LMFAO: An Engine for Batches of Group-By Aggregates

Layered Multiple Functional Aggregate Optimization

Maximilian Schleich
University of Washington
schleich@cs.washington.edu

Dan Olteanu
University of Zurich
olteanu@ifi.uzh.ch

ABSTRACT

LMFAO is an in-memory optimization and execution engine for large batches of group-by aggregates over joins. Such database workloads capture the data-intensive computation of a variety of data science applications.

We demonstrate LMFAO for three popular models: ridge linear regression with batch gradient descent, decision trees with CART, and clustering with Rk-means.

PVLDB Reference Format:
Maximilian Schleich and Dan Olteanu. LMFAO: An Engine for Batches of Group-By Aggregates. *PVLDB*, 13(12): 2945-2948, 2020.
DOI: https://doi.org/110.14778/3415478.3415515

1. LMFAO'S APPROACH TO LEARNING OVER RELATIONAL DATABASES

LMFAO is born out of the necessity to efficiently support ubiquitous data science workloads that involve learning models over relational queries [5]. From a database perspective, the data-intensive computation required by such workloads can be expressed as batches of group-by aggregates over the join of the underlying database relations. By tightly integrating the query processing and the learning tasks, LMFAO can outperform mainstream solutions based on TensorFlow and scikit-learn over Pandas by several orders of magnitude [5]. Such workloads pose new challenges to relational data processing engines as they require the computation of hundreds to thousands of similar yet distinct group-by aggregates over the natural join of database relations. Prior experiments with commercial and open-source database systems including MonetDB and PostgreSQL confirm that these challenges are not yet addressed satisfactorily by existing database technology [5].

To address these challenges, LMFAO puts forward a layered architecture of optimizations that chiefly target computation sharing at all data processing stages, factoring out repeated computation, and code specialization.

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/110.14778/3415478.3415515

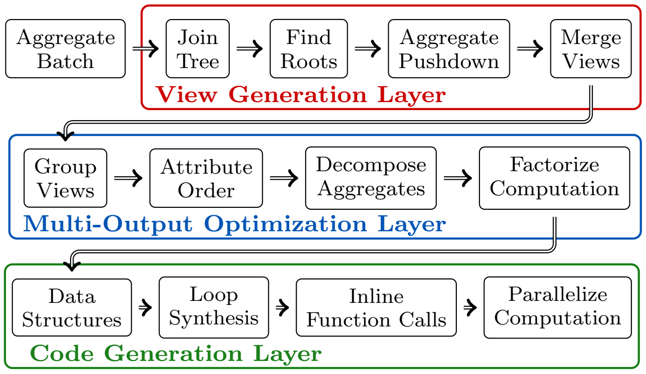


Figure 1: The layers of LMFAO.

We demonstrate LMFAO for three popular models: ridge linear regression using batch gradient descent, decision trees using CART [2], and clustering using Rk-means [3]. We learn them over commercial (Retailer [5]; 84M tuples) and public (Favorita [4]; 120M tuples) multi-relational datasets, which have been previously used for benchmarking LMFAO and its predecessors [6, 1, 5, 3]. Prior and on-going work by the authors (`https://fdbresearch.github.io`) showed that the LMFAO approach is useful for a variety of further discriminative and generative models, e.g., Generalized Linear Models, Support Vector Machines, (robust) PCA, Factorization Machines, and Sum-Product Networks.

2. LMFAO BY EXAMPLE

Figure 1 depicts the layered architecture of LMFAO. We next explain these layers using an example with three group-by aggregate queries over the Favorita dataset [4], whose schema is depicted in Figure 2; D is the natural join $\texttt{S} \bowtie \texttt{T} \bowtie \texttt{R} \bowtie \texttt{O} \bowtie \texttt{H} \bowtie \texttt{I}$ of all relations, h and g are user-defined aggregate functions returning numerical values.

Q_1 = `SELECT SUM(units) FROM D`
Q_2 = `SELECT store, SUM(g(item)*h(date)) FROM D GROUP BY store`
Q_3 = `SELECT class, SUM(units*price) FROM D GROUP BY class`

The **View Generation** layer takes the batch of queries, the database schema, and cardinality constraints (e.g., sizes of relations and attribute domains) and produces one query plan for all queries. The backbone of this plan is a join tree.

