AC/DC: In-Database Learning Thunderstruck

Mahmoud Abo Khamis
RelationalAI, Inc

Hung Q. Ngo
RelationalAI, Inc

XuanLong Nguyen
University of Michigan

Dan Olteanu
University of Oxford

Maximilian Schleich
University of Oxford

ABSTRACT

We report on the design and implementation of the AC/DC gradient descent solver for a class of optimization problems over normalized databases. AC/DC decomposes an optimization problem into a set of aggregates over the join of the database relations. It then uses the answers to these aggregates to iteratively improve the solution to the problem until it converges.

The challenges faced by AC/DC are the large database size, the mixture of continuous and categorical features, and the large number of aggregates to compute. AC/DC addresses these challenges by employing a sparse data representation, factorized computation, problem reparameterization under functional dependencies, and a data structure that supports shared computation of aggregates.

To train polynomial regression models and factorization machines of up to 141K features over the join of a real-world dataset of up to 86M tuples, AC/DC needs up to 30 minutes on one core of a commodity machine. This is up to three orders of magnitude faster than its competitors R, MadLib, libFM, and TensorFlow whenever they finish and thus do not exceed memory limitation, 24-hour timeout, or internal design limitations.

Rode down the highway
Broke the limit, we hit the town
Went through to Texas, yeah Texas, and we had some fun.
– Thunderstruck (AC/DC)

1 INTRODUCTION

In this paper we report our on-going work on the design and implementation of AC/DC, a gradient descent solver for a class of optimization problems including ridge linear regression, polynomial regression, and factorization machines. It extends our prior system F for factorized learning of linear regression models [21] to capture non-linear models, categorical features, and model reparameterization under functional dependencies (FDs). Its design is but one fruit of our exploration of the design space for the AI engine currently under development at RelationalAI, Inc. It subscribes to a recent effort to bring analytics inside the database [7, 9, 12, 13, 21] and thereby avoid the non-trivial time spent on data import/export at the interface between database systems and statistical packages.

AC/DC[1] solves optimization problems over design matrices defined by feature extraction queries over databases of possibly many relations. It is a unified approach for computing both the optimizations and the underlying database queries; the two tasks not only live in the same process space, they are intertwined in one execution plan with asymptotically lower complexity than that of any one of them in isolation. This is possible due to several key contributions.

First, AC/DC *decomposes* a given optimization problem into a set of aggregates whose answers are fed into a gradient descent solver that iteratively approximates the solution to the problem until it reaches convergence. The aggregates capture the combinations of features in the input data, as required for computing the gradients of an objective function. They are group-by aggregates in case of combinations with at least one categorical feature and plain scalars for combinations of continuous features only. The former aggregates are grouped by the query variables with a categorical domain. Prior work on in-database machine learning mostly considered continuous features and one-hot encoded categorical features, e.g., [7, 9, 13, 21]. We avoid the expensive one-hot encoding of categorical features by a *sparse representation using group-by aggregates* [2]. Several tools, e.g., libFM [8, 20] for factorization machines and LIBSVM [6] for support vector machines, employ sparse data representations that avoid the redundancy introduced by one-hot encoding. These are computed on the result of the feature extraction query once it is exported out of the database.

Second, AC/DC *factorizes* the computation of these aggregates over the feature extraction query to achieve the lowest known complexity. We recently pinpointed the complexity of AC/DC [2]. The factorized computation of the queries obtained by decomposing optimization problems can be asymptotically faster than the computation of the underlying join query alone. This means that all machine learning approaches that work on a design matrix defined by the result of the database join are asymptotically suboptimal. The only other in-database learning system [13] that may outperform the underlying database join only works for generalized linear models over key-foreign key joins, does not decompose the task to AC/DC's granularity, and cannot recover the good complexity of AC/DC since it does not employ factorized computation.

Third, AC/DC *massively shares computation* across the aggregate-join queries. These *different* queries use the same underlying join and their aggregates have similar structures.

Fourth, AC/DC *exploits functional dependencies* (FDs) in the input database to reduce the dimensionality of the optimization problem. Prior work exploited FDs for Naïve Bayes classification and feature selection [14]. AC/DC can reparameterize (non)linear regression models with non-linear regularizers [2]. This reparameterization requires taking the inverses of matrices, which are sums of identity matrices and vector dot products. To achieve performance improvements by model reparameterization, AC/DC uses an interplay of its own data structure and the Eigen library for linear algebra [11].

