

Learning Linear Regression Models over Factorized Joins

Maximilian Schleich

Dan Olteanu

Radu Ciucanu

Department of Computer Science, University of Oxford
{max.schleich,dan.olteanu,radu.ciucanu}@cs.ox.ac.uk

*Cultural Learnings of Factorized Joins for Make
Benefit Glorious Family of Regression Tasks¹*

ABSTRACT

We investigate the problem of building least squares regression models over training datasets defined by arbitrary join queries on database tables. Our key observation is that joins entail a high degree of redundancy in both computation and data representation, which is not required for the end-to-end solution to learning over joins.

We propose a new paradigm for computing batch gradient descent that exploits the factorized computation and representation of the training datasets, a rewriting of the regression objective function that decouples the computation of cofactors of model parameters from their convergence, and the commutativity of cofactor computation with relational union and projection. We introduce three flavors of this approach: **F/FDB** computes the cofactors in one pass over the materialized factorized join; **F** avoids this materialization and intermixes cofactor and join computation; **F/SQL** expresses this mixture as one SQL query.

Our approach has the complexity of join factorization, which can be exponentially lower than of standard joins. Experiments with commercial, public, and synthetic datasets show that it outperforms MADlib, Python StatsModels, and R, by up to three orders of magnitude.

1. INTRODUCTION

There is increasing interest in academia and industry in building systems that integrate databases and machine learning [17, 34, 1, 18]. This is driven by web companies, e.g., Google Brain, Twitter, Facebook’s DeepFace, or Microsoft’s platform, though it is also prominent in other industry segments such as retail-planning and forecasting applications [2]. State-of-art commercial analytics systems support descriptive, or backward-looking analytics, predictive, or forward-

looking analytics such as classification and regression, and prescriptive analytics, which are also forward-looking and usually take the output of a predictive model as input. The typical data sources of interest are weekly sales data, promotions, and product descriptions. In these settings, the input to analytics is a relation representing the natural join of those data sources stored in a database. A typical prediction a retailer would like to compute is the additional demand generated for a given product due to promotion.

In our exploration with such systems on a real dataset from a large US retailer and following discussions with industry experts [23], we realized three major shortcomings of these systems: (1) Poor integration of analytics and databases, which are traditionally confined to distinct specialized systems in the ever-growing technology stack [2]; (2) poor efficiency already for few data sources; and (3) insufficient accuracy due to omission of further relevant data sources.

The use of typical data sources already stretches the scalability of existing systems and the current processing regime is to manually partition the data, e.g., by market and category, and run analytics independently on each partition [23]. Domain-expertise focuses on how to best partition the data, to the point where it becomes black art. Data partitioning misses relevant correlations and prohibits common forecast patterns across different categories and markets. By processing separately by market/category, today’s forecasting cannot leverage similar demand behavior across geography or time. For this reason, data partitioning is seen as a strong limitation of the state of the art and preference is given to running analytics on the entire dataset.

More data sources would enable forecasting with higher accuracy and at a more granular level, such as customer reviews, basket data transactions, competitive promotions and prices, flu trends, loyalty program history, customer transaction history, social media text related to the retailer products sold, store attributes such as demographics, weather, or nearby competition. Incorporating more data sources would allow customer-specific promotions and separate out the impact of actions taken by the retailer (e.g., changing discounts and prices) from weather and environment [23]. However, it can increase the load by several orders of magnitude as the size of the input to analytics explodes.

A feasible approach to address these shortcomings would at least need to support efficient joining of many relations and running in-database analytics on large join results.

The main contribution of this paper is a batch gradient descent approach that can build linear regression models on training datasets defined by arbitrary join queries on tables.

¹With apologies to Sacha Baron Cohen and Borat

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD’16, June 26–July 01, 2016, San Francisco, CA, USA

© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...\$15.00

DOI: <http://dx.doi.org/10.1145/2882903.2882939>

Our key observation is that the intermediate join step represents the main bottleneck and it is unnecessarily expensive. It entails a high degree of redundancy in both computation and data representation, yet this is not required for the end-to-end solution, whose result is a list of real-valued parameters of the learned model. By computing a *factorized join* [5] instead of the standard flat join, we reduce data redundancy and improve performance for both the join and the learning steps. The theoretical and practical gains in both required memory space and time performance for factorized joins are well-understood and can be asymptotically exponential in the size of the join [28], which translates to orders of magnitude for various datasets reported in experiments [5, 4].

Our learning approach is based on several contributions:

- It uses an algebraic rewriting of the regression’s objective function that decouples the computation of cofactors of model parameters from their convergence.
- Cofactor computation commutes with relational union and projection. The commutativity with union enables the computation of the parameter cofactors for the entire dataset as the sum of corresponding cofactors for disjoint partitions of the input dataset. This property is essential for efficiency in a centralized setting and also desirable for *concurrent learning*, where cofactors of partitions are computed on different cores or machines. The commutativity with projection allows us to compute all cofactors once and then explore the space of possible models by only running convergence for a subset of parameters. This property is desirable for *model selection*, whose goal is to find a subset of features that best predict a test dataset.
- We introduce three flavors of our approach: **F/FDB** uses FDB [5] to materialize the factorized join and computes the cofactors in one pass over it; our baseline **F** avoids this materialization and blends cofactor and join computation; and **F/SQL** expresses this mixture of joins and cofactors as one optimized SQL query.
- Given a database **D** and a join query Q , our approach needs $O(|\mathbf{D}|^{fhtw(Q)})$ time (modulo log factors), which is *worst-case optimal* for computing the factorized join whose nesting structure is defined by a hypertree decomposition of the query Q ; in contrast, any relational engine can achieve at best $O(|\mathbf{D}|^{\rho^*(Q)})$ [3, 26, 41]. The gap between the fractional hypertree width $fhtw(Q)$ and the fractional edge cover number $\rho^*(Q)$ can be as large as the number of relations in Q . For instance, $fhtw = 1$ for acyclic queries and our approach takes linear time. Learning over traditional flat joins is thus bound to be suboptimal.
- We can also learn linear functions over arbitrary basis functions, which include feature interactions.
- We benchmarked **F** and its variants against R, which uses QR decomposition [15], Python StatsModels [40], which uses Moore-Penrose pseudoinverse to compute closed-form solutions (ols) [29], and MADlib [17], which also supports ols and Newton optimization for generalized linear models. In our experiments with public, commercial, and artificial datasets, **F** outperforms these competitors by up to three orders of magnitude while preserving the same accuracy.

2. FACTORIZED DATABASES: A PRIMER

In this paper, we rely on factorized databases to compute and represent join results that are input to learning regression models. We next introduce such databases by example and refer to the literature [5, 4, 28] for a rigorous treatment.

Factorized databases form a representation system for relational data that exploits laws of relational algebra, such as the distributivity of the Cartesian product over union, to reduce data and computation redundancy.

EXAMPLE 2.1. Figure 1(a) depicts a database consisting of three relations along with their natural join: The relation *House* records house prices and living areas (in squared meters) within locations given by zipcodes; *TaxBand* relates city/state tax bands with house living areas; *Shops* list shops with zipcode and opening hours (in our experiments, we consider an extended dataset from a large US retailer; the join condition may also use a user-defined distance function).

The join result exhibits a high degree of redundancy. The value z_1 occurs in 24 tuples, each value h_1 to h_3 occurs in eight tuples and they are paired with the same combinations of values for the other attributes. Since z_1 is paired in *House* with p_1 to p_3 and in *Shops* with h_1 to h_3 , all combinations (indeed, the Cartesian product) of the former and the latter values occur in the join result. We can represent this local product symbolically instead of eagerly materializing it. If we systematically apply this observation, we obtain an equivalent *factorized representation* of the entire join result that is much more compact than the flat, tabular representation of the join result, cf. Figure 1(c) (the attribute names are clear from context). Each tuple in the result is represented once in the factorized join and can be constructed by following one branch of every union and all branches of a product. Whereas the flat join has 130 values (26 tuples of 5 values each), the factorized join only has 18 values.

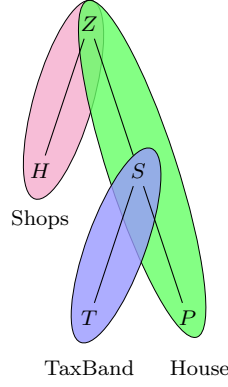
The factorized join in Figure 1(c) has the nesting structure depicted in Figure 1(b): It is a union of Z -values occurring in the join of *Shops* and *House* on Z . For each Z -value z , we represent separately the union of H -values paired with z in *Shops* and the union of S -values paired with z in *House* and with T -values in *TaxBand*. That is, given z , the H -values are (*conditionally*) *independent* of the S -values and can be stored separately. This is where the factorization saves computation and space as it avoids an explicit enumeration of all combinations of H -values and S -values for a given Z -value. Also, under each S -value, there is a union of T -values and a union of P -values. The factorization can be compacted further by *caching* subexpressions [28]: The S -value s_2 occurs with its union of T -values $t_4 \cup t_5$ from *TaxBand*, regardless of which Z -values s_2 is paired with in *House*. This T -union can be stored once and reused for every occurrence of s_2 . \square

The nesting structures of factorized joins are called *d-trees* [28]. They are rooted forests representing partial orders of join variables. Each variable A in a d-tree Δ has a subset *key*(A) of the set *anc*(A) of its ancestor variables on which A and its descendant variables may depend. A d-tree satisfies the following constraints. (1) The variables of each relation symbol in Q lie along the same root-to-leaf path (since they depend on each other). (2) For any child B of a variable A , *key*(B) \subseteq *key*(A) \cup { A }. Caching of factorizations rooted at unions of A -values is useful when *key*(A) \subset *anc*(A), since A and its descendant variables do not depend on variables in *anc*(A) \setminus *key*(A) and thus factorizations rooted at A -values

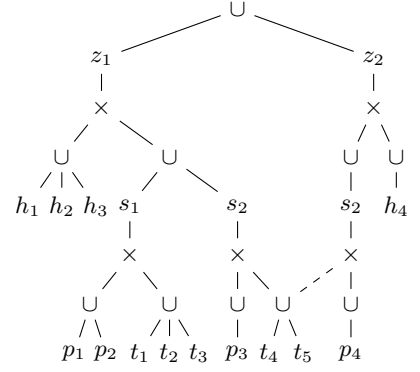
Shops		House			Shops \bowtie House \bowtie TaxBand				
Z	H	Z	S	P	Z	H	S	P	T
z_1	h_1	z_1	s_1	p_1	z_1	h_1	s_1	p_1	t_1
z_1	h_2	z_1	s_1	p_2	z_1	h_1	s_1	p_1	t_2
z_1	h_3	z_1	s_2	p_3	z_1	h_1	s_1	p_1	t_3
z_2	h_4	z_2	s_2	p_4	z_1	h_1	s_1	p_2	t_1
					z_1	h_1	s_1	p_2	t_2
					z_1	h_1	s_1	p_2	t_3
					z_1	h_1	s_2	p_3	t_4
					z_1	h_1	s_2	p_3	t_5

TaxBand	
S	T
s_1	t_1
s_1	t_2
s_1	t_3
s_2	t_4
s_2	t_5

(a) The three relations of database \mathbf{D} and natural join $Q(\mathbf{D})$.



(b) D-tree Δ .



(c) Factorization of $Q(\mathbf{D})$ over Δ .

Figure 1: (a) Database \mathbf{D} with relations **House**(Zipcode, Sqm, Price), **TaxBand**(Sqm, Tax), **Shops**(Zipcode, Hours), where the attribute names are abbreviated and the values are not necessarily distinct; (b) Nesting structure (d-tree) Δ for the natural join of the relations; (c) Factorization $\Delta(\mathbf{D})$ of the natural join over Δ .

repeat for every tuple of values for these variables. For instance, $key(T) = \{S\} \subset anc(T) = \{S, Z\}$.

The construction of d-trees is guided by the joins, cardinalities, and join selectivities. They can lead to factorizations of greatly varying sizes, where the size of a representation (flat or factorized) is defined as the number of its values. Within the class of factorizations over d-trees, we can find the worst-case optimal ones and also compute them in worst-case optimal time:

PROPOSITION 2.2. *Given a join query Q , for every database \mathbf{D} , the join result $Q(\mathbf{D})$ admits*

- a flat representation of size $\Theta(|\mathbf{D}|^{\rho^*(Q)})$ [3];
- a factorization over d-trees of size $\Theta(|\mathbf{D}|^{fhtw(Q)})$ [28].

