and to MKL is the materialized join output. As expected, FiGARO-THIN clearly outperforms MKL for all considered datasets.

Experiment 6: Accuracy of computed \mathbf{R} . To assess the accuracy of computing \mathbf{R} , we designed a synthetic dataset consisting of two relations and want to compute the upper-triangular \mathbf{R} in the QR decomposition of their Cartesian product. The input relations are defined based on a given matrix \mathbf{R}_{fixed} such that \mathbf{R}_{fixed} is part of \mathbf{R} . The construction is

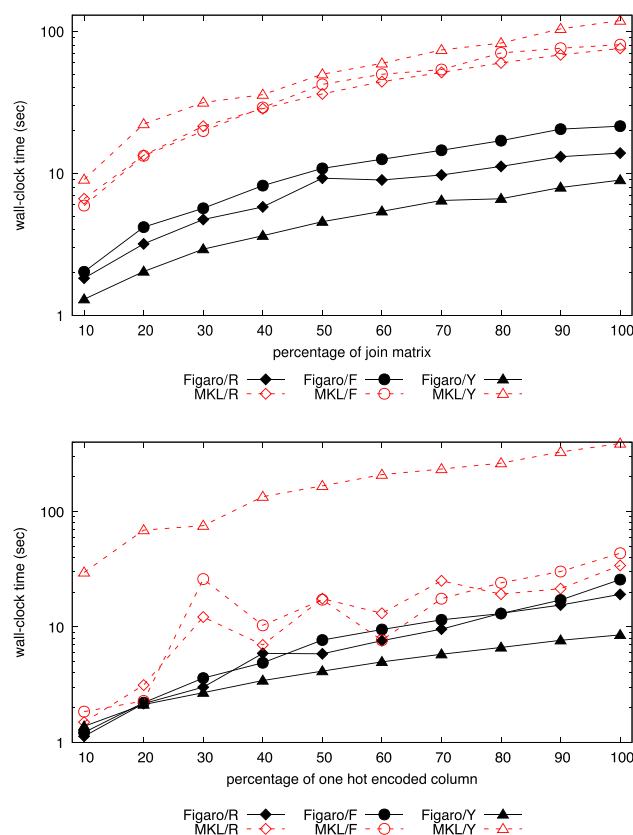


Fig. 8 Exp. 4: Runtime performance for computing the fully materialized \mathbf{Q} for the original datasets R, F, Y (top) and their OHE versions (bottom)

detailed in App. A. We report the relative error $\frac{\|\mathbf{R}_{fixed} - \hat{\mathbf{R}}_{fixed}\|_F}{\|\mathbf{R}_{fixed}\|_F}$ of the computed partial result $\hat{\mathbf{R}}_{fixed}$ compared to the ground truth \mathbf{R}_{fixed} , where $\|\cdot\|_F$ is the Frobenius norm.

Table 3 (left) shows the error of FiGARO-MKL as we vary the number of rows and columns of the two relations following a geometric progression. As the number of rows increases, the accuracy only changes slightly. The accuracy drops as the number of columns increases. The error remains however sufficiently close to the machine representation

Table 3 Exp. 6: (Left) Error for FiGARO-MKL relative to the ground truth. (Right) Division of relative error of MKL over the relative error of FiGARO-MKL. The empty bottom-right cell is due to the out-of-memory error for MKL

#rows	#columns			#columns		
	2 ⁴	2 ⁶	2 ⁸	2 ⁴	2 ⁶	2 ⁸
2 ⁹	2.3e-15	1.8e-14	3.7e-14	26	1.5	1.2
2 ¹⁰	3.5e-15	3.3e-14	1.3e-13	73	5.3	1.3
2 ¹¹	4.7e-15	4.3e-14	3.2e-13	250	20	1.6
2 ¹²	6e-15	5.4e-14	5.2e-13	830	64	5.4
2 ¹³	7.9e-15	6.3e-14		2600	250	

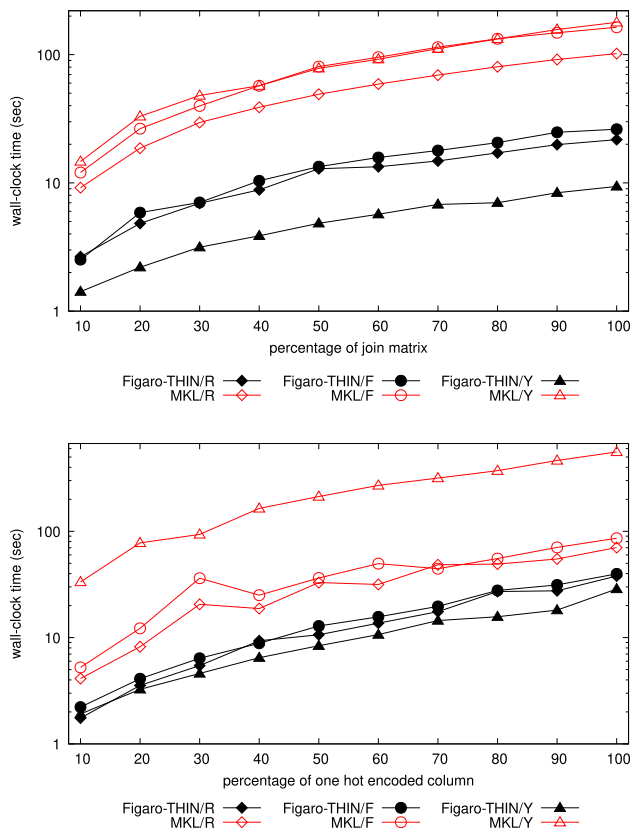


Fig. 9 Exp. 5: Runtime performance for end-to-end QR decomposition for the original datasets R, F, Y (top) and their OHE versions (bottom)

unit (10^{-16}). We also verified that FiGARO-THIN's error is very similar.

We also computed the relative error for MKL. Table 3 (right) shows the result of dividing the relative errors of MKL and FiGARO-MKL. A number greater than 1 means that FiGARO-MKL is more accurate than MKL. The error gap increases with the number of rows and decreases with the number of columns. The latter is as expected, as post-processing dominates the computation for wide matrices. FiGARO-MKL is up to three orders of magnitude more accurate than MKL.