The contributions of this paper are as follows.

[1]AC/DC supports both types of features, i.e., categorical and continuous, and fast processing. Its name allures at another duality, that of alternating and discrete currents, and at the fast-paced sound of a homonymous rock band.

- We translate the theory behind AC/DC into a practical tool. We explain AC/DC's approach used to factorized and shared computation of aggregates needed to solve optimizations.
- We report on the performance of AC/DC against MADlib [12], R [19], libFM [20], and TensorFlow[1]. We used a real-world dataset with five relations of 86M tuples in total and 141K features and trained a ridge linear regression model, a polynomial regression model of degree two, and a factorization machine of degree two.
  AC/DC needs up to 30 minutes on one core of a commodity machine and computes up to 46M aggregates. This is up to three orders of magnitude faster than its competitors whenever they finish and thus do not exceed memory limitation, 24-hour timeout, or internal design limitations.
- Our results confirm a counter-intuitive theoretical result from [2] stating that, under certain conditions, exploiting query structures and smart aggregation algorithms, one can train a model using batch gradient descent (BGD) *faster* than scanning through the data once. In particular, this means BGD can be faster than one epoch of stochastic gradient descent (SGD), in contrast to the commonly accepted assumption that SGD is typically faster than BGD.

The paper is organized as follows. Section 2 presents the class of optimization problems supported by AC/DC. Section 3 overviews the foundations of AC/DC. Section 4 describes the data structures used by AC/DC, its optimizations for factorized and shared computation of aggregates. Section 5 discusses the reparameterization of optimization problems and their computation using the aggregates computed in a previous step. Section 6 reports on experiments.

2 OPTIMIZATION PROBLEMS

We consider solving an optimization problem of a particular form inside a database. Suppose we have p parameters $\boldsymbol{\theta} = (\theta_1, \dots, \theta_p) \in \mathbb{R}^p$. Let n denote the number of numeric features. In our formulation, there is a positive integer m, and two vector-valued functions $g : \mathbb{R}^p \to \mathbb{R}^m$ and $h : \mathbb{R}^n \to \mathbb{R}^m$. Each component function g_j of $g = (g_j)_{j \in [m]}$ is a multivariate *polynomial*. Each component function h_j of $h = (h_j)_{j \in [m]}$ is a multivariate *monomial*.

Whether *maximum likelihood estimation*, *maximum a posteriori* estimation, or *structural risk minimization* approach is adopted, the "learning" phase in machine learning typically comes down to solving the optimization problem $\boldsymbol{\theta}^* := \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$, where

$$J(\boldsymbol{\theta}) := \sum_{(\mathbf{x}, y) \in T} \mathcal{L}\left(\langle g(\boldsymbol{\theta}), h(\mathbf{x}) \rangle, y\right) + \Omega(\boldsymbol{\theta}). \tag{1}$$

Here, $\mathcal{L}$ is some *loss function*, e.g., square loss, Ω is the regularizer, e.g., ℓ_1- or ℓ_2-norm of $\boldsymbol{\theta}$, and T is the training dataset with features (regressors) $\mathbf{x}$ and response (regressand) y. The features come in two flavors: continuous (e.g. price) and quantitative/categorical (e.g. city). The former are encoded as scalars, e.g., the price is 10.5. The latter are encoded as indicator vectors; e.g., if there were three cities then the vector $[1, 0, 0]$ indicates that the first city appears in the training record. Here are some examples relevant to this paper:

Example 2.1. The *ridge linear regression* (LR) model with response y and regressors $x_1, \dots, x_n$ has $p = n + 1$ parameters $\boldsymbol{\theta} = (\theta_0, \dots, \theta_n)$. For convenience, we set $x_0 = 1$ corresponding to the bias parameter θ_0. The data points $(\mathbf{x}, y) = (x_0, x_1, \dots, x_n, y)$ are taken from T. We would like to minimize the objective

$$J(\boldsymbol{\theta}) = \frac{1}{2|T|} \sum_{(\mathbf{x}, y) \in T} \left(\sum_{i=0}^{n} \theta_i x_i - y \right)^2 + \frac{\lambda}{2} \|\boldsymbol{\theta}\|_2^2 . \tag{2}$$

This is the ℓ_2-loss version of (1) with $m = n + 1$, where both g and h are identity functions: $\mathbf{g}(\boldsymbol{\theta}) = \boldsymbol{\theta}$, and $h(\mathbf{x}) = \mathbf{x}$.