There are worst-case optimal join algorithms to compute the join result in these representations [26, 28].

The measures $\rho^*(Q)$ and $fhtw(Q)$ are the fractional edge cover number and the fractional hypertree width respectively. We know that $1 \leq fhtw(Q) \leq \rho^*(Q) \leq |Q|$, where $|Q|$ is the number of relations in query Q [28]. The gap between $fhtw(Q)$ and $\rho^*(Q)$ can be as large as $|Q|$. The fractional hypertree width is fundamental to problem tractability with applications spanning constraint satisfaction, databases, matrix operations, logic, and probabilistic graphical models [19].

A key observation is that aggregates defined by arithmetic expressions over data values with operations summation and multiplication, e.g., count and sum, can be computed in one pass over factorized joins [4]; we only need to change the union-Cartesian product semiring to the sum-multiplication semiring. For instance, to count the number of tuples of a relation represented by a subtree of the factorized join, we interpret each value as 1 and turn unions and Cartesian products into sums and multiplication respectively. The count at the topmost union node is then the number of tuples in the query result. To sum over all values of an attribute A , the only difference to the previous count algorithm is that we now preserve the A -values. The core computation underlying the construction of linear regression models concerns a family of such aggregates.

3. LEARNING REGRESSION MODELS OVER FACTORIZED JOINS

We are given a training dataset of size m that is computed as a join of database tables:

$$\{(y^{(1)}, x_1^{(1)}, \dots, x_n^{(1)}), \dots, (y^{(m)}, x_1^{(m)}, \dots, x_n^{(m)})\}.$$

The values $y^{(i)}$ are called *labels* and the other values are called *features*. They are all real numbers. For our training dataset, a natural label would be P to predict the price given the other features. Our goal is to learn the parameters $\theta = (\theta_0, \dots, \theta_n)$ of the linear function:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \dots + \theta_n x_n$$

that approximates the label y of unseen tuples (x_1, \dots, x_n) . For uniformity, we add $x_0 = 1$ so that $h_\theta(x) = \sum_{k=0}^n \theta_k x_k$.

The error of our model is given by the so-called least squares regression objective function:

$$J(\theta) = \frac{1}{2} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2. \quad (1)$$

This is a much studied objective function, since minimizing sum-of-squared errors is equivalent to finding a maximum likelihood solution under a Gaussian noise model [6].

We consider the *batch gradient descent* (BGD) [6], which repeatedly updates the parameters of h_θ in the direction of the gradient to decrease the error given by $J(\theta)$:

$$\begin{aligned} \forall 0 \leq j \leq n : \theta_j &:= \theta_j - \alpha \frac{\delta}{\delta \theta_j} J(\theta) \\ &:= \theta_j - \alpha \sum_{i=1}^m \left(\sum_{k=0}^n \theta_k x_k^{(i)} - y^{(i)} \right) x_j^{(i)}. \end{aligned}$$

The value α is the learning rate and can adapt with the iterations. A naïve implementation of the above expression would start with some initial values for parameters θ_k and perform one pass over the dataset to compute the value of the sum aggregate, followed by one approximation step for the parameters, and repeat this process until convergence.

This is not practical since it is inefficient to go over the entire dataset for each iteration. BGD can however become very competitive when adapted to work on factorized joins.

BGD has two logically independent tasks that are intertwined in the above expression: The computation of the sum aggregate and the convergence of the parameters. The sum aggregate can be rewritten so that the label y becomes part of the features and has a predefined parameter $\theta = -1$:

$$\forall 0 \leq j \leq n : S_j = \sum_{i=1}^m \left(\sum_{k=0}^n \theta_k x_k^{(i)} \right) x_j^{(i)} \quad (2)$$

Our approach is guided by two main insights.

Our **first insight** is that the sum aggregates can be rewritten so that we explicate the *cofactor* of each parameter θ_k in each sum aggregate S_j :

$$\forall 0 \leq j \leq n : S_j = \sum_{k=0}^n \theta_k \times \text{Cofactor}[k, j] \quad (3)$$

$$\text{where } \text{Cofactor}[k, j] = \sum_{i=1}^m x_k^{(i)} x_j^{(i)} \quad (4)$$

This reformulation allows to decouple cofactor computation from parameter convergence. This is crucial for performance as we do not require to scan the dataset for each approximation step. Our system **F** computes the cofactors once and performs parameter convergence directly on the matrix of cofactors, whose size is independent of the data size m . For convergence, **F** uses an adaptation of AdaGrad [13].

Furthermore, the cofactor matrix has desirable properties:

PROPOSITION 3.1. *Let (Q, \mathbf{D}) be a pair of join query Q and database \mathbf{D} defining the training dataset $Q(\mathbf{D})$ with schema/features $\sigma = (A_0, \dots, A_n)$. Let Cofactor be the cofactor matrix for learning the parameters $\theta_{A_0}, \dots, \theta_{A_n}$ of the function $f_\theta = \sum_{k=0}^n (\theta_{A_k} x_{A_k})$ using batch gradient descent.*

The cofactor matrix has the following properties:

1. Cofactor is symmetric:

$$\forall 0 \leq k, j \leq n : \text{Cofactor}[A_k, A_j] = \text{Cofactor}[A_j, A_k].$$

2. Cofactor computation commutes with union: Given training datasets $Q(\mathbf{D}_1), \dots, Q(\mathbf{D}_p)$ with cofactor matrices $\text{Cofactor}_1, \dots, \text{Cofactor}_p$ where $\mathbf{D} = \bigcup_{l=1}^p \mathbf{D}_l$, then

$$\forall 0 \leq k, j \leq n : \text{Cofactor}[A_k, A_j] = \sum_{l=1}^p \text{Cofactor}_l[A_k, A_j].$$

3. Cofactor computation commutes with projection: Given a feature set $L \subseteq \sigma$ and the cofactor matrix Cofactor_L for the training dataset $\pi_L(Q(\mathbf{D}))$, then

$$\forall 0 \leq k, j \leq n \text{ such that } A_k, A_j \in L : \\ \text{Cofactor}_L[A_k, A_j] = \text{Cofactor}[A_k, A_j].$$

The symmetry property implies that we only need to compute the upper half of the cofactor matrix.

The commutativity with union means that the cofactor matrix for the union of several training datasets is the entry-wise sum of the cofactor matrices of these training datasets. This property is key to the efficiency of our approach, since we can locally compute partial cofactors over different partitions of the training dataset and then add them up. It is also desirable for *concurrent computation*, where partial cofactors can be computed on different cores or machines.

The commutativity with projection allows us to use the cofactor matrix to compute any subset of the parameters: All it takes is to ignore from the matrix the columns and rows for the irrelevant parameters. This is beneficial if some features are necessary for constructing the dataset but irrelevant for learning, e.g., relation keys supporting the join such as zipcode in our training dataset in Figure 1(a). It is also beneficial for *model selection*, a key challenge in machine learning centered around finding the subset of features that best predict a test dataset. Model selection is a laborious and time-intensive process, since it requires to learn independently parameters corresponding to subsets of the available features. With our reformulation, we first compute the cofactor matrix for all features and then perform convergence on top of the cofactor matrix for the entire lattice of parameters independently of the data. Besides choosing the features after cofactor computation, we may also choose the label and fix its model parameter to -1.

The two commutativity properties in Proposition 3.1 hold under bag (SQL) semantics in the sense that the relational projection and union operators do not remove duplicates. This is important, since learning is sensitive to duplicates.

Our **second insight** is that we can compute the cofactors in one pass over *any factorized join* representing the training dataset, which has the flat join as a special case:

PROPOSITION 3.2. *Let (Q, \mathbf{D}, Δ) be a triple of a join query Q , a database \mathbf{D} , and a d -tree Δ of Q . Let the training dataset be the factorized join $\Delta(\mathbf{D})$ with features (A_0, \dots, A_n) , and let Cofactor be the cofactor matrix for learning the parameters $\theta_{A_0}, \dots, \theta_{A_n}$ of the function $\sum_{k=0}^n (\theta_{A_k} x_{A_k})$ using batch gradient descent. Then, Cofactor can be computed in one pass over the factorized join $\Delta(\mathbf{D})$.*

Section 4 gives an algorithm to compute the cofactor matrix over the materialized factorized join. Section 5 then shows how to avoid this materialization by intertwining cofactor and join computation. Finally, Section 6 shows how to encode the previous algorithm in one SQL query. An immediate implication is that the redundancy in the flat join result is not necessary for learning:

THEOREM 3.3. *The parameters of any linear function over features from a training dataset defined by a database \mathbf{D} and a join query Q can be learned in time $O(|\mathbf{D}|^{f_{\text{htw}}(Q)})$.*

Theorem 3.3 is a direct corollary of Propositions 2.2 and 3.2. We recall our discussion in Section 2 that within the class of factorized representations over d -trees, this time complexity is essentially worst-case optimal in the sense that there is no join algorithm that can achieve a lower worst-case time complexity. To put this result into a broader context, any worst-case optimal join algorithm that would produce flat join results, such as NPRR [26] or LogicBlox's LeapFrog TrieJoin [41], would need time at least $O(|\mathbf{D}|^{\rho^*(Q)})$ to create the training dataset, yet the gap between $\rho^*(Q)$ and $f_{\text{htw}}(Q)$ can be as large as the number of relations in the join query.

Our approach extends to *linear functions* $\sum_{k=0}^n \theta_k \phi_k(\bar{x})$ with arbitrary basis functions ϕ_k over a tuple \bar{x} of features. We previously discussed the identity basis functions $\phi_k(x_k) = x_k$. Further examples are polynomials and Gaussian Radial Basis Functions [6]. While basis functions over a single feature x_k are trivially supported, feature interactions are challenging as they may restrict the structure of the factorized

join. For instance, the basis function $\phi_k(x_i, x_j) = x_i \cdot x_j$ can only be supported efficiently by d-trees where the attributes x_i and x_j are along the same root-to-leaf path as if they were attributes of a same relation, since we require to compute all possible combinations of values for x_i and x_j . We can enforce this path constraint by enriching the database with one (not materialized) relation over the schema $R_k(x_i, x_j)$ and the query Q with a natural join with R_k . The d-trees for the enriched query will necessarily satisfy the new path constraint and the factorization will have a new value for every combination of values for x_i and x_j along a same path. We can thus add to the factorization a value $\langle \phi_k : \phi_k(x_i, x_j) \rangle$ under each pair of values for x_i and x_j .

We can rephrase Theorem 3.3 for linear functions with basis functions as follows. We say that the basis functions ϕ_0, \dots, ϕ_b over the sets of features S_0, \dots, S_b induce a relational schema $\sigma = (R_0(S_0), \dots, R_b(S_b))$. Given a join query Q and the above schema σ , an *extension of Q with respect to σ* is a join query $Q_\sigma = Q \bowtie R_0 \bowtie \dots \bowtie R_b$.

THEOREM 3.4. *Let Q be a join query and \mathbf{D} a database that define the training dataset $Q(\mathbf{D})$, and f_θ a linear function with basis functions that induce a relational schema σ . Let Q_σ be the extension of Q with respect to σ . Then, the parameters of f_θ can be learned in time $O(|\mathbf{D}|^{f_{\text{htw}}(Q_\sigma)})$.*

The **two insights** discussed above complement each other, in particular Proposition 3.1 still holds in the presence of factorized joins. The commutativity with projection is especially useful in conjunction with factorization since it does not restrict our choice of possible d-trees for the factorized join depending on the input features used for learning. We may choose the best possible factorization of the join result and at learning time skip over irrelevant attributes (e.g., join attributes). Explicitly removing join attributes from the factorized join may in fact lead to larger representations (this contrasts with the flat case). For instance, if we would eliminate from the factorized join in Figure 1(c) all Z and S -values, then the remaining attributes H , T , and P would become dependent on each other (they were independent conditioned on values for Z and S). The only permissible d-trees would be paths, and the factorized join may be asymptotically as large as the flat join.

For simplicity of exposition, we assume the following (not required in practice). Each relation has one extra attribute I with value 1 to accommodate the intercept θ_0 . The d-trees for the join of the relations have the variable I as root. The factorized join has the I -value 1 as root.

4. F/FDB: COFACTOR COMPUTATION ON MATERIALIZED FACTORIZED JOINS

The cofactors in Equation (4) can be rewritten to mirror the factorization in the factorized join. In this section, we introduce **F/FDB**, an approach that relies on FDB [5] to compute the factorized join and that computes the rewritten cofactors in one pass over the factorized join.

4.1 Cofactor Computation By Factorization

We explain cofactor factorization by examples.