In the absence of group-by clauses, LMFAO computes each query Q in one bottom-up pass over the join tree by

Sales: date, store, item, units, promo
Holidays: date, htype, locale, transferred
StoRes: store, city, state, stype, cluster
Items: item, family, class, perishable
Transactions: date, store, txns
Oil: date, price

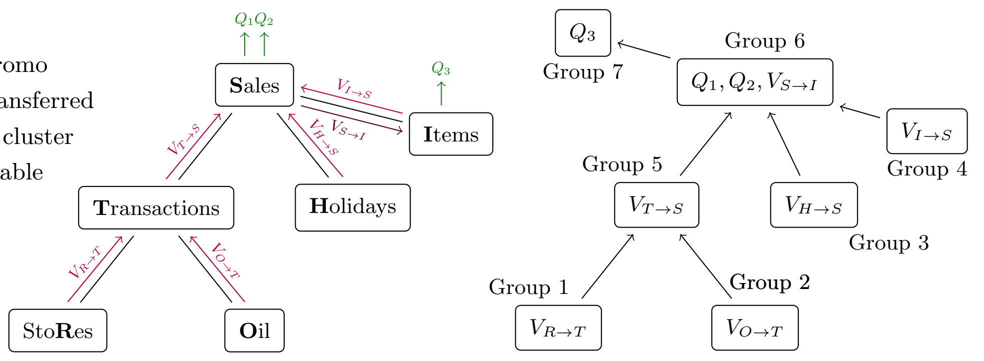


Figure 2: (left) The schema for the Favorita dataset. (middle) A join tree for this schema with directional views and three queries, partitioned in 7 groups. (right) The dependency graph of the groups of views and output queries.

decomposing it into views computed along each edge in the tree. The view at an edge going out of a node n computes the subquery that is the restriction of Q to the attributes in the subtree rooted at n and over the join of the relation at n and of the views at the incoming edges of n.

In the presence of group-by clauses, these views would have to carry the values for the group-by attributes along the paths from leaves to the root of the join tree. These views may be large and require significant compute time.

To alleviate this problem, one approach is to use a different join tree for each query so as to minimize the sizes of these views, e.g., by choosing a tree whose root has the group-by attributes of the query with the largest domains. This approach is however expensive as it would require to recompute the joins for each query. There is also no sharing of computation across the queries.

Instead, LMFAO compromises between the two aforementioned approaches. It uses one join tree for all queries, but assigns one root per query (using a simple heuristic [5]). Each query is thus decomposed into one view per edge in the join tree in a top-down traversal starting at its assigned root. This means that some edges may be traversed in both directions. This can reduce the sizes of the views and increase the sharing of their computation, thereby reducing the overall compute time. In our example, we choose Sales as root for Q_1 and Q_2, and Items as root for Q_3.

After each query is decomposed into views at edges in the join tree, LMFAO merges views whenever they have the same direction and group-by attributes. A single view may thus be used for several queries. Figure 2 (middle) depicts the merged views for Q_1, Q_2, and Q_3. Several edges in the join tree only have one view, which is used for all three queries.

The **Multi-Output Optimization** layer groups the views and output queries going out of a node such that they can be computed together over the join of the relation at the node and of its incoming views. For our running example, LMFAO groups Q_1, Q_2, and $V_{S\to I}$ because they can be computed together over the join of Sales with the incoming views $V_{T\to S}$, $V_{H\to S}$, and $V_{I\to S}$. The groups form a dependency graph as shown in Figure 2 (right).

LMFAO constructs a *multi-output execution plan* for each group that computes all of its outgoing views and output queries in one pass over the relation at the node and using lookups into the incoming views. This is yet another instance of sharing in LMFAO: The computation of different views share the scan of the relation at the node.

```
       $\beta_0 = 0;$
item   foreach $i \in \pi_{\text{item}}(S \bowtie_{\text{item}} V_{I\to S})$
         $\alpha_1 = V_{I\to S}(i);$   $\alpha_2 = g(i) \cdot \alpha_1;$
         $\beta_1 = 0;$
date     foreach $d \in \pi_{\text{date}}(\sigma_{\text{item}=i} S \bowtie_{\text{date}} V_{H\to S} \bowtie_{\text{date}} V_{T\to S})$
           $\alpha_3 = V_{H\to S}(d);$   $\alpha_4 = h(d) \cdot \alpha_2 \cdot \alpha_3;$
           $\beta_2 = 0;$
store      foreach $s \in \pi_{\text{store}}(\sigma_{\text{item}=i,\text{date}=d} S \bowtie_{\text{store}} \sigma_{\text{date}=d} V_{T\to S})$
             $\alpha_5 = V_{T\to S}(d, s);$
             $\beta_3 = 0;$   $\alpha_6 = |\sigma_{\text{item}=i,\text{date}=d,\text{store}=s} S| \cdot \alpha_4 \cdot \alpha_5;$
             foreach $u \in \pi_{\text{units}} \sigma_{\text{item}=i,\text{date}=d,\text{store}=s} S$
               $\beta_3 \mathrel{+}= u;$
             $\beta_2 \mathrel{+}= \beta_3 \cdot \alpha_5;$
             if $Q_2(s)$ then $Q_2(s) \mathrel{+}= \alpha_6$ else $Q_2(s) = \alpha_6;$
           $\beta_1 \mathrel{+}= \beta_2 \cdot \alpha_3;$
         $\beta_0 \mathrel{+}= \beta_1 \cdot \alpha_1;$   $V_{S\to I}(i) = \beta_1;$
       $Q_1 = \beta_0;$
```