Experiment 7: Accuracy of computed \mathbf{Q} . The accuracy of computing $\mathbf{Q} \in \mathbb{R}^{m \times n}$ is given by the orthogonality error $\frac{\|\mathbf{Q}^T \mathbf{Q} - \mathbf{I}\|_F}{\|\mathbf{I}\|_F}$, where $\mathbf{I} \in \mathbb{R}^{n \times n}$ is the identity matrix and $\|\cdot\|_F$ is the Frobenius norm [30, p. 360]. An orthogonality error of 0 is the ideal outcome, it means that \mathbf{Q} is perfectly orthogonal.

Figure 10 depicts the orthogonality errors of the matrices \mathbf{Q} constructed by FiGARO-MKL, FiGARO-THIN and MKL. We disregard FiGARO-THIN in the following analysis, as it is less accurate than FiGARO-MKL in this experiment. This is as expected, as the THIN post-processing is far less optimized with respect to numerical stability than MKL. We see that FiGARO-MKL is more accurate for Retailer (one order of magnitude) and Favorita (two orders of magnitude), but

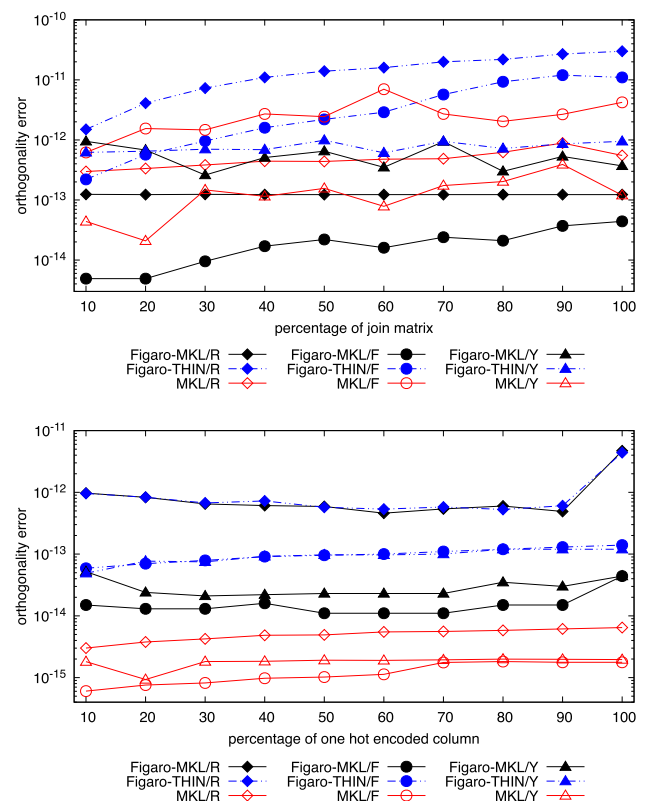


Fig. 10 Exp. 7: Orthogonality error of \mathbf{Q} computed by both FiGARO variants and MKL for the three datasets (top) and their OHE versions (bottom)

Table 4 Exp. 7: Condition numbers and smallest singular values for the three datasets and their OHE versions

Measurement	Dataset	Real	OHE
Condition number	Retailer	3.75E+07	2.72E+09
	Favorita	6.90E+04	3.41E+06
	Yelp	1.70E+10	1.78E+10
Smallest singular value	Retailer	36.2052	0.047772
	Favorita	482.358	0.367022
	Yelp	2.19208	0.208876

less accurate for Yelp (one order of magnitude). The reason is that the *condition number* [30] of the join matrix \mathbf{A} influences the orthogonality of \mathbf{Q} computed by FiGARO. The condition number is the largest singular value of \mathbf{A} divided by the smallest singular value of \mathbf{A} . A very large condition number means that the smallest singular value is very close to 0, so, the matrix is almost singular [14, 46]. If this is the case for \mathbf{A} , the same holds for the matrix \mathbf{R} of its QR decomposition, as they have the same singular values. Recall that FiGARO inverts \mathbf{R} to compute \mathbf{Q} (Sect. 8), this step may introduce larger rounding errors when \mathbf{R} is close to being singular.

Table 4 gives the condition numbers and smallest singular values for the join matrix \mathbf{A} for each of our three datasets

Table 5 Exp. 7: (Left) Orthogonality error of \mathbf{Q} for FiGARO-MKL over the Cartesian product. (Right) Division of orthogonality error of MKL by the orthogonality error of FiGARO-MKL. The right-bottom cell is empty as MKL ran out of memory

#rows	#columns			#columns		
	2^4	2^6	2^8	2^4	2^6	2^8
2^9	1.3E-14	1.7E-14	3.9E-14	3.7	1.9	0.9
2^{10}	6.2E-15	2.1E-14	5.7E-13	21	8.8	2.3
2^{11}	7.0E-14	7.3E-14	1.3E-13	9.8	11.9	3.9
2^{12}	2.8E-14	4.1E-14	1.5E-13	112.1	93.1	18.6
2^{13}	2.2E-13	2.0E-13		56.1	81.0	

and their OHE versions. Since Favorita has the smallest condition number, the orthogonality error of \mathbf{Q} computed by FiGARO-MKL is very low, i.e., \mathbf{Q} is very close to an orthogonal matrix. In contrast, Yelp has the largest condition number and FiGARO produces a less orthogonal matrix \mathbf{Q} for this dataset.

Table 4 also shows that the condition numbers are much larger for the OHE versions of Retailer and Favorita (by a factor of 100x), while the condition number remains almost the same for the OHE version of Yelp. We therefore expect less accurate matrices \mathbf{Q} computed by FiGARO-MKL. This is indeed the case, as shown in Fig. 10 (bottom). Remarkably, MKL improves the orthogonality of \mathbf{Q} in case of the OHE datasets relative to the original datasets, which suggests that it can effectively exploit the sparsity due to the one-hot encoding to avoid accumulating too many rounding errors.

Table 5 gives the orthogonality error of \mathbf{Q} for the Cartesian product of two relations as we vary their numbers of rows and columns. The condition numbers, the smallest singular values, and even the orthogonality vary in the same range as for Retailer original and OHE in Table 4. The orthogonality error for \mathbf{Q} computed by FiGARO-MKL remains rather low and is 1–100x lower than for MKL (Table 5 right). The gap between the two systems closes as we increase the number of columns.