EXAMPLE 4.1. For the training dataset TD in Figure 1(a), there is one sum aggregate in Equation (2) per feature (i.e., attribute) in the dataset. For feature Z , we obtain:

$$\begin{aligned} S_Z = & (\theta_Z z_1 + \theta_H h_1 + \theta_S s_1 + \theta_P p_1 + \theta_T t_1) z_1 + \\ & (\theta_Z z_1 + \theta_H h_1 + \theta_S s_1 + \theta_P p_1 + \theta_T t_2) z_1 + \\ & (\theta_Z z_1 + \theta_H h_1 + \theta_S s_1 + \theta_P p_1 + \theta_T t_3) z_1 + \\ & \dots \text{(the above block repeated for } p_2) \\ & (\theta_Z z_1 + \theta_H h_1 + \theta_S s_2 + \theta_P p_3 + \theta_T t_4) z_1 + \\ & (\theta_Z z_1 + \theta_H h_1 + \theta_S s_2 + \theta_P p_3 + \theta_T t_5) z_1 + \\ & \dots \text{(all above repeated for } h_2 \text{ and } h_3) \\ & (\theta_Z z_2 + \theta_H h_4 + \theta_S s_2 + \theta_P p_4 + \theta_T t_4) z_2 + \\ & (\theta_Z z_2 + \theta_H h_4 + \theta_S s_2 + \theta_P p_4 + \theta_T t_5) z_2. \end{aligned}$$

We can reformulate the aggregate S_Z using the rewritings

$$\begin{aligned} \sum_{i=1}^n x & \rightarrow x \cdot n \text{ and } \sum_{i=1}^n x \cdot a_i \rightarrow x \cdot \sum_{i=1}^n a_i : \\ S_Z = & \theta_Z [z_1 (\underbrace{z_1 + \dots + z_1}_{|\sigma_{Z=z_1}(TD)|}) + z_2 (\underbrace{z_2 + \dots + z_2}_{|\sigma_{Z=z_2}(TD)|}) + \\ & \theta_H [z_1 (\underbrace{\sum_{i=1}^3 \underbrace{h_i + \dots + h_i}_{|\sigma_{Z=z_1, H=h_i}(TD)|}}_{|\sigma_{Z=z_1, H=h_i}(TD)|}) + z_2 (\underbrace{h_4 + \dots + h_4}_{|\sigma_{Z=z_2, H=h_4}(TD)|})] + \\ & \theta_S [z_1 (\underbrace{\sum_{i=1}^2 \underbrace{s_i + \dots + s_i}_{|\sigma_{Z=z_1, S=s_i}(TD)|}}_{|\sigma_{Z=z_1, S=s_i}(TD)|}) + z_2 (\underbrace{s_2 + \dots + s_2}_{|\sigma_{Z=z_2, S=s_2}(TD)|})] + \\ & \theta_P [z_1 (\underbrace{\sum_{i=1}^3 \underbrace{p_i + \dots + p_i}_{|\sigma_{Z=z_1, P=p_i}(TD)|}}_{|\sigma_{Z=z_1, P=p_i}(TD)|}) + z_2 (\underbrace{p_4 + \dots + p_4}_{|\sigma_{Z=z_2, P=p_4}(TD)|})] + \\ & \theta_T [z_1 (\underbrace{\sum_{i=1}^5 \underbrace{t_i + \dots + t_i}_{|\sigma_{Z=z_1, T=t_i}(TD)|}}_{|\sigma_{Z=z_1, T=t_i}(TD)|}) + z_2 (\underbrace{\sum_{i=4}^5 \underbrace{t_i + \dots + t_i}_{|\sigma_{Z=z_2, T=t_i}(TD)|})]. \end{aligned}$$

We then obtain the following cofactors in the sum S_Z :

$$\begin{aligned} Q[Z, Z] &= z_1 \cdot 24z_1 + z_2 \cdot 2z_2 \\ Q[H, Z] &= z_1 \cdot 8(h_1 + h_2 + h_3) + z_2 \cdot 2h_4. \\ Q[S, Z] &= z_1 \cdot 3(6s_1 + 2s_2) + z_2 \cdot 2s_2. \\ Q[P, Z] &= z_1 \cdot 3[3(p_1 + p_2) + 2p_3] + z_2 \cdot 2p_4. \\ Q[T, Z] &= z_1 \cdot 3[2(t_1 + t_2 + t_3) + t_4 + t_5] + z_2 \cdot (t_4 + t_5). \quad \square \end{aligned}$$

This arithmetic factorization is not arbitrary. It considers the arithmetic expressions grouped by the join Z -values, as done by the d-tree in Figure 1(b). Each cofactor in S_Z is expressed as a sum of terms with one term per each join Z -value z_1 and z_2 . The numerical values occurring in the cofactors represent *occurrence counts*, e.g., 24 in $L[Z] = 24z_1$ states that z_1 occurs in 24 tuples in the training dataset, while 8 in $L[H] = 8(h_1 + h_2 + h_3)$ states that each of h_1 , h_2 , and h_3 occurs in 8 tuples with z_1 . The expressions $L[Z]$ and $L[H]$ represent *sums* of Z -values and respectively H -values that occur in the same tuples with z_1 and that are *weighted* by their occurrence counts.

The above rewritings do not capture the full spectrum of possible computational savings: They are sufficient for cofactors of features θ_X in Sum_Y , where X and Y are from the same input relation. Moreover, they do not bring asymptotic savings. In case X and Y are from different input relations, then we can potentially save more computation.

EXAMPLE 4.2. Consider now a rewriting of the cofactor of parameter θ_P in sum S_T :

$$Q[P, T] = 3[(p_1 + p_2) \cdot (t_1 + t_2 + t_3)] + 3[p_3 \cdot (t_4 + t_5)] + p_4 \cdot (t_4 + t_5).$$

The three terms in the outermost sum correspond to different pairs of join values for Z and S , namely (z_1, s_1) , (z_1, s_2) ,


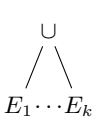
f-fdb (Factorization E)	
if (<i>visited</i>) return ; switch <i>E</i> :	
	$C_E = 1;$ foreach $j \in [k]$ do { f-fdb (E_j); $C_E = C_E \cdot C_{E_j};$ } $L_E[A] = a \cdot C_E;$ $Q_E[A, A] = a \cdot L_E[A];$ foreach $j \in [k]$ do { foreach $B \in \text{Schema}[E_j]$ do { $L_E[B] = L_{E_j}[B] \cdot C_E / C_{E_j};$ $Q_E[A, B] = a \cdot L_{E_j}[B] \cdot C_E / C_{E_j};$ } foreach $B, D \in \text{Schema}[E_j]$ s.t. $B < D$ do $Q_E[B, D] = Q_{E_j}[B, D] \cdot C_E / C_{E_j};$ foreach $j < l \in [k], B \in \text{Schema}[E_l], D \in \text{Schema}[E_j]$ do $Q_E[B, D] = L_{E_l}[B] \cdot L_{E_j}[D] \cdot C_E / (C_{E_l} \cdot C_{E_j});$ }
	$C_E = 0;$ foreach $j \in [k], B, D \in \text{Schema}[E_j]$ do { $L_E[B] = 0;$ $Q_E[B, D] = 0;$ } foreach $j \in [k]$ do { f-fdb (E_j); $C_E += C_{E_j};$ } foreach $j \in [k], B \in \text{Schema}[E_j]$ do { $L_E[B] += L_{E_j}[B];$ foreach $D \in \text{Schema}[E_j]$ do $Q_E[B, D] += Q_{E_j}[B, D];$ } $\}$
<i>visited</i> = <i>true</i> ;	

Figure 2: F/FDB: Algorithm for computing regression aggregates (constant C_E , linear L_E , quadratic Q_E) in one pass over a factorized join E .

and (z_2, s_2) . This rewriting thus follows the same join order as the d-tree in Figure 1(b). These terms read as follows: Each of the P -values p_1 and p_2 occurs in three tuples with each of the T -values t_1 to t_3 ; the P -value p_3 occurs in three tuples with each of the T -values t_4 and t_5 ; the P -value p_4 occurs in one tuple with each of the T -values t_4 and t_5 . \square

The rewritten expression for $Q[P, T]$ factors out sums, e.g., $p_1 + p_2$, using a rewriting more powerful than those for the sum S_Z given in Example 4.1 that can transform expressions to exponentially smaller equivalent ones:

$$\sum_{i=1}^r \sum_{j=1}^s (x_i \cdot y_j) \rightarrow \left(\sum_{i=1}^r x_i \right) \cdot \left(\sum_{j=1}^s y_j \right).$$

The above rewritings are already implemented by the factorized join from Figure 1(c). For instance, the sums of values in the cofactors mentioned in Examples 4.1 and 4.2 can be recovered via unions of their corresponding values in the factorization. Since Z -values are above the values for the other attributes, the former are in one-to-many relationships with the latter. This explains the rewritings in Example 4.1 for the cofactors in sum S_Z : Each of z_1 and z_2 are paired with the weighted sums of all H -values underneath, namely $8(h_1 + h_2 + h_3)$ and respectively $2h_4$. Similar pairings are with the weighted sums of values for each of the other attribute. Since P and T are on different branches in the d-tree, a Cartesian product of a union of P -values and a union of T -values becomes a product of the sums of the corresponding P -values and T -values.

Examples 4.1 and 4.2 show that cofactor computation requires three types of aggregates: Constant aggregates that are occurrence counts; linear aggregates that are weighted sums, i.e., sums over features or products of linear and constant aggregates; and quadratic aggregates that are cofactors, i.e., products of features and/or linear aggregates, or of quadratic and constant aggregates. Constant aggregates are real numbers and denoted by C . Linear aggregates are arrays of reals, one per feature A and denoted by $L[A]$. Quadratic aggregates are matrices of reals, one per each pair of features (A, B) and denoted by $Q[A, B]$.

DEFINITION 4.3. A *regression aggregate* is a tuple (C, L, Q) of constant, linear, and quadratic aggregates. We use indices E and Δ to refer to the regression aggregates (C_E, L_E, Q_E) for a factorization E and $(C_\Delta, L_\Delta, Q_\Delta)$ for a d-tree Δ .

4.2 Computing Regression Aggregates

The algorithm **f-fdb** in Figure 2 computes the regression aggregates, and in particular the cofactors, at each node in the input factorization E in one pass over E .

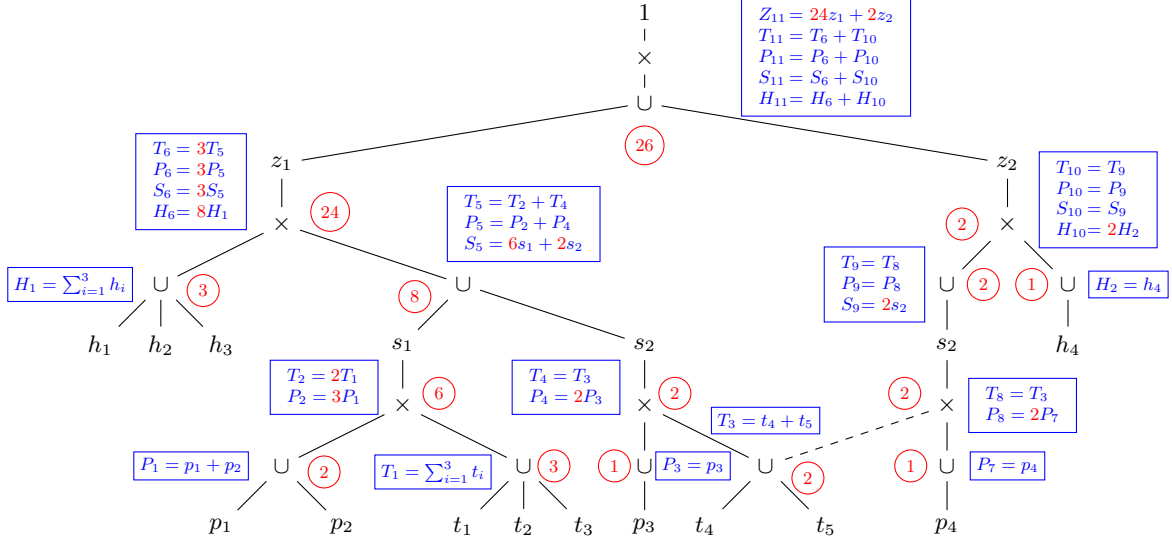
Let us denote by $\llbracket E \rrbracket$ the relation represented by the factorization E . The schema $\text{Schema}[E]$ of $\llbracket E \rrbracket$ is the set of features (or query variables) from the d-tree of E . The Cartesian product and union operators in the factorization translate to multiplication and summation for regression aggregates. Since they commute with union, we compute them for each child of a union and then add them entrywise.