Example 2.2. The *degree-d polynomial regression* (PR^d) model with response y and regressors $x_0 = 1, x_1, \dots, x_n$ has $p = m = \sum_{i=0}^{d} n^i$ parameters $\boldsymbol{\theta} = (\theta_{\mathbf{a}})$, where $\mathbf{a} = (a_1, \dots, a_n)$ is a tuple of non-negative integers such that $\sum_{i=1}^{n} a_i \le d$. In this case, $g(\boldsymbol{\theta}) = \boldsymbol{\theta}$ and h is defined by the component functions $h_{\mathbf{a}}(\mathbf{x}) = \prod_{i=1}^{n} x_i^{a_i}$.

Example 2.3. The *degree-2 rank-r factorization machines* (FaMa_2^r) model with regressors $x_0 = 1, x_1, \dots, x_n$ and regressand y has parameters $\boldsymbol{\theta}$ consisting of θ_i for $i \in \{0, \dots, n\}$ and $\theta_i^{(j)}$ for $i \in [n]$ and $j \in [r]$. Training FaMa_2^r corresponds to minimizing the function

$$J(\boldsymbol{\theta}) = \frac{1}{2|T|} \sum_{(\mathbf{x}, y) \in T} \left(\sum_{i=0}^{n} \theta_i x_i + \sum_{\substack{\{i,j\} \in \binom{[n]}{2} \\ \ell \in [r]}} \theta_i^{(\ell)} \theta_j^{(\ell)} x_i x_j - y \right)^2 + \frac{\lambda}{2} \|\boldsymbol{\theta}\|_2^2 .$$

The number of parameters is $p = 1 + n + rn$. Let $m = 1 + n + \binom{n}{2}$. Then, training FaMa_2^r corresponds precisely to optimizing (1) with square loss, ℓ_2-penalty and the following g and h functions:

$$\begin{aligned} h_S(\mathbf{x}) &= \prod_{i \in S} x_i, \text{ for } S \subseteq [n], |S| \le 2 \\ g_S(\boldsymbol{\theta}) &= \begin{cases} \theta_0 & \text{when } |S| = 0 \\ \theta_i & \text{when } S = \{i\} \\ \sum_{\ell=1}^{r} \theta_i^{(\ell)} \theta_j^{(\ell)} & \text{when } S = \{i, j\}. \end{cases} \end{aligned}$$

3 OVERVIEW OF AC/DC FOUNDATIONS

AC/DC is a batch gradient-descent (BGD) solver that optimizes the objective $J(\boldsymbol{\theta})$ over the training dataset T defined by a feature extraction query Q over a database D. Its high-level structure is given in Algorithm 1. AC/DC's inner loop repeatedly compute $J(\boldsymbol{\theta})$ and $\boldsymbol{\nabla} J(\boldsymbol{\theta})$, which can be sped up massively by factoring out the data-dependent computation from the optimization loop [2, 21]. The data-dependent computation is cast as computing aggregates over joins, which benefit from recent algorithmic advances [3, 4].

Algorithm 1: BGD with Armijo line search.

```
$\boldsymbol{\theta} \leftarrow$ a random point;
while not converged yet do
    $\alpha \leftarrow$ next step size        // Barzilai-Borwein [5];
    $\mathbf{d} \leftarrow \boldsymbol{\nabla} J(\boldsymbol{\theta})$;
    while $\left(J(\boldsymbol{\theta} - \alpha\mathbf{d}) \ge J(\boldsymbol{\theta}) - \frac{\alpha}{2} \|\mathbf{d}\|_2^2\right)$ do
        $\alpha \leftarrow \alpha/2$              // line search;
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha\mathbf{d}$;
```