9.3 Singular value decomposition

We extended FiGARO to compute the SVD of the join matrix \mathbf{A} as detailed in Sect. 8. We call this extension SVD- Fig. It works directly on the input database. We compare it against SVD- MKL, the divide&conquer approach of MKL that computes the QR decomposition of \mathbf{A} and then bidiagonalizes the upper-triangular matrix \mathbf{R} . When only singular values are computed, all methods use the dqds algorithm [20] to compute these values from the intermediate bidiagonal matrices. We also experimented with the QR iteration, the power iteration, and the eigendecomposition approaches. QR iteration performs similar to SVD- MKL, although for the OHE datasets it is one order of magnitude less accurate, where

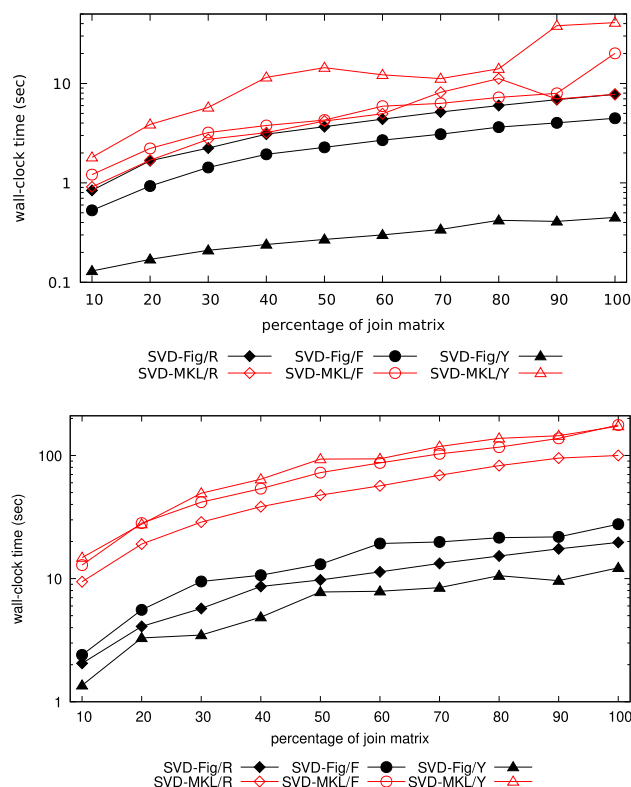


Fig. 11 Exp. 8: Runtime performance of SVD- Fig and MKL over the original datasets R, F and Y. Top: Only singular values (Σ) are computed. Bottom: The entire SVD is computed: \mathbf{U} , Σ , \mathbf{V}

accuracy is measured as the orthogonality of the matrix \mathbf{U} . Power iteration is at least one order of magnitude slower than SVD- Fig for achieving a comparable accuracy. In our tests, the orthogonality error of the eigendecomposition approach was much larger than for the other approaches.

Experiment 8: Runtime performance for computing SVD. Fig. 11 (bottom) shows that SVD- Fig clearly outperforms SVD- MKL. They both scale linearly with the join output size, yet SVD- Fig is 5x faster than SVD- MKL for Retailer and Favorita and 16x for Yelp. When only the singular values are computed (Fig. 11 top), SVD- Fig outperforms SVD- MKL by factors 1–1.9x for Retailer, 2–4.5x for Favorita, and up to 90x for Yelp. The reason for the large gap for Yelp is twofold. The singular values of \mathbf{A} are the same as for the upper-triangular matrix \mathbf{R} in the QR decomposition of \mathbf{A} . To compute \mathbf{R} , FiGARO does not need the materialized join matrix \mathbf{A} , which is much larger than the Yelp input dataset. For the OHE datasets, SVD- Fig is faster than SVD- MKL by 2–3x faster for Retailer, 2–5x for Favorita, and 16–22x for Yelp (not shown in the figure). These speed-ups are at the same scale as in Experiment 4.

Experiment 9: Accuracy of computed SVD. We first investigate the orthogonality error for the truncated matrix \mathbf{U} of left singular vectors as we vary the percentage of non-

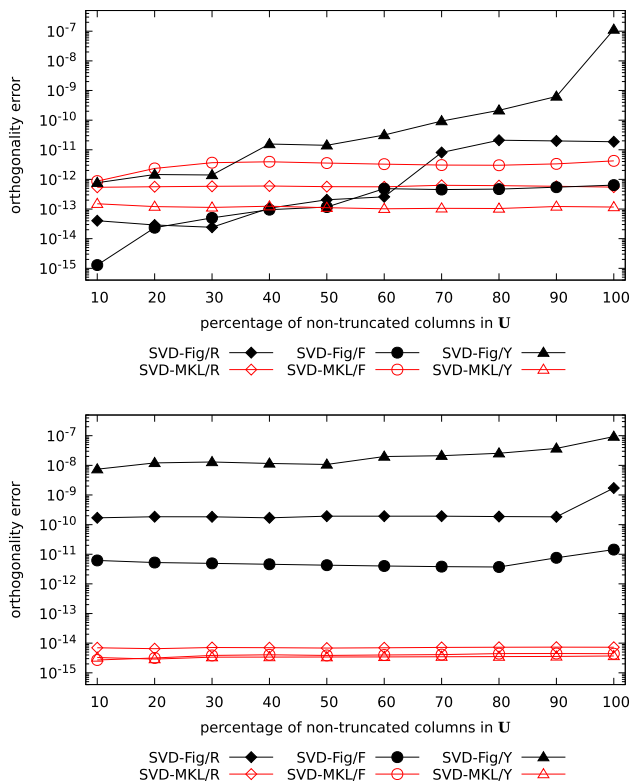


Fig. 12 Exp. 9: Orthogonality error for \mathbf{U} computed by SVD- FIG and MKL for our datasets R, F and Y as we vary the percentage of non-truncated columns in \mathbf{U} . Top: Original datasets. Bottom: OHE versions

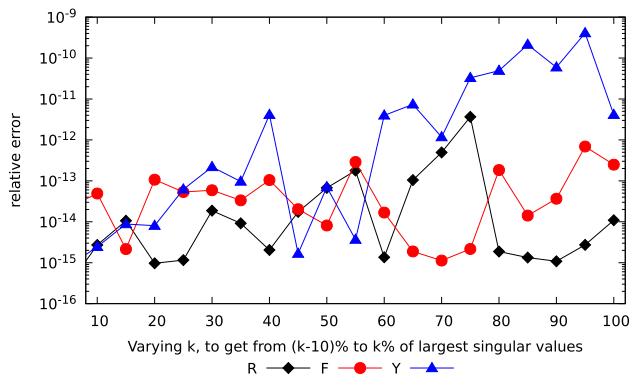


Fig. 13 Exp. 9: Error of SVD- FIG relative to MKL for computing the top $k - 5$ to k per cent of the largest singular values of the join matrices for the original datasets

truncated columns (Fig. 12). The reason we look into the accuracy of truncated \mathbf{U} is twofold. First, this is the largest matrix in the SVD of \mathbf{A} . Second, its truncated version is used for PCA. When the percentage of non-truncated columns increases, the orthogonality error increases for SVD- FIG, for MKL it remains roughly the same.