The constant aggregate C_E is the number of tuples in $\llbracket E \rrbracket$. The constant aggregates are used to compute occurrence counts as follows. If E is the child of a product E_\times , its occurrence count in E_\times (i.e., the occurrence count of each tuple in $\llbracket E \rrbracket$) is the product of the constant aggregates of its siblings. This is correct, since each of the tuples represented by E is extended in the relation $\llbracket E_\times \rrbracket$ by each tuple represented by each of E 's siblings.

The linear aggregate $L_E[A]$ for a feature $A \in \text{Schema}[E]$ is the sum of A -values in E , each weighted by their occurrence counts given by constant aggregates.

The quadratic aggregate $Q_E[B, D]$ for features $B, D \in \text{Schema}[E]$ is the sum of quadratic aggregates $Q_{E_j}[B, D]$ for the child E_j of E if both B and D occur in E_j , or the product of linear aggregates for each of B and D weighted by their occurrence counts, if B and D are from different children of E . We use the symmetry of the cofactor matrix (cf. Proposition 3.1) to only compute the quadratic aggregates for B and D in case B occurs before D in the depth-first left-to-right preorder traversal of the d-tree (and factorization). The cofactor matrix is formed by the quadratic aggregates at the root of the factorization.

To support caching, we use a Boolean flag *visited* for each node in the factorization. Initially, this flag is false. After



	θ_I	θ_Z	θ_S	θ_P	θ_T	θ_H
Σ_I	26	$24z_1 + 2z_2$	$3(6s_1 + 2s_2) + 2s_2$	$9(p_1 + p_2) + 6p_3 + 2p_4$	$6(t_1 + t_2 + t_3) + 3(t_4 + t_5)$	$8(h_1 + h_2 + h_3) + 2h_4$
Σ_Z	Σ_I/θ_Z	$24z_1^2 + 2z_2^2$	$z_1S_6 + z_2S_{10}$	$z_1P_6 + z_2P_{10}$	$z_1T_6 + z_2T_{10}$	$z_1H_6 + z_2H_{10}$
Σ_S	Σ_I/θ_S	Σ_Z/θ_S	$3(6s_1^2 + 2s_2^2) + 2s_2^2$	$3(s_1P_2 + s_2P_4) + s_2P_8$	$3s_1T_2 + 3s_2T_4 + s_2T_8$	$S_5H_1 + S_9H_2$
Σ_P	Σ_I/θ_P	Σ_Z/θ_P	Σ_S/θ_P	$9(p_1^2 + p_2^2) + 6p_3^2 + 2p_4^2$	$3P_1T_1 + 3P_3T_3 + P_7T_3$	$P_5H_1 + P_9H_2$
Σ_T	Σ_I/θ_T	Σ_Z/θ_T	Σ_S/θ_T	Σ_P/θ_T	$6(t_1^2 + t_2^2 + t_3^2) + 3(t_4^2 + t_5^2)$	$T_5H_1 + T_9H_2$
Σ_H	Σ_I/θ_H	Σ_Z/θ_H	Σ_S/θ_H	Σ_P/θ_H	Σ_T/θ_H	$8(h_1^2 + h_2^2 + h_3^2) + 2h_4^2$

Figure 3: The factorized join in Figure 1(c) with extra root value 1 for intercept. Constant (encircled) and linear (boxed) aggregates are shown for product and union nodes. The cofactor matrix (bottom) is formed by the quadratic aggregates for the root node. We may first compute the matrix and then explore possible regression models by choosing different sets of features and label to predict.

the the node is visited, the flag is toggled so that we recognize visited nodes and skip them should we arrive again at them. The parent of an already visited node ν can just use the previously computed regression aggregates for ν .

EXAMPLE 4.4. Figure 3 shows the factorization in Figure 1(c) annotated with constant and linear aggregates as computed in one bottom-up pass. The cofactor matrix is computed in the same pass. The top left entry in the matrix is the cofactor of the intercept θ_I in the sum S_I and represents the number of tuples in the flat join result. Since the matrix is symmetric, we only compute the entries above and including the diagonal. \square

Our algorithm inherits the data complexity of computing the factorized join as it is linear in its size. At each node ν , the algorithm **f-fdb** recurses once for each child of ν and we need time linear in its schema to compute the constant and linear aggregates, and quadratic in its schema to compute the quadratic aggregates. The time complexity is thus linear in the size of the input factorized data and quadratic in the size of the schema (number of features). We require space to store the factorized data. For each node in the factorization, we also require space quadratic in the schema size to store the regression aggregates. In the next section, we show how to avoid to store the factorized data and also to reduce the number of regression aggregates to only having one per variable in the d-tree.

5. F: MIXING COFACTOR AND FACTORIZED JOIN COMPUTATION

This section introduces two improvements to **F/FDB**. (1) We avoid the materialization of the factorized join. (2) We reduce the number of regression aggregates from one per node in the factorized join to one per node in its d-tree. This new algorithm is called **F**. It is an in-memory monolithic engine optimized for regression aggregates over joins.

Figure 4.2 gives **F**'s core procedure. It takes as input the database relations R_1, \dots, R_d and a d-tree Δ for the given join query. It computes a tuple $(C_\Delta, L_\Delta, Q_\Delta)$ of regression aggregates for the factorized join over Δ . Instead of materializing the factorized join, we iterate over its paths of values that are mapped to paths of variables in Δ . Such a mapping is kept in `varMap`. This join approach is reminiscent of LeapFrog TrieJoin [41] with the difference that instead of using a total variable (or join) order, we use a partial variable order defined by a d-tree. More branches in a d-tree mean a higher factorization degree and performance improvement.

The relations are assumed sorted on their attributes following a depth-first pre-order traversal of Δ . Each call takes a range defined by start and end indices in each relation. Initially, these ranges span the entire relations. Let us assume the variable at the root of Δ is A . Once A is mapped to a value a in the intersection of possible A -values from the relations with attribute A , then these ranges are narrowed down to those tuples with value a for A . We may further narrow down these ranges using mappings for variables below A in Δ at higher recursion depths. Each A -value a in this


```

f (d-tree  $\Delta$ , varMap, ranges[(start1, end1), ..., (startd, endd)])

 $A = \text{var}(\Delta)$ ;  keyMap =  $\pi_{\text{key}(A)}(\text{varMap})$ ;  reset( $C_\Delta, L_\Delta, Q_\Delta$ );

if ( $\text{key}(A) \neq \text{anc}(A)$ ) {  ( $C_\Delta, L_\Delta, Q_\Delta$ ) = cache[keyMap];  if ( $C_\Delta \neq 0$ ) return; }

foreach  $a \in \bigcap_{i \in [d], A \in \text{Schema}[R_i]} \pi_A(R_i[\text{start}_i, \text{end}_i])$  do {
  foreach  $i \in [d]$  do find ranges  $R_i[\text{start}'_i, \text{end}'_i] \subseteq R_i[\text{start}_i, \text{end}_i]$  such that  $\pi_A(R_i[\text{start}'_i, \text{end}'_i]) = \{(A : a)\}$ ;
  switch( $\Delta$ ) :
    leaf node  $A : C_\Delta += 1$ ;   $L_\Delta[A] += a$ ;   $Q_\Delta[A, A] += a \cdot a$ ;
    inner node  $A(\Delta_1, \dots, \Delta_k)$  :
      foreach  $j \in [k]$  do f( $\Delta_j$ , varMap  $\times \{(A : a)\}$ , ranges[(start'1, end'1), ..., (start'd, end'd)]);
       $C = C_{\Delta_1} \dots C_{\Delta_k}$ ;
      if ( $C \neq 0$ ) {
         $C_\Delta += C$ ;
        foreach  $j \in [k]$  do {
          foreach  $B \in \text{Schema}[\Delta_j]$  do {   $L_\Delta[B] += L_{\Delta_j}[B] \cdot C / C_{\Delta_j}$ ;   $Q_\Delta[A, B] += a \cdot L_{\Delta_j}[B] \cdot C / C_{\Delta_j}$ ;  }
          foreach  $B, D \in \text{Schema}[\Delta_j]$  s.t.  $B < D$  do  $Q_\Delta[B, D] += Q_{\Delta_j}[B, D] \cdot C / C_{\Delta_j}$ ;
          foreach  $j < l \in [k], B \in \text{Schema}[\Delta_l], D \in \text{Schema}[\Delta_j]$  do  $Q_\Delta[B, D] += L_{\Delta_l}[B] \cdot L_{\Delta_j}[D] \cdot C / (C_{\Delta_l} \cdot C_{\Delta_j})$ ;
        }
      }
    }
  }
if ( $\text{key}(A) \neq \text{anc}(A)$ )  cache[keyMap] = ( $C_\Delta, L_\Delta, Q_\Delta$ );

```

Figure 4: F: Algorithm for computing regression aggregates (constant C_Δ , linear L_Δ , quadratic Q_Δ) for a given d-tree Δ of the join query and a database (R_1, \dots, R_d) . The parameters of the initial call are the d-tree Δ , an empty variable map, and the full range of tuples for each relation.

intersection is the root of a factorization fragment over Δ . Instead of using one regression aggregate for each A -value, we may use one running regression aggregate for all of them since it commutes with their union. We thus only need as many regression aggregates as variables in the d-tree.

The regression aggregate for A is computed using the A -values and the aggregates for its children if any. We first reset it, since it might have been used for different factorization fragments with the same nesting structure Δ . We next check whether we previously computed this aggregate for the same factorization fragment and cached it. Whereas for **F/FDB** caching is done by FDB, **F** needs to manage the cache itself. As explained in Section 2, the *key* of each variable tells us when caching may be useful: This happens when $\text{key}(A)$ is strictly contained in $\text{anc}(A)$, since this means that the factorization fragments over the d-tree rooted at A are repeated for every distinct combination of values for variables in $\text{anc}(A) \setminus \text{key}(A)$. In this case, we probe the cache using as key the mappings from varMap of the variables in $\text{key}(A)$. If we have already visited that factorization fragment, then we can just use its previously computed regression aggregate and avoid recomputation. If this is the first visit, we insert the aggregate in the cache after we compute it. If the aggregate is not already in cache, we compute it precisely as for **F/FDB**. The case when Δ is an inner node is a combination of the two cases in Figure 2: We have a union of A -values, each on top of a product of subexpressions. The case when Δ is a leaf node corresponds to the first case in Figure 2 without subexpressions E_j .

F has the same time complexity as **F/FDB** (including FDB's computation of the factorized join). The key operation needed for joins is the intersection of the arrays of ordered values defined by the relation ranges. This is done

efficiently using the unary leapfrog join [41]. Given d ordered arrays L_1, \dots, L_d , where $N_{\min} = \min\{|L_1|, \dots, |L_d|\}$ and $N_{\max} = \max\{|L_1|, \dots, |L_d|\}$ are the sizes of the smallest and respectively largest array, their intersection takes time $O(N_{\min} \log(N_{\max}/N_{\min}))$, which is the size of the smallest array if we ignore log factors. The time to compute the regression aggregates is the same as for **F/FDB**. The amount of extra memory necessary for computing the join is however limited: For each relation, there is one index range that is a pair of numbers, varMap has at most one mapping per variable at any one time, and the number of regression aggregates is bounded by the number of variables in the d-tree.

6. F/SQL: F IN SQL

F's cofactor computation can be expressed using one SQL query whose result is a table with one row that contains all cofactors. This SQL encoding of **F**, called **F/SQL**, has desirable properties. (1) It can be computed by any relational database system and is thus readily deployable in practice with a small implementation overhead. (2) It works for datasets that do not fit in memory. (3) Like **F**, it pushes aggregates past joins for efficiency.