From Optimization to Aggregates. Let us define the matrix $\Sigma = (\sigma_{ij})_{i,j\in[m]}$, the vector $\mathbf{c} = (c_i)_{i\in[m]}$, and the scalar s_Y by

$$\Sigma = \frac{1}{|Q(D)|} \sum_{(\mathbf{x},y)\in Q(D)} h(\mathbf{x})h(\mathbf{x})^\top \tag{3}$$

$$\mathbf{c} = \frac{1}{|Q(D)|} \sum_{(\mathbf{x},y)\in Q(D)} y \cdot h(\mathbf{x}) \tag{4}$$

$$s_Y = \frac{1}{|Q(D)|} \sum_{(\mathbf{x},y)\in Q(D)} y^2. \tag{5}$$

Then,

$$J(\boldsymbol{\theta}) = \frac{1}{2} g(\boldsymbol{\theta})^\top \Sigma g(\boldsymbol{\theta}) - \langle g(\boldsymbol{\theta}), \mathbf{c}\rangle + \frac{s_Y}{2} + \frac{\lambda}{2}\left\|\boldsymbol{\theta}\right\|^2 \tag{6}$$

$$\boldsymbol{\nabla} J(\boldsymbol{\theta}) = \frac{\partial g(\boldsymbol{\theta})^\top}{\partial \boldsymbol{\theta}} \Sigma g(\boldsymbol{\theta}) - \frac{\partial g(\boldsymbol{\theta})^\top}{\partial \boldsymbol{\theta}} \mathbf{c} + \lambda \boldsymbol{\theta} \tag{7}$$

The quantity $\frac{\partial g(\boldsymbol{\theta})^\top}{\partial \boldsymbol{\theta}}$ is a $p \times m$ matrix, and Σ is an $m \times m$ matrix of *sparse tensors*. Statistically, Σ is related to the covariance matrix, $\mathbf{c}$ to the correlation between the response and the regressors, and s_Y to the empirical second moment of the regressand. When all input features are numeric, each component function $h_j(\mathbf{x})$ is a monomial giving a scalar value. In real-world workloads there is always a mix of categorical and numeric features, where the monomials $h_j(\mathbf{x})$ become *tensor products*, and so the aggregates $\sigma_{ij} = h_i(\mathbf{x}) \otimes h_j(\mathbf{x})$ are also tensor products, represented by relational queries with group-by. The group-by variables for the aggregate query computing σ_{ij} are precisely the categorical variables occuring in the monomials defining the component functions h_i and h_j.

For example, if $h_i(\mathbf{x}) = x_A$ and $h_j(\mathbf{x}) = x_B$, where A and B are continuous features, then there is no group-by variable and σ_{ij} is represented as follows in SQL: `SELECT sum(A*B) from Q;` On the other hand, in the same setting if both A and B are categorical variables then there will be two group-by variables A and B, and σ_{ij} is represented by

```
SELECT A, B, count(*) FROM Q GROUP BY A, B;
```

These aggregates exploit the sparsity of the representation of σ_{ij} over categorical features to achieve succinct representation: The group-by clause ensures that only combinations of categories for the query variables A and B that exist in the training dataset are considered. The aggregates $\boldsymbol{c}$ and s_Y are treated similarly.

The above rewriting allows us to compute the data-dependent quantities Σ, $\mathbf{c}$, and s_Y in the objective function J and its gradient $\boldsymbol{\nabla} J$ once for all iterations of AC/DC. They can be computed efficiently *inside the database* as aggregates over the query Q. In theory, aggregates with different group-by clauses would require different evaluation strategies to attain the lowest known complexity [3, 4]. AC/DC settles instead for one evaluation strategy for all aggregates, as detailed in Section 4. This has at least two benefits. First, the underlying join is only computed once for all the aggregates. Second, the computation of the aggregates can be shared massively. These benefits may easily dwarf the gain of using specialised evaluation strategies for individual aggregates in case of very many aggregates (hundreds of millions) and large databases.

Reparameterization under Functional Dependencies. AC/DC exploits the FDs among variables in the feature extraction query Q to reduce the dimensionality of the optimization problem by eliminating functionally determined variables and re-parameterizing the model. We thus only compute the quantities Σ, $\mathbf{c}$, and s_Y on the subset of the features that are not functionally determined and solve the lower-dimensional optimization problem. The effect on the objective and its gradient is immediate and uniform across all optimization problems in our class: We have less terms to compute since the functionally determined variables are dropped. The effect on the non-linear penalty term Ω is however non-trivial and depends on the model at hand [2].