This is because the condition number of the join matrix \mathbf{A} grows with the number of non-truncated columns.

Table 6 Exp. 9: (Left) Orthogonality error for \mathbf{U} computed by SVD- FIG and truncated to 40% columns. (Right) Division of orthogonality error of MKL by the orthogonality error of SVD- FIG, the result greater than one means SVD- FIG is more accurate. The right-bottom cell is empty as MKL ran out of memory

#rows	#columns			#columns		
	2 ⁴	2 ⁶	2 ⁸	2 ⁴	2 ⁶	2 ⁸
2 ⁹	1.8E-14	2.7E-15	6.1E-15	0.3	2.5	1.8
2 ¹⁰	3.1E-15	2.9E-15	7.2E-15	3.9	4.8	4.1
2 ¹¹	1.8E-15	1.1E-14	7.9E-15	12.9	2.7	6.9
2 ¹²	2.4E-15	7.7E-15	6.3E-15	21.7	8.0	29.0
2 ¹³	3.9E-15	5.4E-15		25.9	25.1	

Table 7 Exp. 9: (Left) Orthogonality error for the full \mathbf{U} computed by SVD- FIG. (Right) Division of the orthogonality error of MKL by the orthogonality error of SVD- FIG, the result greater than one means SVD- FIG is more accurate. The right-bottom cell is empty as MKL ran out of memory

#rows	#columns			#columns		
	2 ⁴	2 ⁶	2 ⁸	2 ⁴	2 ⁶	2 ⁸
2 ⁹	4.2E-12	6.3E-12	4.2E-11	1.3E-03	5.0E-03	8.7E-04
2 ¹⁰	9.5E-12	6.5E-12	4.5E-11	1.4E-02	2.9E-02	2.9E-03
2 ¹¹	1.5E-12	1.1E-11	5.0E-11	4.5E-01	8.0E-02	9.9E-03
2 ¹²	1.2E-11	3.0E-11	6.1E-11	2.7E-01	1.3E-01	4.7E-02
2 ¹³	2.0E-11	3.7E-11		6.3E-01	4.3E-01	

We see a similar behaviour for the accuracy of the computed singular values. Figure 13 reports the relative error of a sliding window of 5% of the singular values as computed by SVD- FIG and by MKL, sliding over all singular values in decreasing order. For vectors \mathbf{v}_1 and \mathbf{v}_2 computed by SVD- FIG and respectively MKL, the relative error is $\frac{\|\mathbf{v}_1 - \mathbf{v}_2\|_F}{\|\mathbf{v}_2\|_F}$. The difference between the singular values as computed by SVD- FIG and MKL tends to be small for the largest singular values and increases for smaller singular values. Also, the singular values computed by SVD- FIG are closer to the values computed by MKL for the join matrices with smaller condition numbers such as Favorita, while for Retailer and Yelp they are more different. We observed similar behaviour for the OHE datasets.

We finally investigate the accuracy for the synthetic dataset and considered different percentages of truncated columns in the matrix \mathbf{U} . Table 6 reports the orthogonality error of the matrix \mathbf{U} truncated to 40% of columns. The matrix \mathbf{U} computed by SVD- FIG is more orthogonal than the one computed by MKL, the reasoning is similar to the orthogonality of \mathbf{Q} in Experiment 7. If we compute the entire matrix \mathbf{U} , then SVD- FIG incurs a higher orthogonality error than MKL due to the large condition numbers, as shown in Table 7.

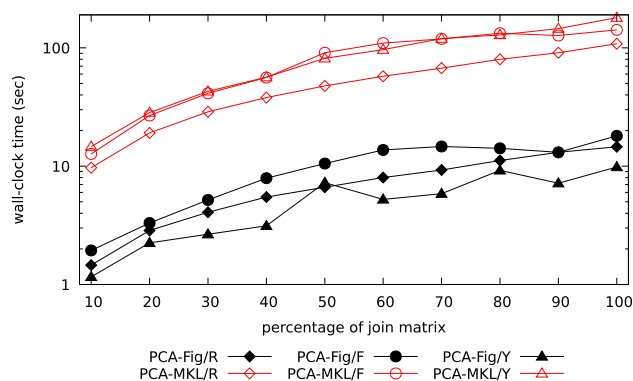


Fig. 14 Exp. 10: Runtime performance of PCA- Fig and MKL for our datasets R, F, and Y

9.4 Principal component analysis

Section 8 shows how to compute PCA using SVD. The top- k principal components of the join matrix \mathbf{A} are the right singular vectors in \mathbf{V} that correspond to the top- k largest singular values of \mathbf{A} . Both \mathbf{V} and the singular values of \mathbf{A} are also of the upper-triangular matrix \mathbf{R} computed by FiGARO. The projection of \mathbf{A} onto the k -dimensional space is given by $\mathbf{U}_{:,1:k} \Sigma_{1:k,1:k}$, where $\mathbf{U}_{:,1:k}$ is the truncated matrix \mathbf{U} of left singular vectors and the diagonal matrix $\Sigma_{1:k,1:k}$ has the k largest singular values along the diagonal. Experiment 9 on the accuracy of the computed truncated \mathbf{U} and the singular values carry over to PCA immediately.

Experiment 10: Runtime performance for computing PCA. Fig. 14 reports the runtimes to compute $\mathbf{U}_{:,1:k} \Sigma_{1:k,1:k}$ for $k = 10\%$ of the number of data columns, i.e., k is 4 for Retailer, 3 for Favorita, and 5 for Yelp. PCA- Fig is the FiGARO adaptation to PCA. It takes as input the database. PCA- MKL is a custom implementation that uses MKL to compute SVD from the join matrix \mathbf{A} and then multiplies the truncated matrices \mathbf{U} and Σ . For this experiment, we did not center the data. Figure 14 shows a consistent gap of one order of magnitude between the two systems. This is similar to the performance reported in Fig. 11, since the most expensive computation is taken by the SVD.

FiGARO can compute the principal components corresponding to the largest singular values in our experiments faster than MKL and with similar accuracy. FiGARO is thus a good alternative to MKL in case of matrices defined by joins over relational data.