We generate the **F/SQL** query in two steps: We first rewrite the query so that cyclic joins are isolated and the rewritten query becomes (α) -acyclic. To retain the complexity of **F**, these isolated cyclic joins have to be computed using a worst-case optimal algorithm [41]. We then traverse bottom up an optimal d-tree Δ for the rewritten query and at each node with variable g we create a query that joins on g and then computes the regression aggregates for g . This evaluation strategy thus intertwines the joins and the aggregates like in **F**, which is prerequisite for its good complexity. In contrast to **F**, **F/SQL** requires as input an *extended*

rewrite(d-tree Δ)
$QS = \emptyset; \quad A = \text{var}(\Delta);$ if $(\Delta = A(\Delta_1, \dots, \Delta_k)) \quad QS = \bigcup_{j \in [k]} \text{rewrite}(\Delta_j);$ $Q_A = \text{relations}(\text{key}(A) \cup \{A\});$ if $(\nexists Q \in QS \text{ s.t. } Q_A \subseteq Q) \quad \textbf{return } QS \cup \{Q_A\}$ else return QS

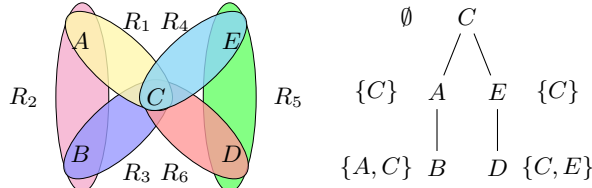
Figure 5: F/SQL: Query rewriting procedure.

d-tree where each input table R becomes a leaf under the lowest variable that corresponds to an attribute of R .

6.1 Rewriting Arbitrary Joins to Acyclic Joins

The rewriting procedure is given in Figure 5 and works on a d-tree Δ of the input join query Q_{in} . The set QS is a disjoint partitioning of the set of relations in Q_{in} into sets of relations or partitions. Each partition Q_A corresponds to a join query that is materialized to a relation with the same name Q_A . The set $\text{key}(A) \cup \{A\}$ consists of those variables on which the variables in Δ depend. D-trees are a different syntax for hypertree decompositions and the set $\text{key}(A) \cup \{A\}$ represents the bag at a node in such a decomposition [28]. By materializing the joins defined by such bags, we simplify the hypertree decomposition or d-tree of Q_{in} to that of an acyclic query Q_{out} equivalent to Q_{in} . In case Q_{in} is already acyclic, then each partition has one relation and hence Q_{out} is syntactically equal to Q_{in} .

EXAMPLE 6.1. Consider the following bowtie join query Q over relations R_1, \dots, R_6 together with a d-tree for it:



We state the keys next to each variable in the d-tree. We apply the rewriting algorithm. When we reach leaf B in the left branch, we create the join query Q_B over the relations $\{R_1, R_2, R_3\}$ and add it to QS . When we return from recursion to variable A , we create the query Q_A over the same relations, so we do not add it to QS . We proceed similarly in the right branch, create the join query Q_D over relations $\{R_4, R_5, R_6\}$, and add it to QS . The queries at E and C are not added to QS . Whereas the original query and the two subqueries Q_B and Q_D are cyclic, the rewritten query Q_{out} is the join of Q_B and Q_D on C and is acyclic. The triangle queries Q_B and Q_D cannot be computed worst-case optimally with traditional relational query plans [3], which calls for specialized engines to compute them [41].

The query in Figure 1(a) is acyclic. Using its d-tree from Figure 1(b), we obtain one identity query per relation. \square

6.2 SQL for Learning over Acyclic Joins

The SQL construction algorithm is recursive on the structure of the extended d-tree. As we proceed bottom up, we aggregate over the input attributes \mathcal{A} and create sets of constant \mathcal{C} , linear \mathcal{L} , and quadratic \mathcal{Q} aggregates.

At a leaf for a table R , we create a trivial query \mathbf{R} :

$\mathbf{R} = \text{SELECT } \mathcal{A}(R).*, \mathcal{C}(R).*, \mathcal{L}(R).*, \mathcal{Q}(R).* \text{ FROM } R;$

where \mathcal{A} is the schema of R , there is no linear and quadratic aggregate, and we add one constant aggregate with value 1.

At an inner node with variable g , where \mathbf{G}_i is the query created at the i -th child, we create the aggregate query \mathbf{G} over the natural join \mathbf{G}_{\bowtie} of queries $\mathbf{G}_1, \dots, \mathbf{G}_k$ on g .

$$\begin{aligned}
 \mathbf{G}_{\bowtie} &= \text{SELECT } \mathcal{Q}(\mathbf{G}_{\bowtie}).*, \mathcal{L}(\mathbf{G}_{\bowtie}).*, \mathcal{C}(\mathbf{G}_{\bowtie}).*, \mathcal{A}(\mathbf{G}_{\bowtie}).* \\
 &\quad \text{FROM } \mathbf{G}_1 \text{ NATURAL JOIN } \dots \mathbf{G}_k; \\
 \mathcal{Q}(\mathbf{G}_{\bowtie}) &= \{g \cdot \#_{-i} \mid q \in \mathcal{Q}(\mathbf{G}_i), i \in [k]\} \cup \{g \cdot 1 \cdot \# \} \cup \\
 &\quad \{l_i \cdot l_j \mid l_i \in \mathcal{L}(\mathbf{G}_i), l_j \in \mathcal{L}(\mathbf{G}_j), 1 \leq i < j \leq k\} \cup \\
 &\quad \{g \cdot l_i \cdot \#_{-i} \mid l_i \in \mathcal{L}(\mathbf{G}_i), i \in [k]\} \cup \{g^2 \cdot \# \} \\
 \mathcal{L}(\mathbf{G}_{\bowtie}) &= \{l_i \cdot \#_{-i} \mid l_i \in \mathcal{L}(\mathbf{G}_i), i \in [k]\} \cup \{g \cdot \# \} \\
 \mathcal{C}(\mathbf{G}_{\bowtie}) &= \{\#\} \quad \mathcal{A}(\mathbf{G}_{\bowtie}) = \bigcup_{i=1}^k \mathcal{A}(\mathbf{G}_i) - \{g\} \\
 \# &= \prod_{j=1}^k \{c \mid c \in \mathcal{C}(\mathbf{G}_j)\} \quad \#_{-i} = \# / (\prod_{c \in \mathcal{C}(\mathbf{G}_i)} c)
 \end{aligned}$$

Two constant aggregates are used in several regression aggregates: $\#$ is the number of tuples in the result of \mathbf{G}_{\bowtie} and $\#_{-i}$ is $\#$ where we ignore child i from the join. The regression aggregates are computed as for **F**. We next group by the remaining attributes $\mathcal{A}(\mathbf{G}_{\bowtie})$ and sum over each aggregate:

$$\begin{aligned}
 \mathbf{G} &= \text{SELECT } \mathcal{A}(\mathbf{G}).*, \mathcal{C}(\mathbf{G}).*, \mathcal{L}(\mathbf{G}).*, \mathcal{Q}(\mathbf{G}).* \\
 &\quad \text{FROM } \mathbf{G}_{\bowtie} \text{ GROUP BY } \mathcal{A}(\mathbf{G}).*; \\
 \mathcal{Q}(\mathbf{G}) &= \{\sum q \mid q \in \mathcal{Q}(\mathbf{G}_{\bowtie})\} \quad \mathcal{L}(\mathbf{G}) = \{\sum l \mid l \in \mathcal{L}(\mathbf{G}_{\bowtie})\} \\
 \mathcal{C}(\mathbf{G}) &= \{\sum c \mid c \in \mathcal{C}(\mathbf{G}_{\bowtie})\} \quad \mathcal{A}(\mathbf{G}) = \mathcal{A}(\mathbf{G}_{\bowtie})
 \end{aligned}$$

EXAMPLE 6.2. Consider the d-tree in Figure 1(b). We create the queries Q_P at node P and then Q_T at node T :

```

SELECT Z, S, sum(House_c) as P_c, sum(P) as P_11,
       sum(P*P*House_c) as P_q1
FROM (SELECT Z, S, P, 1 as House_c FROM House)
GROUP BY Z, S;
SELECT S, sum(TaxBand_c) as T_c, sum(T) as T_11,
       sum(T*T*TaxBand_c) as T_q1
FROM (SELECT S, T, 1 as TaxBand_c FROM TaxBand)
GROUP BY S;

```

At node S , we create the query Q_S using Q_P and Q_T :

```

SELECT Z, sum(S_c) as S_c, sum(S_11) as S_11,
       sum(S_12) as S_12, sum(S_13) as S_13,
       sum(S_q1) as S_q1, sum(S_q2) as S_q2,
       sum(S_q3) as S_q3, sum(S_q4) as S_q4,
       sum(S_q5) as S_q5, sum(S_q6) as S_q6
FROM (SELECT Z, P_c*T_c as S_c, P_11*T_c as S_11,
       T_11*P_c as S_12, S*S_c as S_13,
       P_q1*T_c as S_q1, T_q1*P_c as S_q2,
       P_11*T_11 as S_q3, S*P_11*T_c as S_q4,
       S*T_11*P_c as S_q5, S*S_13 as S_q6
FROM Q_P NATURAL JOIN Q_T)
GROUP BY Z;

```

The nodes H and Z are treated similarly. The constructed query exploits caching: Indeed, for each distinct S -value s , Q_T computes once the aggregates over T in TaxBand. Without caching, we would repeat the computation of Q_T for every occurrence of the S -value s in House. \square

F/SQL has the time complexity of **F** under the assumption that cyclic joins are computed using a worst-case optimal join algorithm like LFTJ. For (input and rewritten) acyclic queries, we may use standard query plans.

7. EXPERIMENTS

We report on the performance of an end-to-end solution for learning regression models over joins, which includes: (a) Constructing the training dataset via joins; (b) importing the dataset in the learning module; and (c) learning the parameters of regression functions. We show experimentally that the intermediate join step is unnecessarily expensive. It entails a high degree of redundancy in both computation and data representation, yet this is not required for the end-to-end solution, whose result is a list of real-valued parameters. By factorizing the join we reduce data redundancy while improving performance for both the join and the learning steps. A tight integration of learning and join processing also eliminates the need for the data import step.

We benchmark **F** and its flavors against three open-source systems: M (MADlib [17]), P (Python StatsModels [40]), and R [33]. We show that for the used datasets and learning tasks, **F** outperforms these systems by up to three orders of magnitude, while maintaining their accuracy; we verified that for all systems the learned parameters coincide with high precision. This performance boost is due to three orthogonal optimizations: Factorization and caching of data and computation; decoupling parameter convergence from cofactor computation; and shared cofactor computation.

Regression Learners. We implemented **F** and its flavors in C++. They all use AdaGrad to adapt the learning rate (α) based on history in the convergence component [13]. **F/FDB** uses FDB [5] to compute factorized joins via an in-memory multiway sort-merge join. **F/SQL** is **F**'s encoding in SQL. For R we used *lm* (linear model) based on QR-decomposition [15]. For P we used *ols* (ordinary least squares) based on the Moore-Penrose pseudoinverse [29]. For M we used *ols* to compute the closed-form solution and *glm* based on Newton optimization for generalized linear models. M (*glm*), P, and R use PostgreSQL 9.4.4 for join computation (done by query plans with sort-merge joins for the Housing dataset and with in-memory hash joins for the other datasets). We tuned PostgreSQL for in-memory processing by setting its working memory to 14GB and shared buffers to 128MB and by turning off parameters that affect performance (*fsync*, *synchronous commit*, *full page writes*, *bgwriter LRU maxpages*). We verified that it runs in memory by monitoring IO. From all the systems, M (*ols*), **F**, and **F/SQL** do not require the materialization of the join query. P and R need to export data from PostgreSQL and load it into their memory space, which requires one pass over the flat join and construction of its internal representation.

Experimental Setup. All experiments were performed on an Intel(R) Core(TM) i7-4770 3.40GHz/64bit/32GB with Linux 3.13.0/g++4.8.4 (no compiler optimization flags were used, *ulimit* was set to unlimited). We report wall-clock times by running each system once and then reporting the average of four subsequent runs with warm cache. For P, R, M (*glm*), and **F/FDB** we break down the times for: Computing the join in memory, importing the data, and learning the parameters; we do not report the times to load the database into memory as they can differ substantially between PostgreSQL and FDB and are orthogonal to this work. All tables are given sorted by their join attributes.

Datasets and Learning Tasks. We experimented with a real-world dataset, which is used by a large US retailer for forecasting user demands and sales, with two public datasets LastFM [10] and MovieLens [16], and a synthetic dataset

modeling the textbook example on house price market [25]. We next briefly introduce the learning tasks for each dataset; a detailed description is given in Appendix.

US retailer: We considered three linear regression tasks that predict the amount of inventory units based on all other features: L considers all features; N_1 and N_2 also consider two interactions of features from the same tables (no factorization restructuring necessary), and respectively from different tables (factorization restructuring necessary).

LastFM: We consider two training datasets L_1 and L_2 to relate how often friends listen to the same artists.

MovieLens: The regression task is to predict the rating given by a user to a movie.