Following Section 4.1 in [2], we next explain the reparameterization of the ridge linear regression from Example 2.1 under an FD $A \to B$ where A and B are categorical variables. The categorical features $\mathbf{x}_A$ and $\mathbf{x}_B$ are represented by indicator vectors and the latter can be recovered from the former using the mapping between values for A and B in the input database. We can extract this map $R(B, A)$ that is a sparse representation of a matrix $\mathbf{R}$ and then $\mathbf{x}_B = \mathbf{R}\mathbf{x}_A$. The linear model then becomes:

$$\langle \boldsymbol{\theta}, \mathbf{x} \rangle = \sum_{j \notin \{A,B\}} \langle \boldsymbol{\theta}_j, \mathbf{x}_j \rangle + \left\langle \underbrace{\boldsymbol{\theta}_A + \mathbf{R}^\top \boldsymbol{\theta}_B}_{\boldsymbol{\gamma}_A}, \mathbf{x}_A \right\rangle.$$

The parameters $\boldsymbol{\theta}_A$ and $\boldsymbol{\theta}_B$ are dropped and new parameters $\boldsymbol{\gamma}_A$ are introduced for the categorical features $\mathbf{x}_A$. This new form can be used in the objective function (2). We can further optimize out $\boldsymbol{\theta}_B$ from the non-linear penalty term by setting the partial derivative of J with respect to $\boldsymbol{\theta}_B$ to 0. Then, the new penalty term becomes:

$$\begin{aligned} \|\boldsymbol{\theta}\|_2^2 &= \sum_{j \neq A,B} \|\boldsymbol{\theta}_j\|_2^2 + \left\|\boldsymbol{\gamma}_A - \mathbf{R}^\top \boldsymbol{\theta}_B\right\|_2^2 + \|\boldsymbol{\theta}_B\|_2^2 \\ &= \sum_{j \neq A,B} \|\boldsymbol{\theta}_j\|_2^2 + \left\langle (\mathbf{I}_A + \mathbf{R}^\top \mathbf{R})^{-1} \boldsymbol{\gamma}_A, \boldsymbol{\gamma}_A \right\rangle, \end{aligned}$$

where $\mathbf{I}_A$ is the identity matrix in the order of the active domain size of A. A similar formulation of the penalty term holds for polynomial regression models. The impact of FDs on AC/DC is twofold. First, fewer aggregates are needed to compute $J(\boldsymbol{\theta})$ and $\boldsymbol{\nabla} J(\boldsymbol{\theta})$, at the cost of a more complex regularizer term, requiring matrix inversion and multiplication. While they are expressible as database queries and can be evaluated efficiently similarly to the other aggregates [2], AC/DC uses instead the Eigen library for linear algebra to achieve significant speedups for model reparameterization over the strawman approach that does not exploit FDs.

4 AGGREGATE COMPUTATION IN AC/DC

An immediate approach to computing the aggregates in Σ, $\boldsymbol{c}$, and s_Y for a given optimization problem is to first materialize the result of the feature extraction query Q using an efficient query engine, e.g., a worst-case optimal join algorithm, and then compute the aggregates in one pass over the query result. This approach, however, is suboptimal, since the listing representation of the query result is highly redundant and not necessary for the computation of the aggregates. AC/DC avoids this redundancy by factorizing the computation of aggregates over joins, as detailed in Section 4.1. In a nutshell, this factorized approach unifies three powerful ideas: worst-case optimality for join processing, query plans defined by fractional hypertree decompositions of join queries, and an optimization that partially pushes aggregates past joins. AC/DC further exploits similarities across the aggregates to massively share their computation, as detailed in Section 4.2.

4.1 Factorized Computation of Aggregates

Factorized aggregate computation [3, 4, 17] relies on a variable order Δ for the query Q to avoid redundant computation.

Variable Orders. State-of-the-art query evaluation uses query plans that dictate the order in which the relations are joined. We use variable-at-a-time query plans, which we call variable orders. These are partial orders on the variables in the query, capture the join dependencies in the query, and dictate the order in which we solve each join variable. For each variable, we join all relations with that variable. Our choice is motivated by the complexity of join evaluation: Relation-at-a-time query plans are provably suboptimal, whereas variable-at-a-time query plans can be optimal [16].

Given a join query Q, a variable X *depends* on a variable Y if both are in the schema of a relation in Q.