10 Related work

Our work sits at the interface of linear algebra and databases and is the first to investigate QR decomposition over database joins using Givens rotations. It complements seminal seven-

decades-old work on QR decomposition of matrices. Our motivation for this work is the emergence of machine learning algorithms and applications that are expressed using linear algebra and are computed over relational data.

Improving performance of linear algebra operations.

Factorized computation [42] is a paradigm that uses structural compression to lower the cost of computing a variety of database and machine learning workloads. It has been used for the efficient computation of $\mathbf{A}^T \mathbf{A}$ over training datasets \mathbf{A} created by feature extraction queries, as used for learning a variety of linear and nonlinear regression models [33, 36, 40, 50, 51]. It is also used for linear algebra operations, such as matrix multiplication and element-wise matrix and scalar operations [9]. Our work also uses a factorized multiplication operator of the non-materialized matrix \mathbf{A} and another materialized matrix for the computation of the orthogonal matrix \mathbf{Q} in the QR decomposition of \mathbf{A} and for the SVD and PCA of \mathbf{A} . Besides structural compression, also value-based compression is useful for fast in-memory matrix-vector multiplication [18]. Existing techniques for matrix multiplication, as supported by open-source systems like SystemML [7], Spark [62], and Julia [3], exploit the sparsity of the matrices. This calls for sparsity estimation in the form of zeroes [54].

The matrix layout can have a large impact on the performance of matrix computation in a distributed environment [38]. Distributed database systems that can support linear algebra computation [37] may represent an alternative to high-performance computing efforts such as ScaLAPACK [10]. There are distributed algorithms for QR decomposition in ScaLAPACK [8, 58].

QR decomposition. There are three main approaches to QR decomposition of a materialized matrix \mathbf{A} . The first one is based on *Givens rotations* [24], as also considered in our work. Each application of a Givens rotation to a matrix \mathbf{A} sets one entry of \mathbf{A} to 0. This method can be parallelized, as each rotation only affects two rows of \mathbf{A} . It is particularly efficient for sparse matrices. One form of sparsity is the presence of many zeros. We show its efficiency for another form of sparsity: the presence of repeating blocks whose one-off transformation can be reused multiple times.

Householder transformations [31] are particularly efficient for dense matrices [30, p. 366]. MKL and openblas used in our experiments implement this method. One Householder transformation sets all but one entry of a vector to 0, so the QR decomposition of an $m \times n$ matrix can be obtained using n transformations. Both Givens and Householder are numerically stable [30, Chapter 19].

The *Gram-Schmidt process* [28, 52] computes the orthogonal columns of \mathbf{Q} one at a time by subtracting iteratively from the i -th column of \mathbf{A} all of its projections onto the previously computed $i - 1$ orthogonal columns of \mathbf{Q} . A slightly modified variant [5] is numerically stable. The modified

Gram-Schmidt is mathematically and numerically equivalent to applying Householder transformations to a matrix that is padded with zeros [6].

Implementing Givens rotations. There are several options on how to set the values c and s of a Givens rotation matrix in the presence of negative values. We follow the choice proposed by Bindel [4]. Anderson [2] defends the original choice [24] of signs using numerical stability arguments. Gentleman [23] shows that one can compute an upper-triangular matrix \mathbf{R}' and a diagonal matrix \mathbf{D} without computing square roots such that $\mathbf{R} = \mathbf{D}^{\frac{1}{2}} \mathbf{R}'$. Stewart [55, p. 291] discusses in depth these fast rotations.

SVD The approach to computing the SVD that is closest to ours proceeds as follows [26, p. 285], [35]. It computes the matrices \mathbf{R} and \mathbf{Q} in the QR decomposition of \mathbf{A} , followed by the computation of the SVD of $\mathbf{R} = \mathbf{U}_R \Sigma_R \mathbf{V}_R^T$. Finally, it computes the orthogonal matrix $\mathbf{U} = \mathbf{Q} \mathbf{U}_R$. The SVD of \mathbf{A} is then $\mathbf{U} \Sigma_R \mathbf{V}_R^T$. Our approach does not compute \mathbf{Q} but instead expresses it as a multiplication of the non-materialized \mathbf{A} and the inverse of \mathbf{R} . Further approaches use Householder transformations to transform \mathbf{A} into a bidiagonal matrix [26, p. 284], [25], or compute the eigenvalue decomposition of the matrix $\mathbf{A}^T \mathbf{A}$ [26, p. 486], where the singular values are the square roots of the eigenvalues and the matrix \mathbf{V} consists of the corresponding eigenvectors. All prior approaches require the input matrix \mathbf{A} to be materialized. The existing approaches to computing an SVD of a bidiagonal matrix vary in terms of accuracy [11]. When only singular values are required, the squareroot-free or dqds method is the most popular [20, 47]. The QR-iteration [13, 15] and divide-and-conquer methods [29] are used in case also the left and right singular matrices are required.

PCA Principal component analysis is a technique for reducing the dimensionality of a dataset [45]. The approach taken in this paper to computing the PCA of a matrix \mathbf{A} relies on the SVD of the upper-triangular matrix \mathbf{R} in the QR decomposition of \mathbf{A} . The novelty of FiGARo over all prior approaches to PCA is that it does not require the materialization of \mathbf{A} to compute \mathbf{R} , in case \mathbf{A} is defined by database joins. A further approach to PCA over database joins was recently proposed in the database theory literature, without a supporting implementation [33]: It uses the min-max theorem based on the Rayleigh quotient [56] and computes iteratively one eigenvector of $\mathbf{A}^T \mathbf{A}$ at a time. The benefit of that approach is that the computation of $\mathbf{A}^T \mathbf{A}$ can be pushed past joins and may take time less than materializing the matrix \mathbf{A} representing the join result.

11 Conclusion

This article introduces FiGARo, an algorithm that computes the QR decomposition of matrices defined by joins over relational data. By pushing the computation past the joins, FiGARo can significantly reduce the number of computation steps and rounding errors. FiGARo can also be used to compute singular value and eigenvalue decompositions as well as the principal component analysis of the non-materialized matrix representing the joins over relational databases. We experimentally verified that FiGARo can outperform in run-time and in many cases also in accuracy the linear algebra package Intel MKL. In the future, we plan to extend FiGARo to work with categorical data directly, to avoid one-hot encoding such data.

Acknowledgements This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 682588.