Housing: The regression task is to predict the housing price based on all the features in the dataset.

1. Flat vs. Factorized Joins. The compression ratio is a direct indicator of how well **F** fares against approaches that rely on flat representation of the training dataset. As shown in Table 1, the compression factor brought by factorizing the joins varies from 4.43 for MovieLens to 26.61 for US retailer and over 100 for LastFM. Caching can improve the compression factor even further: 3x for MovieLens and 8x for LastFM. The effect of caching for US retailer is minimal.

While the speedup of FDB over PostgreSQL is significant, it is less than the data compression ratio possibly due to the less optimized data structures to encode factorizations in FDB. As shown in Figure 6, for scale 10 the factorized join is computed in 1.25 seconds vs. 64 seconds for the flat join. All join queries used in the experiments are acyclic, which means that our **F** flavors can compute them in linear time (modulo log factors).

Table 1 reports the join time for the real-world datasets. For the US retailer, the flat join is too large to be handled by P and R. In practice, such large datasets are partitioned and learning happens independently for each partition. This entails a loss of accuracy as we miss correlations across features. We partitioned the largest table Inventory (84M tuples) into four (ten) disjoint partitions for R (respectively, P) of roughly equal sizes by hashing on the join values for location. By joining each partition with the other tables we obtain a partitioning of the join result. Table 1 reports the overall (starred) time to compute the join for all partitions.

2. Importing Datasets. P and R are the only systems that need one pass over the join result to load it into their internal data structures. This is typical for the existing solutions based on software enterprise stacks consisting of dozens of specialized systems (e.g., for OLAP, OLTP, and BI), where non-trivial integration effort is usually spent at the interface between these systems. Table 1 and Figure 6 report the times for importing the training datasets. For Housing, P and R failed to import the data starting with scale 10 and respectively 11. In contrast, **F** can even finish for scale 100.

3. Learning. Table 1 shows that the speedup factor of learning with **F/FDB** versus competitors is larger than the compression factor, which is partially due to sharing computation across cofactors and, when compared against the gradient descent approximation methods like M(*glm*), also due to our decoupling of cofactor computation from convergence. For R and P on US retailer, we partitioned the dataset as explained in Experiment 2. Table 1 reports the sum of learning times for all partitions. However, the learned parameters for each partition are arbitrarily far from true ones (although

		US retailer L	US retailer N_1	US retailer N_2	LastFM L_1	LastFM L_2	MovieLens L
# parameters		31	33	33	6	10	27
Join Size	Factorized	97,134,867	97,134,867	97,134,867	2,577,556	2,379,264	6,092,286
	cached	97,134,675	97,134,675	97,134,675	376,402	315,818	2,115,610
	Flat	2,585,046,352	2,585,046,352	2,585,046,352	369,986,292	590,793,800	27,005,643
	Compression	26.61×	26.61×	26.61×	143.54×	248.31×	4.43×
	cached	26.61×	26.61×	26.61×	982.86×	1870.68×	12.76×
Join Time	Fact. (FDB)	36.03	36.03	36.03	4.79	9.94	12.28
	Flat (PSQL)	249.41	249.41	249.41	54.25	61.33	1.30
Import Time	R	1189.12*	1189.12*	1189.12*	155.91	276.77	10.86
	P	1164.40*	1164.40*	1164.40*	179.16	328.97	11.33
Learn Time	F/FDB	9.69	9.82	9.79	0.53	0.89	3.87
	M (glm)	2671.88	2937.49	2910.23	572.88	746.50	31.91
	R	810.66*	873.14*	884.47*	268.04	466.52	6.96
	P	1199.50*	1277.10*	1275.08*	35.74	148.84	10.92
	F	16.29	16.56	16.50	0.11	0.25	2.12
Total Time	F/FDB	45.72	45.85	45.82	5.32	10.83	16.15
	F/SQL	108.81	109.02	109.07	0.58	2.00	14.26
	M (ols)	680.60	737.02	738.08	152.37	196.60	7.08
	M (glm)	2921.29	3186.90	3159.64	627.13	807.83	33.21
	R	2249.19*	2311.67*	2323.00*	478.20	804.62	19.12
	P	2613.31*	2690.91*	2688.89*	269.15	539.14	23.55
Speedup	F vs. M (ols)	41.78×	44.51×	44.73×	1385.18×	786.40×	3.34×
	F vs. M (glm)	179.33×	192.45×	191.49×	5701.18×	3231.32×	15.67×
	F vs. R	138.07×	139.59×	140.79×	4347.27×	3218.48×	9.02×
	F vs. P	160.42×	162.49×	162.96×	2446.82×	2156.56×	11.11×

Table 1: Performance comparison for learning over joins (size in number of values, time in seconds). **F**, **F/SQL**, and **M(ols)** intertwine the join and learning computation and we only show their total end-to-end times. **P** and **R** crashed for US retailer due to memory limitation, the starred numbers are for running times on disjoint partitions of the location values for the join (four for **R** and ten for **P**) and adding up the times.

not supported by **P** and **R**, they could have been made correct if the output parameters for a partition would serve as the initial parameter values for the next partition). Nevertheless, even under these simplifying assumptions for **P** and **R**, **F** learns the correct parameters at least two orders of magnitude faster than them.

The tasks N_1 and N_2 in Table 1 for US retailer consider feature interactions. The time to perform the regression task slightly increases, which is due to the fact that additional parameters are learned.

4. End-to-End Solution. Table 1 compares the performance of our end-to-end flavors of **F** and of our competitors, where we summed up their time components for the join, import, and learning. **F** performs the best for all datasets, which can be directly linked to the intermixing of the factorized join and cofactor computation. It is from 3x to 50x faster than **F/FDB** and from 5x to 8x faster than **F/FSQL**. For LastFM, **F** exceeds three-orders of magnitude improvement over its competitors. For US retailer, the performance speedup is around two orders of magnitude in comparison to **R**, **P**, and **M(glm)**, and 40 times in comparison to the closest competitor **M(ols)**. The performance gains for MovieLens are more limited, which is due to the comparatively smaller size of this dataset.

Figure 6 reports the performance of end-to-end solutions for Housing against **M(ols)**, and **R**, which are the best competitors for this dataset. We stress-tested **F** for scale factors beyond 20 for Housing: The total time without data loading is 4.1 seconds for scale 50 (compression ratio 14K) and 5.9 seconds for scale 100 (compression ratio 69K).

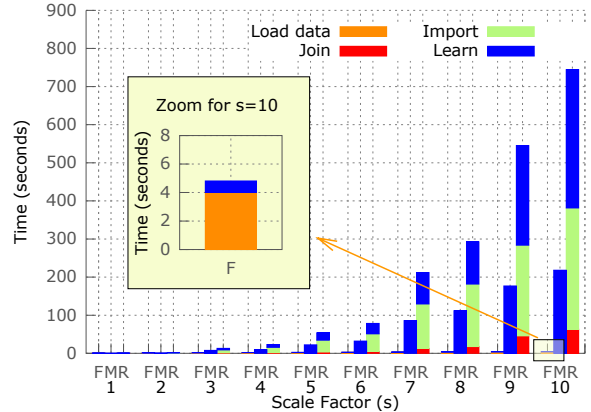


Figure 6: Total time for the end-to-end solution (loading data, join, import, learn) for **F** and the best competitors **M(ols)** and **R** on the Housing dataset. **R** runs out of memory starting with scale 11.

5. More features. We considered the Housing dataset with scale 15 and an increasing number of attributes per relation: 11, 18, 50, 100, and 168. This results in training datasets with around 60, 100, 300, 600 and 1000 features. Figure 7 shows the performance of **F** and **M(ols)** on these datasets (**P** and **R** already fail to load the original dataset with 27 features). **M(ols)** times out (i.e., it takes over four

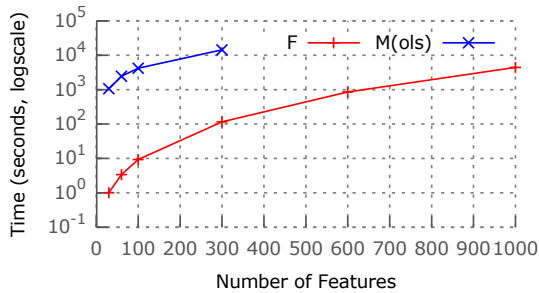


Figure 7: Total time for the end-to-end solution for \mathbf{F} and $\mathbf{M(ols)}$ on the Housing dataset with scale factor scale 15 and increasingly many features. $\mathbf{M(ols)}$ times out after 300 features.

hours) for the dataset with 300 features. \mathbf{F} ’s performance is consistently two orders of magnitude better than of $\mathbf{M(ols)}$.

6. Model selection. In the previous experiments, we learned over all features of the training dataset. We further considered settings with fewer features, as used for model selection. The experiments validated that \mathbf{F} only computes the parameter cofactors once and that the convergence time is consistently the smallest amongst all components of \mathbf{F} . This contrasts with \mathbf{M} , \mathbf{P} , and \mathbf{R} that need to independently learn over the entire dataset for each set of features.

8. RELATED WORK

Our contribution lies at the intersection of *databases* and *machine learning*, as we look at regression learning, which is a fundamental machine learning problem, through database glasses. The crossover between these two communities gained increasing interest during the past years, as also acknowledged in a SIGMOD 2015 panel [34].

Machine learning in databases. Our work follows a very recent line of research on marrying databases and machine learning [17, 14, 11, 36, 7, 20, 1, 24, 18, 9, 38, 34, 31, 32].

While \mathbf{F} ’s learning approach is novel in this landscape, two of these works are closest in spirit to it since they investigate the impact of data factorization for the purpose of boosting learning tasks. Rendle [36] considers a limited form of factorization of the design matrix for regression. His approach performs ad-hoc discovery of repeating patterns on the flat join to save up time in the subsequent regression task, though with two important limitations. This discovery does not seek to capture join dependencies, i.e., to recover the knowledge that the data has been produced via joins, since this is NP-hard. It also does not save time in join computation, but it instead needs additional time to perform the discovery on the flat join. Our approach is different since (i) we avoid the computation of the flat join as it is too expensive and entails redundancy; (ii) we exploit the join structure from the query to identify the repeating patterns due to join dependencies. Kumar et al. [20] propose a framework for learning generalized linear models over key-foreign key joins in a distributed environment and consider, amongst other techniques, factorized computation over the non-materialized join. \mathbf{F} works for arbitrary join queries and factorizations with caching, linear regression with feature interactions, and has theoretical performance guarantees. We show that \mathbf{F} ’s cofactor computation commutes with union, and can thus be trivially distributed. Prior work on factor-

ized databases discusses efficient support for joins [28] and simple aggregates [4]. \mathbf{F} shows that factorization can benefit more complex user-defined aggregate functions, such as the cofactor matrix used in linear regression.

Most efforts in the database community are on designing systems to support large-scale scalable machine learning on top of possibly distributed architectures, with a goal on providing a unified architecture for machine learning and databases [14], e.g., MLLib [1] and DeepDist [24] on Spark [42], distributed gradient descent on GLADE [32], MADlib [17] on PostgreSQL, SystemML [18, 7], system benchmarking [9] and sample generator for cross-validate learning [38]. Our approach is more specialized as it proposes an efficient batch gradient descent variant for linear regression over joins. It achieves efficiency by factorizing data and computation, which enables more data to be kept in the memory of one machine and to be processed fast without the need for distribution. As pointed out recently [22], when benchmarked against one-machine systems, distributed systems can have a non-trivial upfront cost that can be offset by more expensive hardware or large problem settings.

The tight integration of the FDB query engine for factorized databases [5] with our regression learner \mathbf{F} has been inspired by LogicBlox [2], which provides a unified runtime for the enterprise technology stack, and MADlib [17].

Gradient descent optimization. The gradient descent family of optimization algorithms is fundamental to machine learning [6] and very popular, cf. a ICML 2015 tutorial [37]. One of the applications of gradient descent is regression, which is the focus of this paper. A popular variant is the stochastic gradient descent [8] with several recent improvements such as faster convergence rate via adaptive learning rate [13] (which is also used by our system \mathbf{F}) and parallel or distributed versions [43, 30, 35, 12, 27]. Some of these optimizations have made their way in systems such as DeepDive [39, 21] and DeepDist [24, 12]. Our contribution is orthogonal since it focuses on avoiding data and computation redundancy when learning over joins.