Definition 4.1 (adapted from [18]). A variable order Δ for a join query Q is a pair (F, dep), where F is a rooted forest with one node per variable in Q, and dep is a function mapping each variable X to a set of variables in F. It satisfies the following constraints:

- For each relation in Q, its variables lie along the same root-to-leaf path in F.
- For each variable X, $dep(X)$ is the subset of its ancestors in F on which the variables in the subtree rooted at X depend.

Without loss of generality, we use variables orders that are trees instead of forests. We can convert a forest into a tree by adding to each relation the same dummy join variable that takes a single value. For a variable X in the variable order Δ, $anc(X)$ is the set of all ancestor variables of X in Δ. The set of variables in Δ (schema of a relation R) is denoted by $vars(\Delta)$ ($vars(R)$ respectively) and the variable at the root of Δ is denoted by $root(\Delta)$.

Example 4.2. Figure 3(a) shows a variable order for the natural join of relations $R(A, B, C)$, $T(B, D)$, and $S(A, E)$. Then, $anc(D) = \{A, B\}$ and $dep(D) = \{B\}$, i.e., D has ancestors A and B, yet it only depends on B. Given B, the variables C and D are independent of each other. For queries with group-by variables, we choose a variable order where these variables sit above the other variables [4].

Figure 1 presents the AC/DC algorithm for factorized computation of SQL aggregates over the feature extraction query Q. The backbone of the algorithm without the code in boxes explores the factorized join of the input relations $R_1, \ldots, R_d$ over a variable order Δ of Q. As it traverses Δ in depth-first preorder, it assigns values to the query variables. The assignments are kept in varMap and used to compute aggregates by the code in the boxes.

The relations are sorted following a depth-first pre-order traversal of Δ. Each call takes a range $[x_i, y_i]$ of tuples in each relation R_i. Initially, these ranges span the entire relations. Once the root variable A in Δ is assigned a value a from the intersection of possible A-values from the input relations, these ranges are narrowed down to those tuples with value a for A.

To compute an aggregate over the variable order Δ rooted at A, we first initialize the aggregate to zeros. This is needed since the aggregates might have been used earlier for different assignments of ancestor variables in Δ. We next check whether we previously computed the aggregate for the same assignments of variables in $dep(A)$, denoted by context, and cached it in a map cache_A. Caching is useful when $dep(A)$ is strictly contained in $anc(A)$, since this means that the aggregate computed at A does not need to be recomputed for distinct assignments of variables in $anc(A) \setminus dep(A)$. In this case, we probe the cache using as key the assignments in varMap of the $dep(A)$ variables: $\text{cache}_A[\text{context}]$. If we have already computed the aggregates over that assignment for $dep(A)$, then we can just reuse the previously computed aggregates and avoid recomputation.

If A is a group-by variable, then we compute a map from each A-value a to a function of a and aggregates computed at children of A, if any. If A is not a group-by variable, then we compute a map from the empty value () to such a function; in this latter case, we could have just computed the aggregate instead of the map though we use the map for uniformity. In case there are group-by variables under A, the computation at A returns maps whose keys are tuples over all these group-by variables in $vars(\Delta)$.

Example 4.3. Let us compute aggregates over the query Q with the variable order Δ from Figure 3(a). We first compute the assignments for A as $Q_A = \pi_A R \bowtie \pi_A T$. For each assignment $a \in Q_A$, we then find assignments for variables under A within the narrow ranges of tuples that contain a. The assignments for B in the context of a are given by $Q_B^a = \pi_B(\sigma_{A=a} R) \bowtie \pi_B S$. For each $b \in Q_B^a$, the assignments for C and D are given by $Q_C^{a,b} = \pi_C(\sigma_{A=a \wedge B=b} R)$ and $Q_D^b = \pi_D(\sigma_{B=b} S)$. Since D depends on B and not on A, the assignments for D under a given b are repeated for every occurrence of b with assignments for A. The assignments for E given $a \in Q_A$ are computed as $Q_E^a = \pi_E(\sigma_{A=a} T)$.