Figure 3: Multi-output execution plan for Q_1, Q_2, $V_{S\to I}$.

The execution plan for a group is subject to fine-grained optimizations, e.g., factorized aggregate computation and shared computation. To enable them, LMFAO constructs a total order on the join attributes of the node relation. The relation and the incoming views are organized logically as tries: first grouped by the first attribute in the order, then by the next one in the context of values for the first, and so on. LMFAO then decomposes the group computation into simple arithmetic statements and lookups into incoming views that are executed at different levels in the tries.

Figure 3 exemplifies the execution plan for Group 6 in the dependency graph of Figure 2. The attribute order for the trie iteration is shown on the left. For simplicity of exposition, we assume that incoming and outgoing views are functions that map tuples over their group-by attributes to aggregates. The computation of the outgoing views is decomposed into partial aggregates, which are pushed past loops whenever possible (loop invariant code motion) and stored as local variables (α's) or running sums (β's). This code optimization decreases the number of arithmetic operations and dynamic accesses to incoming and outgoing views. For instance, we only look up into $V_{I\to S}$ once for each item value and not for each (item, date, store) triple. Similarly, we only update the result to Q_1 once at the very end. This optimization also allows for sharing computation across the group. For instance, $V_{S\to I}$ shares most of its computation with Q_1, reflected by the running sum β_1.

Finally, the **Code Generation** layer compiles the multi-output execution plan for each group into efficient, low-level C++ code specialized to the database schema and the join tree. This layer also performs low-level code optimizations, e.g., optimizing cache locality, choosing data structures for the views such as sorted arrays and (un)ordered hashmaps, and inlining function calls. LMFAO computes the groups in parallel by exploiting both task and domain parallelism.

3. FROM LEARNING TO AGGREGATES

We next show the aggregates needed for learning the three models. LMFAO computes these aggregates over the non-materialised dataset D, which is defined by a feature extraction query with n attributes over a multi-relational database.

Linear Regression models are linear functions:

$$LR(\boldsymbol{x}) = \langle \boldsymbol{\theta}, \boldsymbol{x} \rangle = \sum_{j\in[n]} \theta_j \cdot x_j.$$

with parameters $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_n)$ and feature vector $\boldsymbol{x} = (x_1, \ldots, x_n)$. We assume without loss of generality that (1) x_1 only takes value 1 and θ_1 is the intercept of the model, and (2) the label is part of the feature vector $\boldsymbol{x}$ and its corresponding parameter is fixed to -1.

We learn the parameters $\boldsymbol{\theta}$ using batch gradient descent (BGD), which requires the computation of the least-squares objective function $J(\boldsymbol{\theta})$ and its gradient $\boldsymbol{\nabla} J(\boldsymbol{\theta})$:

$$J(\boldsymbol{\theta}) = \frac{1}{2|D|}\boldsymbol{\theta}^\top \Big(\sum_{\mathbf{x}\in D} \boldsymbol{x}\boldsymbol{x}^\top\Big)\boldsymbol{\theta} + \frac{\lambda}{2}\,\|\boldsymbol{\theta}\|^2$$

$$\boldsymbol{\nabla} J(\boldsymbol{\theta}) = \frac{1}{|D|}\Big(\sum_{\mathbf{x}\in D} \boldsymbol{x}\boldsymbol{x}^\top\Big)\boldsymbol{\theta} + \lambda\,\boldsymbol{\theta}$$

The data-intensive computation of the optimization algorithm is given by $\Sigma = \sum_{\mathbf{x}\in D} \boldsymbol{x}\boldsymbol{x}^\top$, which defines the non-centered covariance matrix. The (j,k)-entry in Σ accounts for the pairwise multiplication of attributes X_j and X_k. LMFAO computes each of these entries as one aggregate query.