9. CONCLUSIONS AND FUTURE WORK

This paper puts forward \mathbf{F} , a fast learner of least squares regression models over factorized joins that enjoys both theoretical and practical properties. The principles behind \mathbf{F} are applicable beyond least squares regression models, as long as the derivatives of the objective function are expressible in a semiring with multiplication and summation operations (which is the case for gradient descent and Newton approximation methods), much in the spirit of the FAQ framework [19]. This is necessary since factorization relies on commutativity and distributivity laws. For this reason, least squares models are supported while logistic regression, which uses exponential functions, is not. A non-exhaustive list of analytics to benefit from our approach includes factorization machines, boosted trees, and k -nearest neighbors.

Acknowledgements

This work benefitted from interactions with Molham Aref (on $\mathbf{F/SQL}$), Tim Furche (on experiments), Arun Kumar (on related work), Ron Menich (on problem setting and datasets), the MADlib team, and the anonymous reviewers. It was supported by a Google Research Faculty award, the ERC grant 682588, and the EPSRC platform grant DBOnto.

10. REFERENCES

- [1] Apache. MLlib: Machine learning in Spark, <https://spark.apache.org/mllib>, 2015.
- [2] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the LogicBlox system. In *SIGMOD*, pages 1371–1382, 2015.
- [3] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. In *FOCS*, pages 739–748, 2008.
- [4] N. Bakibayev, T. Kociský, D. Olteanu, and J. Závodný. Aggregation and ordering in factorised databases. *PVLDB*, 6(14):1990–2001, 2013.
- [5] N. Bakibayev, D. Olteanu, and J. Závodný. FDB: A query engine for factorised relational databases. *PVLDB*, 5(11):1232–1243, 2012.
- [6] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*, 2006.
- [7] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. Burdick, and S. Vaithyanathan. Hybrid parallelization strategies for large-scale machine learning in SystemML. *PVLDB*, 7(7):553–564, 2014.
- [8] L. Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade (2nd ed)*, pages 421–436, 2012.
- [9] Z. Cai, Z. J. Gao, S. Luo, L. L. Perez, Z. Vagena, and C. M. Jermaine. A comparison of platforms for implementing and running very large scale machine learning algorithms. In *SIGMOD*, pages 1371–1382, 2014.
- [10] I. Cantador, P. Brusilovsky, and T. Kuflik. 2nd workshop on information heterogeneity and fusion in recommender systems. In *RecSys*, pages 387–388, 2011, <http://grouplens.org/datasets/hetrec-2011>.
- [11] T. Condie, P. Mineiro, N. Polyzotis, and M. Weimer. Machine learning for big data. In *SIGMOD*, pages 939–942, 2013.
- [12] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, pages 1232–1240, 2012.
- [13] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *JMLR*, 12:2121–2159, 2011.
- [14] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a unified architecture for in-rdbms analytics. In *SIGMOD*, pages 325–336, 2012.
- [15] J. G. F. Francis. The QR transformation: A unitary analogue to the LR transformation—Part 1. *The Computer Journal*, 4(3):265–271, 1961.
- [16] GroupLens Research. MovieLens, <http://grouplens.org/datasets/movielens>, 2003.
- [17] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib analytics library or MAD skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.
- [18] B. Huang, M. Boehm, Y. Tian, B. Reinwald, S. Tatikonda, and F. R. Reiss. Resource elasticity for large-scale machine learning. In *SIGMOD*, pages 137–152, 2015.
- [19] M. A. Khamis, H. Q. Ngo, and A. Rudra. FAQ: Questions Asked Frequently, CoRR:1504.04044, 2015.
- [20] A. Kumar, J. F. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *SIGMOD*, pages 1969–1984, 2015.
- [21] J. Liu, S. Wright, C. Ré, V. Bittorf, and S. Sridhar. An asynchronous parallel stochastic coordinate descent algorithm. In *ICML*, pages 469–477, 2014.
- [22] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what COST? In *HotOS*, 2015.
- [23] R. Menich and N. Vasiloglou. The future of LogicBlox machine learning. LogicBlox User Days, 2013.
- [24] D. Neumann. Lightning-fast deep learning on Spark via parallel stochastic gradient updates, www.deepdist.com, 2015.
- [25] A. Ng. *CS229 Lecture Notes*. Stanford & Coursera, <http://cs229.stanford.edu/>, 2014.
- [26] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. In *PODS*, pages 37–48, 2012.
- [27] F. Niu, B. Recht, C. Ré, and S. J. Wright. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.
- [28] D. Olteanu and J. Závodný. Size bounds for factorised representations of query results. *TODS*, 40(1):2, 2015.
- [29] R. Penrose. A generalized inverse for matrices. *Math. Proc. Cambridge Phil. Soc.*, 51(03):406–413, 1955.
- [30] F. Petroni and L. Querzoni. GASGD: stochastic gradient descent for distributed asynchronous matrix completion via graph partitioning. In *RecSys*, pages 241–248, 2014.
- [31] C. Qin and F. Rusu. Scalable i/o-bound parallel incremental gradient descent for big data analytics in glade. In *DanaC*, pages 16–20, 2013.
- [32] C. Qin and F. Rusu. Speculative approximations for terascale distributed gradient descent optimization. In *DanaC*, pages 1:1–1:10, 2015.
- [33] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, www.r-project.org, 2013.
- [34] C. Ré, D. Agrawal, M. Balazinska, M. I. Cafarella, M. I. Jordan, T. Kraska, and R. Ramakrishnan. Machine learning and databases: The sound of things to come or a cacophony of hype? In *SIGMOD*, pages 283–284, 2015.
- [35] B. Recht and C. Ré. Parallel stochastic gradient algorithms for large-scale matrix completion. *Math. Program. Comput.*, 5(2):201–226, 2013.
- [36] S. Rendle. Scaling factorization machines to relational data. *PVLDB*, 6(5):337–348, 2013.
- [37] P. Richtárik and M. Schmidt. Modern convex optimization methods for large-scale empirical risk minimization. In *ICML*, 2015. Invited Tutorial.
- [38] S. Schelter, J. Soto, V. Markl, D. Burdick, B. Reinwald, and A. V. Evfimievski. Efficient sample generation for scalable meta learning. In *ICDE*, pages 1191–1202, 2015.
- [39] J. Shin, S. Wu, F. Wang, C. D. Sa, C. Zhang, and C. Ré. Incremental knowledge base construction using DeepDive. *PVLDB*, 8(11):1310–1321, 2015.

- [40] The StatsModels development team. StatsModels: Statistics in Python, <http://statsmodels.sourceforge.net>, 2012.
- [41] T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *ICDT*, pages 96–106, 2014.
- [42] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.
- [43] M. Zinkevich, M. Weimer, A. J. Smola, and L. Li. Parallelized stochastic gradient descent. In *NIPS*, pages 2595–2603, 2010.

APPENDIX

A. PROOFS

Proof of Proposition 3.1. 1. (*Symmetry*). Assume that $Q(\mathbf{D})$ has m tuples. We have:

$$S_k = \sum_{i=1}^m (\theta_0 x_0^{(i)} + \dots + \theta_j x_j^{(i)} + \dots + \theta_n x_n^{(i)}) x_k^{(i)},$$

$$S_j = \sum_{i=1}^m (\theta_0 x_0^{(i)} + \dots + \theta_k x_k^{(i)} + \dots + \theta_n x_n^{(i)}) x_j^{(i)}.$$

This implies that:

$$\text{Cofactor}[A_j, A_k] = \sum_{i=1}^m x_j^{(i)} x_k^{(i)} = \sum_{i=1}^m x_k^{(i)} x_j^{(i)} = \text{Cofactor}[A_k, A_j].$$

2. (*Commutativity with union*). For $1 \leq l \leq p$, assume that the training dataset $Q(\mathbf{D}_l)$ has m_l tuples, that we denote

$$Q(\mathbf{D}_l) = \{(x_{l,0}^{(1)}, \dots, x_{l,n}^{(1)}), \dots, (x_{l,0}^{(m_l)}, \dots, x_{l,n}^{(m_l)})\}.$$

Then, $Q(\mathbf{D})$ has $\sum_{l=1}^p m_l$ tuples. Take $0 \leq k, j \leq n$. It holds that:

$$\begin{aligned} \sum_{l=1}^p \text{Cofactor}_l[A_k, A_j] &= \sum_{l=1}^p \left(\sum_{i_l=1}^{m_l} x_{l,k}^{(i_l)} x_{l,j}^{(i_l)} \right) \\ &= \sum_{i=1}^{\sum_{l=1}^p m_l} x_k^{(i)} x_j^{(i)} = \text{Cofactor}[A_k, A_j]. \end{aligned}$$

3. (*Commutativity with projection*). Assume that $Q(\mathbf{D})$ has m tuples. Since we are under bag semantics, $\pi_L(Q(\mathbf{D}))$ also has m tuples. We assume that L has n_L features that we denote $x_{L,0}, \dots, x_{L,n_L}$. Take $1 \leq k, j \leq n_L$. Then, $\text{Cofactor}_L[A_k, A_j] = \sum_{i=1}^m x_j^{(i)} x_k^{(i)}$, which moreover, is equal to $\text{Cofactor}[A_k, A_j]$.

Proof of Theorem 3.4. Take the basis functions ϕ_0, \dots, ϕ_b over the sets of features S_0, \dots, S_b , the induced relational schema $\sigma = (R_0(S_0), \dots, R_b(S_b))$, and $Q_\sigma = Q \bowtie R_0 \bowtie \dots \bowtie R_b$ as the extension of Q w.r.t. σ .

First, we show that the extension \mathbf{D}_σ of \mathbf{D} with relations over the induced relational schema σ leads to a factorization of the join $Q_\sigma(\mathbf{D}_\sigma)$ of size $O(|\mathbf{D}|^{\text{fhtw}(Q_\sigma)})$. The extension \mathbf{D}_σ has a relation instance R_i (for $1 \leq i \leq b$) for each schema $R_i(S_i)$ that is the projection of $Q(\mathbf{D})$ on S_i i.e., $\pi_{S_i}(Q(\mathbf{D}))$. It then holds that $Q(\mathbf{D}) = Q(\mathbf{D}_\sigma) = Q_\sigma(\mathbf{D}_\sigma)$. By Proposition 2.2, there exists a factorization of \mathbf{D}_σ of size $O(|\mathbf{D}_\sigma|^{\text{fhtw}(Q_\sigma)})$. Under data complexity, the schema σ has constant size and thus $O(|\mathbf{D}_\sigma|^{\text{fhtw}(Q_\sigma)}) = O(|\mathbf{D}|^{\text{fhtw}(Q_\sigma)})$.

Let us denote by E the factorization of $Q_\sigma(\mathbf{D}_\sigma)$ and let Δ be a d-tree under which we obtain the size of $O(|\mathbf{D}|^{\text{fhtw}(Q_\sigma)})$.

We next show that E can be augmented with values corresponding to the interaction terms ϕ_k while keeping the same asymptotic bound on its size. The query variables of any relation atom in the query, and in particular of relations over schemas $R_0(S_0), \dots, R_b(S_b)$, are along the same root-to-leaf path in Δ . This means that E materializes all possible combinations of values for variables (corresponding to attributes) in each schema S_k . We can thus compute the result r_k of the basis function ϕ_k on the values for variables in S_k in one pass over E . For each tuple of values for variables in S_k , we add a product with the value r_k immediately under the lowest value in the tuple in E . We also add a variable ϕ_k in Δ immediately under the lowest variable in S_k . These additions do not modify the asymptotic size bounds of E since we add one value for each S_k -tuple of values in E . Let us denote by E_σ the factorization E extended with values for interaction terms. The factorization E_σ has size $O(|\mathbf{D}|^{\text{fhtw}(Q_\sigma)})$ and values for each basis function ϕ_k . Learning over E_σ has thus reduced to learning with identity basis functions and, by Proposition 3.2, this can be done in one pass over E_σ . The claim follows.

B. DESCRIPTION OF USED DATASETS

• **Housing** is a synthetic dataset emulating the textbook example for the house price market [25]. It consists of six tables: **House** (postcode, price, number of bedrooms/bathrooms/garages/parking lots, living room/kitchen area, etc.), **Shop** (postcode, opening hours, price range, brand, e.g., Costco, Tesco, Sainsbury’s), **Institution** (postcode, type of educational institution, e.g., university or school, and number of students), **Restaurant** (postcode, opening hours, and price range), **Demographics** (postcode, average salary, rate of unemployment, criminality, and number of hospitals), and **Transport** (postcode, the number of bus lines, train stations, and distance to the city center for the postcode). The training dataset is the natural join of all relations (on postcode) and has 27 features. There are 25K postcodes, which appear in all relations. The scale factor s determines the number of generated distinct tuples per postcode in each relation: We generate s tuples in **House** and **Shop**, $\log_2 s$ tuples in **Institution**, $s/2$ in **Restaurant**, and one in each of **Demographics** and **Transport**. The numerical values for each attribute are randomly generated. The domains are intervals simulating the real-world semantics of attributes e.g., large intervals for attributes such as **price**, smaller intervals for attributes such as **nbtrainstations** and Boolean values for attributes such as **house**, **flat**, or **bungalow** indicating the type of housing.