Funding Open access funding provided by University of Zurich.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

Appendix: Further details on the accuracy experiment

We show that given an upper-triangular matrix $\mathbf{R}_{\text{fixed}}$, we can devise matrices \mathbf{S} and \mathbf{T} with arbitrary dimensions such that $\mathbf{S} \times \mathbf{T} = \mathbf{Q} \mathbf{R}$ for an upper-triangular $\mathbf{R} = \begin{bmatrix} \mathbf{R}_{\text{fixed}} & \mathbf{V} \\ \mathbf{0} & \mathbf{W} \end{bmatrix}$, for some matrices \mathbf{V}, \mathbf{W} .

We denote by $\mathbf{1}_{m \times n}$ the $m \times n$ matrix that consists entirely of 1s and by $\mathbf{0}_{m \times n}$ the $m \times n$ matrix that consists entirely of 0s.

We revisit the notion of Kronecker product. For an $m \times n$ matrix \mathbf{A} and a $p \times q$ matrix \mathbf{B} , the Kronecker product $\mathbf{A} \otimes \mathbf{B}$ is the $mp \times nq$ matrix

$$\begin{bmatrix} \mathbf{A}[1 : 1] \mathbf{B} & \cdots & \mathbf{A}[1 : n] \mathbf{B} \\ \vdots & \ddots & \vdots \\ \mathbf{A}[m : 1] \mathbf{B} & \cdots & \mathbf{A}[m : n] \mathbf{B} \end{bmatrix},$$

where each $A[i : j]\mathbf{B}$ is the matrix \mathbf{B} multiplied by the scalar $A[i : j]$. We can express a Cartesian product in terms of Kronecker products: for $\mathbf{S} \in \mathbb{R}^{m_1 \times n_1}$ and $\mathbf{T} \in \mathbb{R}^{m_2 \times n_2}$, we have $\mathbf{S} \times \mathbf{T} = [\mathbf{S} \otimes \mathbf{1}_{m_2 \times 1} \quad \mathbf{1}_{m_1 \times 1} \otimes \mathbf{T}]$.

We use the following two observations.

Lemma 6 ([57]) *For arbitrary real matrices \mathbf{A} and \mathbf{B} with respective QR decompositions $\mathbf{A} = \mathbf{Q}_A \mathbf{R}_A$ and $\mathbf{B} = \mathbf{Q}_B \mathbf{R}_B$ it holds $\mathbf{A} \otimes \mathbf{B} = (\mathbf{Q}_A \otimes \mathbf{Q}_B)(\mathbf{R}_A \otimes \mathbf{R}_B)$.*

Lemma 7 *Let $\mathbf{A} \in \mathbb{R}^{m \times n_1}$ and $\mathbf{B} \in \mathbb{R}^{m \times n_2}$ be arbitrary and let $\mathbf{A} = \mathbf{Q}_A \mathbf{R}_A$ be the QR decomposition of \mathbf{A} , where $\mathbf{Q}_A \in \mathbb{R}^{m \times n_1}$, $\mathbf{R}_A \in \mathbb{R}^{n_1 \times n_1}$. There is an orthogonal matrix $\mathbf{Q}' \in \mathbb{R}^{m \times n_2}$ and matrices $\mathbf{V} \in \mathbb{R}^{n_1 \times n_2}$, $\mathbf{W} \in \mathbb{R}^{n_2 \times n_2}$ such that*

$$[\mathbf{A} \quad \mathbf{B}] = [\mathbf{Q}_A \quad \mathbf{Q}'] \begin{bmatrix} \mathbf{R}_A & \mathbf{V} \\ \mathbf{0}_{n_2 \times n_1} & \mathbf{W} \end{bmatrix}.$$

From Lemma 6 it follows that for a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ with QR decomposition $\mathbf{A} = \mathbf{Q}\mathbf{R}$, we have

$$\mathbf{A} \otimes \mathbf{1}_{m \times 1} = (\mathbf{Q} \otimes \mathbf{Q}_1)(\mathbf{R}\sqrt{m})$$

$$\mathbf{1}_{m \times 1} \otimes \mathbf{A} = (\mathbf{Q}_1 \otimes \mathbf{Q})(\mathbf{R}\sqrt{m}),$$

where $\mathbf{1}_{m \times 1} = \mathbf{Q}_1 [\sqrt{m}]$ is the QR decomposition of $\mathbf{1}_{m \times 1}$, $[\sqrt{m}]$ is a 1×1 matrix with only entry \sqrt{m} , and $\mathbf{R}\sqrt{m}$ is the multiplication of the matrix \mathbf{R} with the scalar \sqrt{m} .

Putting the above observations together, we obtain:

Corollary 1 *Let $\mathbf{S} \in \mathbb{R}^{m_1 \times n_1}$, $\mathbf{T} \in \mathbb{R}^{m_2 \times n_2}$ be arbitrary and let $\mathbf{S} = \mathbf{Q}_S \mathbf{R}_S$ be the QR decomposition of \mathbf{S} , where $\mathbf{Q}_S \in \mathbb{R}^{m_1 \times n_1}$, $\mathbf{R}_S \in \mathbb{R}^{n_1 \times n_1}$. There is an orthogonal matrix $\mathbf{Q}' \in \mathbb{R}^{m_1 m_2 \times n_2}$ and matrices $\mathbf{V} \in \mathbb{R}^{n_1 \times n_2}$, $\mathbf{W} \in \mathbb{R}^{n_2 \times n_2}$ such that*

$$\begin{aligned} \mathbf{S} \times \mathbf{T} &= [\mathbf{S} \otimes \mathbf{1}_{m_2 \times 1} \quad \mathbf{1}_{m_1 \times 1} \otimes \mathbf{T}] \\ &= [\mathbf{Q}_S \otimes \mathbf{Q}_1 \quad \mathbf{Q}'] \begin{bmatrix} \mathbf{R}_S \sqrt{m_2} & \mathbf{V} \\ \mathbf{0}_{n_2 \times n_1} & \mathbf{W} \end{bmatrix}. \end{aligned}$$