If both X_j and X_k are continuous attributes, we compute:

```
SELECT SUM(X_j * X_k) FROM D
```

Categorical attributes are one-hot encoded in a linear regression model. In LMFAO, such attributes become group-by attributes. If only X_j is categorical, we compute:

```
SELECT X_j,SUM(X_k) FROM D GROUP BY X_j
```

If both X_j and X_k are categorical, we compute instead:

```
SELECT X_j,X_k,SUM(1) FROM D GROUP BY X_j,X_k
```

For the Retailer dataset, LMFAO computes 814 aggregates to learn the linear regression model [5]. Since Σ does not depend on the parameters $\boldsymbol{\theta}$, the aggregates are computed once and then reused for all BGD iterations.

Decision Trees are popular machine learning models that use trees with inner nodes representing conditional control statements to model decisions and their consequences. Leaf nodes represent predictions for the label. We focus on learning decision trees for regression scenarios.

We learn the tree with the seminal CART algorithm [2], which greedily constructs the tree one note at a time. The algorithm learns binary trees, with the inner nodes representing threshold conditions X_j op t, where op $\in \{\leq, \geq, =, \neq\}$. For each node N, the algorithm explores all attributes X_j and possible thresholds t_j to find the condition X_j op t_j that minimizes the variance of the label Y:

$$\texttt{VARIANCE} = \sum_{(\mathbf{x},y)\in T} y^2 - \frac{1}{|T|}\Big(\sum_{(\mathbf{x},y)\in T} y\Big)^2$$

where T is the fragment of the dataset D that satisfies the condition X_j op t and all conditions along the path from the root to N. The algorithm thus requires the aggregates SUM(1), SUM(Y), and SUM(Y^2) over T, which can be computed in one query over D:

```
SELECT SUM(1),SUM(Y),SUM(Y^2) FROM D WHERE cond
```

where cond is the conjunction of X_j op t and all threshold conditions along to the path from root to current node.

For the Retailer dataset, LMFAO computes 3,141 aggregate queries for each node in the decision tree [5].

Rk-means computes a constant-factor approximation of the k-means clustering objective by computing the k clusters over a small coreset of D [3]. A coreset of D is a small set of points that provide a good summarization of the original dataset D. Rk-means constructs a so-called *grid coreset*, which is defined as the Cartesian product of cluster centroids computed over the projections on each attribute of D.

Given the feature extraction query that defines D and the constant k that defines the number of clusters, Rk-means clusters the dataset D in four steps.

Step 1. We project D onto each attribute X_j and compute the weight for each point in the projection. This can be computed as one query for each attribute X_j:

```
SELECT X_j, SUM(1) FROM D GROUP BY X_j
```

Step 2. We perform weighted k-means on each projection. We assume that the algorithm returns a "cluster assignment" relation A_j which records for each $\boldsymbol{x} \in \pi_{X_j}(D)$ the closest centroid C_j in the projection.

Step 3. Using the results of these clusterings we assemble a cross-product weighted grid G of centroids, which defines the coreset of D. A grid point $\boldsymbol{g}$ in the coreset is composed of tuples of size n, with the value in dimension $j \in [n]$ ranging over the possible cluster means for the projection on X_j computed in Step 2. The weight of a grid point $\boldsymbol{g} \in G$ is the number of data points in D closest to the grid point. The grid coreset G and the grid point weights can be computed with one aggregate query:

```
SELECT C_1,...,C_n,SUM(1) FROM P GROUP BY C_1,...,C_n
```