We considered one linear regression tasks that predicts the price of a house based on all other features. Since all relations have a common attribute (**postcode**), the above query is acyclic and in particular hierarchical. An optimal d-tree that we would have each root-to-leaf path consisting of query variables for one relation. Caching in **F** is not useful here. The time complexity for **F** is linear (modulo log factors). We present a snapshot of it in Figure 8(a).

• **US retailer**. This dataset consists of three relations: **Inventory** (storing information about the inventory units for products in a location, at a given date), **Census** (storing demographics information per zipcode such as population, median age, repartition per ethnicities, house units and how

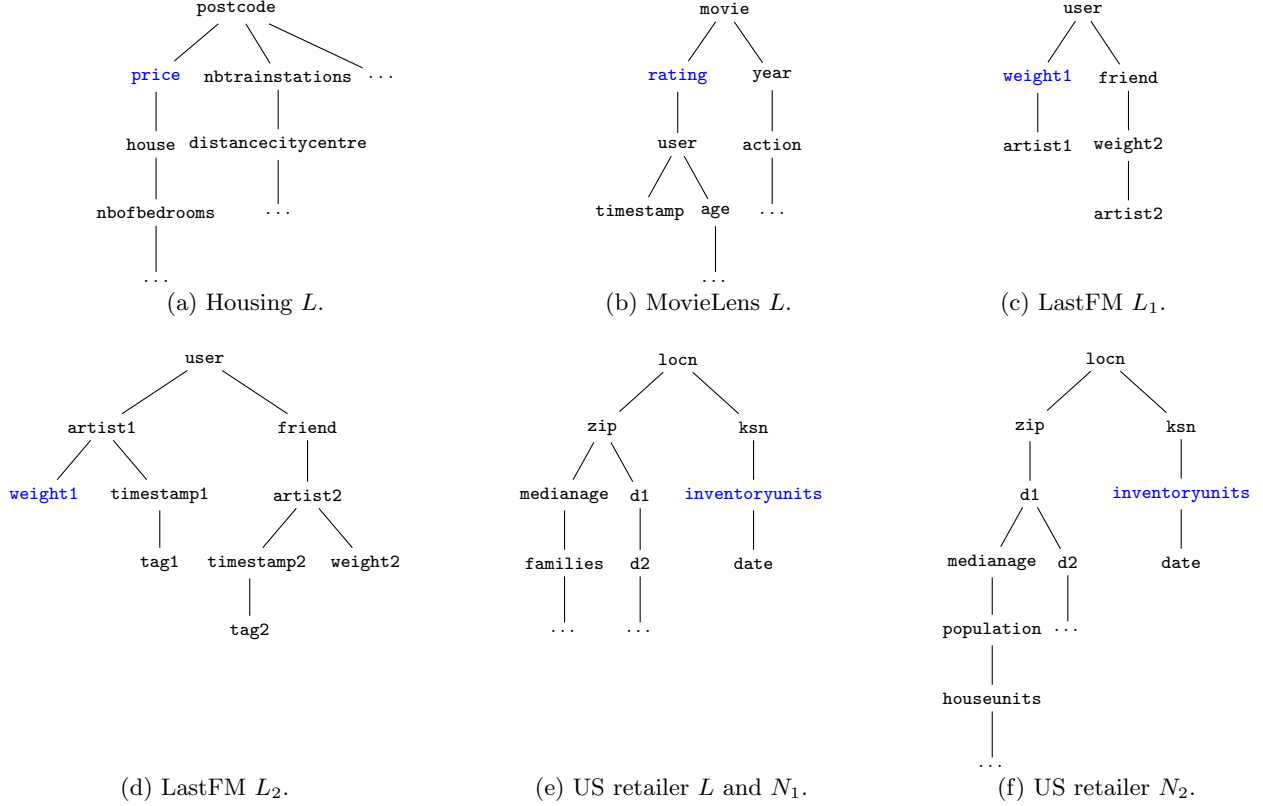


Figure 8: Snapshots of the d-trees considered in the experiments (root variable for intercept omitted).

many are occupied, number of children per household, number of males, females, and families), and **Location** (storing for each zipcode distances $d1$ to $d10$ to several other stores).

The training dataset is the natural join of the three relations and has 31 features. We considered three linear regression tasks that predict the amount of inventory units based on all other features. All join queries for these tasks are acyclic and under data-dependent functional dependencies (no store location can be in two zipcodes), they become hierarchical. Caching is not useful in their cases.

Task L considers the plain features. We consider an asymptotically optimal d-tree for the natural join query, a snapshot of which is presented in Figure 8(e). The relation **Inventory** is encoded along the path $locn/ksn/\dots$, the relation **Location** is encoded along the path $locn/zip/d1/\dots$, while **Census** is encoded along the path $zip/population/\dots$.

Task N_1 considers two interactions of features from the same relation and for which no restructuring is necessary: (i) Between median age and number of families (both from **Census**) and (ii) between different distances to other stores (both from **Location**). The first interaction looks at the effect on inventory units while correlating the number of families and the median age, since the two features are strongly related. The second interaction looks at the effect of having competitors close to the store and how the interaction of the two competitors changes the effect on the inventory units in the given store. Since both features occurring in an interaction are from the same table, we can use the d-tree from the linear case, which is asymptotically optimal.

Task N_2 considers two interactions: (i) Between population and number of house units (both from **Census**), and (ii) between median age and distance to another store (features of **Census** and **Location**, respectively). The first interaction uses the insight that if we have a large population but few houses, then many people live in the same house. This can give an indication of what the general population is like, and therefore, can have a meaningful impact on our prediction. The second interaction looks at how the age of a person influences her tendency to look for satisfying stores at a further distance.

The d-tree used for N_2 needs to additionally satisfy the constraint that the features from its second interaction (population and house units) are on the same root-to-leaf path; a snapshot of it in Figure 8(f). For both tasks N_1 and N_2 , each new interaction term can be seen as a newly-derived feature for learning in addition to the initial 31 features, hence for each task we have $31 + 2 = 33$ features.

• **LastFM** [10] has three relations: **Userfriends** (pairing friends in the social network from the LastFM online music system), **Userartists** (how often a user listens to a certain artist), and **Usertaggedartiststimestamps** (the user classification of artists and the time when a user rated artists). Our regression task is to predict how often a user would listen to an artist based on similar information for its friends. We consider two training datasets: L_1 joins two copies of **Userartists** with **Userfriends** to relate how often friends listen to the same artists; L_2 is L_1 where we also join in the

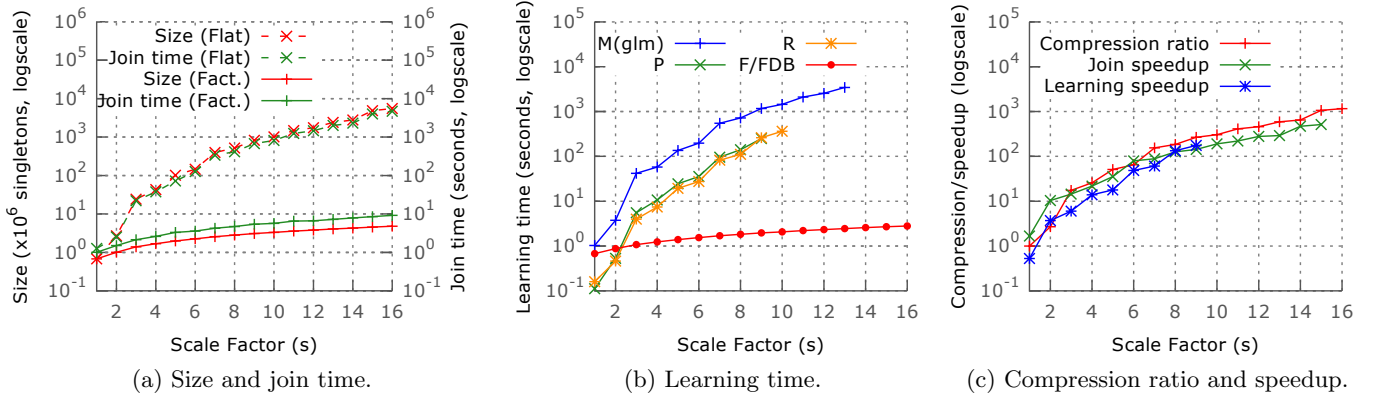


Figure 9: Housing dataset: (a) Size and join time for factorized vs. flat joins; (b) time for learning (excluding join time) with the four systems; (c) the learning speedup is for F/FDB relative to R, which is the fastest competitor in terms of learning time alone. P and R run out of memory starting at $s = 11$ for learning. M (glm) took longer than one hour after $s = 13$, and is not reported.

Usertaggedartiststimestamps copies of both friends:

L_1 : UF $\bowtie_{UF.user=UA1.user}$ UA1 $\bowtie_{UF.friend=UA2.user}$ UA2
 L_2 : UF $\bowtie_{UF.user=UA1.user}$ UA1 $\bowtie_{UF.friend=UA2.user}$ UA2 $\bowtie_{UF.user=UTA1.user}$ UTA1 $\bowtie_{UF.friend=UTA2.user}$ UTA2

Both queries are hierarchical since the **user** is common to all joined relations and use optimal d-trees for both of them. We illustrate them in Figures 8(c) and 8(d).

• **MovieLens** [16] has three relations: **Users** (age, gender, occupation, zipcode of users), **Movies** (movie year and its type, e.g., action, adventure, animation, children, and so on), and **Ratings** that users gave to movies on certain dates. The training dataset is the acyclic natural join of these tables and has 27 features. The regression task is to predict the rating given by a user to a movie. We depict a snapshot of the considered optimal d-tree in Figure 8(b). There are four versions of the MovieLens datasets and we only reported experimental findings for the largest available version (1M records) that has complete information for all three tables; there are two larger versions (10M and 20M) but without Users. We also experimented with the other versions (100K and the larger ones where we synthetically generated the Users relation) and found that they exhibit the same compression ratio and performance gain.

C. DATASET PREPARATION

The learning task requires to prepare the datasets. Firstly, we only kept features that represent quantities or Boolean flags over which we can learn and discarded string features (except if necessary for joins). Secondly, we normalized all number values of a feature A by mapping them to the $[0, 1]$ range of reals as follows. Let \min_A and \max_A be the minimum and respectively maximum value in the active domain of A . Then, a value v for A is normalized to $(v - \min_A) / \max_A$. Normalization is essential so that all

features have the same relative weight, e.g., avoiding that large date values represented in seconds since Jan 1, 1970 are more important than, say, small integer values representing the number of house bedrooms. It also preserves the cardinality of the join results from the original datasets.

D. FURTHER EXPERIMENTS

Figure 9 outlines the relative performance of F/FDB over its competitors on the Housing dataset along three dimensions: Compression ratio, performance speedup, and learning speedup. For the first two dimensions we essentially compare FDB against PostgreSQL, since F/FDB uses the factorized join computed by FDB, while the competitors use the flat join computed by PostgreSQL.

Figure 9(c) shows that Housing can be compressed by a factor of over 10^3 for scale $s = 16$. For a scale factor s , the flat join of all tables on postcode has $25K \times s^3 / 2 \times \log_2 s$ tuples, each of 27 values, whereas the factorized join stay linear in the size of the input tables. The gap between the sizes of flat and factorized joins thus follows a quadratic function in s .

Figures 9(a) and 9(c) confirm the finding from the real datasets for join computation: There is a significant speedup for factorized over flat, which is upper bounded by the data compression ratio.

Figure 9(b) reports the performance of learning for up to scale 16 for F/FDB, up to 13 for M(glm), and up to 10 for P and R. At scale factor 10, the compression ratio is 240 and F/FDB takes 2.2 seconds for learning. M(glm) was stopped at scale factor 13, since it exceeded the timeout of one hour. Learning with R and P already fails for scale factor 11 due to memory limitation. We further tried with scale factors up to 20, where the compression ratio is 1.9K and F/FDB takes 3.4 seconds for learning.