Let $\mathbf{R}_S \in \mathbb{R}^{n_1 \times n_1}$ be an arbitrary upper-triangular matrix. We arbitrarily choose a vector $\mathbf{v} = (v_1, \dots, v_{m_1})^T \in \mathbb{Q}^{m_1}$ with $\|\mathbf{v}\|_2 = 1$ and a $m_2 \times n_2$ matrix \mathbf{T} of natural numbers, where m_2 is square, so $\sqrt{m_2}$ is a natural number. We set \mathbf{Q}_S to be the first n_1 columns of the orthogonal matrix of rational numbers [63]

$$\hat{\mathbf{Q}} = \begin{bmatrix} v_1 & v_2 & v_3 & \cdots & v_{n_1} \\ v_2 & \frac{v_2^2 - v_1 - 1}{v_1 + 1} & \frac{v_2 v_3}{v_1 + 1} & \cdots & \frac{v_2 v_{m_1}}{v_1 + 1} \\ v_3 & \frac{v_3 v_2}{v_1 + 1} & \frac{v_3^2 - v_1 - 1}{v_1 + 1} & \cdots & \frac{v_3 v_{m_1}}{v_1 + 1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ v_{m_1} & \frac{v_{m_1} v_2}{v_1 + 1} & \frac{v_{m_1} v_3}{v_1 + 1} & \cdots & \frac{v_{m_1}^2 - v_1 - 1}{v_1 + 1} \end{bmatrix},$$

so $\mathbf{Q}_S = \hat{\mathbf{Q}}[:, \{1, \dots, n_1\}]$. We obtain \mathbf{S} as $\mathbf{S} = \mathbf{Q}_S \mathbf{R}_S$.

It follows from Corollary 1 that there is an orthogonal matrix \mathbf{Q} as well as matrices \mathbf{V}, \mathbf{W} such that $\mathbf{S} \times \mathbf{T} = \mathbf{Q} \begin{bmatrix} \mathbf{R}_S \sqrt{m_2} & \mathbf{V} \\ \mathbf{0} & \mathbf{W} \end{bmatrix}$, as desired. Furthermore, if \mathbf{R}_S only consists of rational numbers then so do \mathbf{S}, \mathbf{T} and $\mathbf{R}_S \sqrt{m_2}$.

When computing the QR decomposition of $\mathbf{S} \times \mathbf{T}$ we can compare the ground truth $\mathbf{R}_S \sqrt{m_2}$ with the corresponding part of the computed result.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995). <http://webdam.inria.fr/Alice/>
2. Anderson, E.: Discontinuous plane rotations and the symmetric eigenvalue problem. Tech. Rep. 150, LAPACK Working Note (2000)
3. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: a fresh approach to numerical computing. SIAM Rev. **59**(1), 65–98 (2017)
4. Bindel, D., Demmel, J., Kahan, W., Marques, O.: On computing givens rotations reliably and efficiently. ACM Trans. Math. Softw. **28**(2), 206–238 (2002)
5. Björck, Å.: Solving linear least squares problems by gram-schmidt orthogonalization. BIT **7**(1), 1–21 (1967)
6. Björck, Å., Paige, C.C.: Loss and recapture of orthogonality in the modified gram-schmidt algorithm. SIAM J. Matrix Anal. Appl. **13**(1), 176–190 (1992)
7. Böhm, M., Burdick, D.R., Evfimievski, A.V., Reinwald, B., Reiss, F.R., Sen, P., Tatikonda, S., Tian, Y.: SystemML's optimizer: plan generation for large-scale machine learning programs. IEEE Data Eng. Bull. **37**(3), 52–62 (2014)
8. Bujanovic, Z., Drmac, Z.: New robust ScaLAPACK routine for computing the QR factorization with column pivoting. CoRR **abs/1910.05623** (2019)
9. Chen, L., Kumar, A., Naughton, J.F., Patel, J.M.: Towards linear algebra over normalized data. Proc. VLDB Endow. **10**(11), 1214–1225 (2017)
10. Choi, J., Demmel, J., Dhillon, I.S., Dongarra, J.J., Ostrouchov, S., Petitet, A., Stanley, K., Walker, D.W., Whaley, R.C.: ScaLAPACK: a portable linear algebra library for distributed memory computers—design issues and performance. In: Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science, vol. 1041, pp. 95–106 (1995)
11. Cline, A.K., Dhillon, I.S.: Computation of the singular value decomposition. In: Hogben, L., Brualdi, R., Greenbaum, A., Mathias, R. (eds.) Handbook of Linear Algebra, 1st ed. Chapman and Hall/CRC (2006)
12. da Cunha, R.D., Becker, D., Patterson, J.C.: New parallel (rank-revealing) QR factorization algorithms. In: European Conference on Parallel Processing, pp. 677–686 (2002)
13. Demmel, J., Kahan, W.: Accurate singular values of bidiagonal matrices. SIAM J. Sci. Stat. Comput. **11**(5), 873–912 (1990)
14. Demmel, J.W.: On condition numbers and the distance to the nearest ill-posed problem. Numer. Math. **51**, 251–289 (1987)
15. Demmel, J.W., Li, X.: Faster numerical algorithms via exception handling. IEEE Trans. Comput. **43**(8), 983–992 (1994)
16. Dolmatova, O., Augsten, N., Böhlen, M.H.: A relational matrix algebra and its implementation in a column store. In: SIGMOD, pp. 2573–2587 (2020)
17. Eckart, C., Young, G.: The approximation of one matrix by another of lower rank. Psychometrika **1**(3), 211–218 (1936)