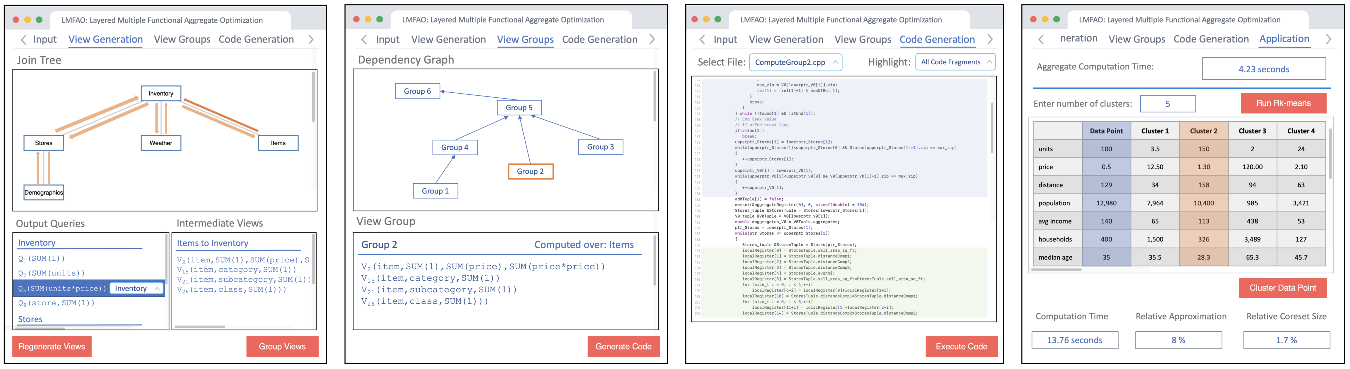
where $\mathtt{P} = \mathtt{D} \bowtie \mathtt{A_1(X_1, C_1)} \bowtie \cdots \mathtt{A_n(X_n, C_n)}$ is the join of D and the cluster assignments from Step 2.

Step 4. Finally, we perform weighted k-means clustering on the coreset G to compute the desired result of k centroids.

We use LMFAO to compute steps 1 and 3 of the algorithm. This requires $n + 1$ queries.

4. DEMONSTRATION SCENARIOS

We next describe how users can interact with LMFAO's user interface. Figure 4 depicts snapshots of the interface.



Inventory
Q_1(SUM(1))
Q_2(SUM(units))
Q_3(SUM(units*price)) Inventory
Q_8(store,SUM(1))
Stores

Items to Inventory
V_2(item,SUM(1),SUM(price),S
V_{15}(item,category,SUM(1))
V_{21}(item,subcategory,SUM(1)
V_{26}(item,class,SUM(1)))

Group 2 Computed over: Items
V_2(item,SUM(1),SUM(price),SUM(price*price))
V_{15}(item,category,SUM(1))
V_{21}(item,subcategory,SUM(1))
V_{26}(item,class,SUM(1)))

Select File: ComputeGroup2.cpp Highlight: All Code Fragments

```
            max_zip = V0[lowerptr_V0[1]].zip;
            rel[1] = (rel[1]+1) % numOfRel[1];
         }
         break;
      }
   } while (!found[1] && !atEnd[1]);
   // End Seek Value
   // If atEnd break loop
   if(atEnd[1])
      break;
   upperptr_Stores[1] = lowerptr_Stores[1];
   while(upperptr_Stores[1]<upperptr_Stores[0] && Stores[upperptr_Stores[1]+1].zip == max_zip)
   {
      ++upperptr_Stores[1];
   }
   upperptr_V0[1] = lowerptr_V0[1];
   while(upperptr_V0[1]<upperptr_V0[0] && V0[upperptr_V0[1]+1].zip == max_zip)
   {
      ++upperptr_V0[1];
   }
   addTuple[1] = false;
   memset(&aggregateRegister[0], 0, sizeof(double) * 104);
   Stores_tuple &StoresTuple = Stores[lowerptr_Stores[1]];
   V0_tuple &V0Tuple = V0[lowerptr_V0[1]];
   double *aggregates_V0 = V0Tuple.aggregates;
   ptr_Stores = lowerptr_Stores[1];
   while(ptr_Stores <= upperptr_Stores[1])
   {
      Stores_tuple &StoresTuple = Stores[ptr_Stores];
      localRegister[0] = StoresTuple.sell_area_sq_ft;
      localRegister[1] = StoresTuple.distanceComp1;
      localRegister[2] = StoresTuple.distanceComp2;
      localRegister[3] = StoresTuple.distanceComp3;
      localRegister[4] = StoresTuple.avghhi;
      localRegister[5] = StoresTuple.sell_area_sq_ft*StoresTuple.sell_area_sq_ft;
      for (size_t i = 0; i < 4;++i)
         localRegister[6+i] = localRegister[0]*localRegister[1+i];
      localRegister[10] = StoresTuple.distanceComp1*StoresTuple.distanceComp1;
      for (size_t i = 0; i < 3;++i)
         localRegister[11+i] = localRegister[1]*localRegister[2+i];
      localRegister[14] = StoresTuple.distanceComp2*StoresTuple.distanceComp2;
```

Aggregate Computation Time: 4.23 seconds

Enter number of clusters: 5

	Data Point	Cluster 1	Cluster 2	Cluster 3	Cluster 4
units	100	3.5	150	2	24
price	0.5	12.50	1.30	120.00	2.10
distance	129	34	158	94	63
population	12,980	7,964	10,400	985	3,421
avg income	140	65	113	438	53
households	400	1,500	326	3,489	127
median age	35	35.5	28.3	65.3	45.7

Computation Time: 13.76 seconds
Relative Approximation: 8 %
Relative Coreset Size: 1.7 %

(a) View Generation **(b)** View Groups **(c)** Code Generation **(d)** Rk-Means Application

Figure 4: Snapshots of LMFAO's user interface. Users first select a database to load and an ML application. Then, users can (a) inspect and modify the root assignment and the generated views; (b) review the grouping of views; (c) dive into the generated code for each view group; and (d) compute the ML application and analyze its performance and output.

In the *Input* tab (not shown), the user chooses the database and one of three machine learning scenarios: (1) learning linear regression models, (2) learning regression trees, and (3) clustering using Rk-means. After selecting the dataset, the tab depicts the join tree and database schema, so that the user can inspect it. LMFAO then generates the batch of aggregates for the respective application.

Next, LMFAO computes the root assignment for each aggregate query and generates the corresponding views. The top of the *View Generation* tab depicts the join tree annotated by intermediate views, which are shown as arrows along the edges. The width of the arrow indicates the number of views computed in this direction. Below, the tab lists the output queries and intermediate views, where the output queries are grouped by their root node, and intermediate views are grouped by their directions. By default all queries and views are shown. If the user selects a node in the join tree, only the output queries and intermediate views that are computed over this node are listed. Similarly, by selecting one of the arrows, only the views computed in the direction of the arrow are shown. Figure 4 (a) depicts the selection of the arrow from Items to Inventory.

When selecting a query in the output query list, a drop-down list for the root of this query is shown. This allows the user to reassign the query to a different root and change the views that are generated. The views for the new root assignment are then regenerated and the join tree is updated.

The *View Groups* tab depicts the dependency graph of the view groups. The user can inspect the view groups by selecting the corresponding node.

The *Code Generation* tab depicts the C++ code that is generated for a given view group. Different types of code fragments are highlighted, e.g., the computation of the join, aggregates, or running sums. The user can choose to highlight all code fragments, or only one of them.

The user can execute the code for the aggregate computation and inspect the application that is computed over the aggregates. Since the execution takes a few seconds in LMFAO, we will run it on the fly during the demonstration. Figure 4 (d) depicts the interface for Rk-means clustering, the interface for the other two applications is similar. At the top, we show the time it took to compute the aggregates for clustering in each dimension. The user enters the desired number of clusters and then runs Rk-means. Once computed, the interface presents the cluster centroids. It also allows the user to enter the values for a data point, and find the centroid that is closest to this point. The data point entered in Figure 4 (d) is closest to the highlighted centroid for Cluster 2. We further present the time it took to compute the clusters, the relative approximation of the clusters, and the relative size of the grid coreset with respect to the size of the dataset D. For the approximation, we compute the intra-cluster distance, and take the difference between the distances for Rk-means and the conventional Lloyd's algorithm relative to the distance for Lloyd's. We report the average relative difference over ten precomputed runs of Lloyd's algorithm.

Acknowledgements

Olteanu acknowledges a research gift from Infor. Schleich is supported by a RelationalAI fellowship. The authors acknowledge Haozhe Zhang for his help with the user interface. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 682588.

5. REFERENCES

[1] M. Abo Khamis, H. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. AC/DC: In-database learning thunderstruck. In *DEEM*, pages 8:1–8:10, 2018.

[2] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Wadsworth and Brooks, Monterey, CA, 1984.

[3] R. Curtin, B. Moseley, H. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. Rk-means: Fast clustering for relational data. In *AISTATS*, pages 2742–2752, 2020.

[4] C. Favorita. 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/`.

[5] M. Schleich, D. Olteanu, M. Abo Khamis, H. Ngo, and X. Nguyen. A layered aggregate engine for analytics workloads. In *SIGMOD*, pages 1642–1659, 2019.

[6] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *SIGMOD*, pages 3–18, 2016.