18. Elgohary, A., Boehm, M., Haas, P.J., Reiss, F.R., Reinwald, B.: Compressed linear algebra for declarative large-scale machine learning. *Commun. ACM* **62**(5), 83–91 (2019)
19. Favorita, C.: Corp. Favorita Grocery Sales Forecasting: Can you accurately predict sales for a large grocery chain? (2017). <https://www.kaggle.com/c/favorita-grocery-sales-forecasting/>
20. Fernando, K.V., Parlett, B.N.: Accurate singular values and differential qd algorithms. *Numer. Math.* **67**(2), 191–229 (1994)
21. Francis, J.: The QR transformation, i. *Comput. J.* **4**(3), 265–271 (1961). Received October 1959
22. Geerts, F., Muñoz, T., Riveros, C., Vrgoc, D.: Expressive power of linear algebra query languages. In: PODS, pp. 342–354 (2021)
23. Gentleman, W.M.: Least squares computations by givens transformations without square roots. *IMA J. Appl. Math.* **12**(3), 329–336 (1973)
24. Givens, W.: Computation of plain unitary rotations transforming a general matrix to triangular form. *J. Soc. Ind. Appl. Math.* **6**(1), 26–50 (1958)
25. Golub, G., Kahan, W.: Calculating the singular values and pseudo-inverse of a matrix. *J. Soc. Ind. Appl. Math. Ser. B Numer. Anal.* **2**(2), 205–224 (1965)
26. Golub, G.H., Van Loan, C.F.: *Matrix Computations*, 4th edn. The Johns Hopkins University Press (2013)
27. Golub, G.H.G.H., Plemmons, R.J., Sameh, A.: Parallel block schemes for large scale least squares computations. Technical Report CSRD-574, UIUC Center for Supercomputing Research and Development (1986)
28. Gram, J.: Ueber die Entwicklung reeller Functionen in Reihen mittelst der Methode der kleinsten Quadrate. *Journal für die reine und angewandte Mathematik* **1883**(94), 41–73 (1883)
29. Gu, M., Eisenstat, S.C.: A divide-and-conquer algorithm for the bidiagonal SVD. *SIAM J. Matrix Anal. Appl.* **16**(1), 79–92 (1995)
30. Higham, N.J.: *Accuracy and stability of numerical algorithms*, 2nd edn. SIAM (2002)
31. Householder, A.S.: Unitary triangularization of a nonsymmetric matrix. *J. ACM* **5**(4), 339–342 (1958)
32. Hutchison, D., Howe, B., Suci, D.: Laradb: a minimalist kernel for linear and relational algebra computation. In: BeyondMR@SIGMOD, pp. 2:1–2:10 (2017)
33. Khamis, M.A., Ngo, H.Q., Nguyen, X., Olteanu, D., Schleich, M.: Learning models over relational data using sparse tensors and functional dependencies. *ACM Trans. Database Syst.* **45**(2), 7:1–7:66 (2020)
34. Khamis, M.A., Ngo, H.Q., Rudra, A.: FAQ: questions asked frequently. In: PODS, pp. 13–28 (2016)
35. Lawson, C.L., Hanson, R.J.: *Solving least squares problems*. SIAM (1995)
36. Li, S., Chen, L., Kumar, A.: Enabling and optimizing non-linear feature interactions in factorized linear algebra. In: SIGMOD, pp. 1571–1588 (2019)
37. Luo, S., Gao, Z.J., Gubanov, M.N., Perez, L.L., Jankov, D., Jermaine, C.M.: Scalable linear algebra on a relational database system. *Commun. ACM* **63**(8), 93–101 (2020)
38. Luo, S., Jankov, D., Yuan, B., Jermaine, C.: Automatic optimization of matrix implementations for distributed machine learning and linear algebra. In: SIGMOD, pp. 1222–1234 (2021)
39. Mirsky, L.: Symmetric gauge functions and unitarily invariant norms. *Q. J. Math.* **11**(1), 50–59 (1960)
40. Olteanu, D.: The relational data borg is learning. *Proc. VLDB Endow.* **13**(12), 3502–3515 (2020)
41. Olteanu, D., Huang, J., Koch, C.: SPROUT: lazy vs. eager query plans for tuple-independent probabilistic databases. In: ICDE, pp. 640–651 (2009)
42. Olteanu, D., Schleich, M.: Factorized databases. *SIGMOD Rec.* **45**(2), 5–16 (2016)
43. Olteanu, D., Vortmeier, N., Đorđe Živanović: givens QR decomposition over relational databases. In: SIGMOD, pp. 1948–1961 (2022)
44. Olteanu, D., Závodný, J.: Size bounds for factorised representations of query results. *ACM Trans. Database Syst.* **40**(1), 2:1–2:44 (2015)
45. Pearson, K.: Liii. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* **2**(11), 559–572 (1901)
46. Rump, S.M.: Bounds for the componentwise distance to the nearest singular matrix. *SIAM J. Matrix Anal. Appl.* **18**(1), 83–103 (1997)
47. Rutishauser, H.: Der quotienten-differenzen-algorithmus. *Zeitschrift für angewandte Mathematik und Physik ZAMP* **5**(3), 233–251 (1954)
48. Rutishauser, H.: Solution of eigenvalue problems with the LR-transformation. Further contributions to the solution of simultaneous linear equations and the determination of eigenvalues. *Applied Mathematics Series. National Bureau of Standards* **49**, 47–81 (1958)
49. Sagadeeva, S., Boehm, M.: Sliceline: Fast, linear-algebra-based slice finding for ML model debugging. In: SIGMOD, pp. 2290–2299 (2021)
50. Schleich, M., Olteanu, D., Abo Khamis, M., Ngo, H.Q., Nguyen, X.: A layered aggregate engine for analytics workloads. In: SIGMOD, pp. 1642–1659 (2019)
51. Schleich, M., Olteanu, D., Ciucanu, R.: Learning linear regression models over factorized joins. In: SIGMOD, pp. 3–18 (2016)
52. Schmidt, E.: Zur Theorie der linearen und nichtlinearen Integralgleichungen. *Math. Ann.* **63**(4), 433–476 (1907)
53. Sharma, A., Paliwal, K.K., Imoto, S., Miyano, S.: Principal component analysis using QR decomposition. *Int. J. Mach. Learn. Cybern.* **4**(6), 679–683 (2013)
54. Sommer, J., Boehm, M., Evfimievski, A.V., Reinwald, B., Haas, P.J.: MNC: structure-exploiting sparsity estimation for matrix expressions. In: SIGMOD, pp. 1607–1623 (2019)
55. Stewart, G.W.: *Matrix Algorithms*. Soc. Ind. Appl. Math. (1998)
56. Strang, G.: *Linear Algebra and its Applications*. Thomson, Brooks/Cole (2006)
57. Van Loan, C.F.: The ubiquitous Kronecker product. *J. Comput. Appl. Math.* **123**(1–2), 85–100 (2000)
58. Wu, P., Chen, Z.: FT-ScaLAPACK: correcting soft errors on-line for ScaLAPACK Cholesky, QR, and LU factorization routines. In: High-Performance Parallel and Distributed Computing, pp. 49–60 (2014)
59. Yan, W.P., Larson, P.: Eager aggregation and lazy aggregation. In: VLDB, pp. 345–357 (1995)
60. Yelp: Yelp dataset challenge (2017). <https://www.yelp.com/dataset/challenge/>
61. Yuan, B., Jankov, D., Zou, J., Tang, Y., Bourgeois, D., Jermaine, C.: Tensor relational algebra for distributed machine learning system design. *Proc. VLDB Endow.* **14**(8), 1338–1350 (2021)
62. Zadeh, R.B., Meng, X., Ulanov, A., Yavuz, B., Pu, L., Venkataraman, S., Sparks, E.R., Staple, A., Zaharia, M.: Matrix computations and optimization in Apache Spark. In: SIGKDD, pp. 31–38 (2016)
63. Zihwei, C.: Extending an orthonormal rational set of vectors into an orthonormal rational basis. Unpublished online notes (2006)