Consider the aggregate $\texttt{COUNT}(Q)$. The count at each variable X is computed as the sum over all value assignments of X of the product of the counts at the children of X in Δ; if X is a leaf in Δ, the product at children is considered 1. For our variable order, this computation is captured by the following factorized expression:

$$\texttt{COUNT} = \sum_{a \in Q_A} 1 \times \left(\sum_{b \in Q_B^a} 1 \times \left(\sum_{c \in Q_C^{a,b}} 1 \times V_D(b) \right) \right) \times \sum_{e \in Q_E^a} 1 \quad (8)$$

where $V_D(b) = \sum_{d \in Q_D^b} 1$ is cached the first time we encounter the assignment b for B and reused for all subsequent occurrences of this assignment under assignments for A.

Summing all X-values in the result of Q for a variable X is done similarly, with the difference that at the variable X in Δ we compute the sum of the values of X weighted by the product of the counts of their children. For instance, the aggregate $\texttt{SUM}(C * E)$ is computed over our variable order by the following factorized expression:

$$\texttt{SUM}(C \cdot E) = \sum_{a \in Q_A} 1 \times \left(\sum_{b \in Q_B^a} 1 \times \left(\sum_{c \in Q_C^{a,b}} c \times V_D(b) \right) \right) \times \sum_{e \in Q_E^a} e \quad (9)$$

```
aggregates (variable order $\Delta$, varMap, relation ranges $R_1[x_1, y_1], \dots, R_d[x_d, y_d]$)
$A = root(\Delta)$;    context = $\pi_{dep(A)}$(varMap);    reset(aggregates$_A$);    #aggregates = $|$aggregates$_A|$;

if  ($dep(A) \neq anc(A)$)   {   aggregates$_A$ = cache$_A$[context];   if  (aggregates$_A[0] \neq \emptyset$)  return;  }

foreach  $i \in [d]$  do  $R_i[x'_i, y'_i] = R_i[x_i, y_i]$;
foreach  $a \in \bigcap_{i \in [d]}$ such that $A \in vars(R_i)$ $\pi_A(R_i[x_i, y_i])$  do  {
  foreach  $i \in [d]$ such that $A \in vars(R_i)$  do  find range $R_i[x'_i, y'_i] \subseteq R_i[x_i, y_i]$ such that $\pi_A(R_i[x'_i, y'_i]) = \{(A : a)\}$;
  switch ($A$) :
     continuous feature  :    $\lambda_A = [\{() \mapsto 1\}, \{() \mapsto a^1\}, \dots, \{() \mapsto a^{2 \cdot degree}\}]$;
     categorical feature  :   $\lambda_A = [\{() \mapsto 1\}, \{a \mapsto 1\}]$;
     no feature  :            $\lambda_A = [\{() \mapsto 1\}]$;
  switch ($\Delta$) :
    leaf node $A$ :
      foreach  $l \in$ [#aggregates]  do  {   $[i_0] = \mathcal{R}_A[l]$;   aggregates$_A[l]$ += $\lambda_A[i_0]$;   }
    inner node $A(\Delta_1, \dots, \Delta_k)$ :
      foreach  $j \in [k]$  do  aggregates($\Delta_j$, varMap $\times \{(A : a)\}$, ranges $R_1[x'_1, y'_1], \dots, R_d[x'_d, y'_d]$);
      if  ($\forall j \in [k]$ : aggregates$_{root(\Delta_j)}[0] \neq \emptyset$)
        foreach  $l \in$ [#aggregates]  do { $[i_0, i_1, \dots, i_k] = \mathcal{R}_A[l]$;   aggregates$_A[l]$ += $\lambda_A[i_0] \otimes \bigotimes_{j \in [k]}$ aggregates$_{root(\Delta_j)}[i_j]$; }
}
if  ($dep(A) \neq anc(A)$)   cache$_A$[context] = aggregates$_A$;
```

Figure 1: Algorithm for computing aggregates aggregates$_A$**. Each aggregate is a map from tuples over its group-by variables to scalars. The parameters of the initial call are the variable order Δ of the feature extraction query, an empty map from variables to values, and the full range of tuples for each relation $R_1, \dots, R_d$ in the input database.**

To compute the aggregate SUM($C * E$) GROUP BY A, we compute SUM($C * E$) for each assignment for A instead of marginalizing away A. The result is a map from A-values to values of SUM($C * E$).

A good variable order may include variables that are not explicitly used in the optimization problem. This is the case of join variables whose presence in the variable order ensures a good factorization. For instance, if we remove the variable B from the variable order in Figure 3(a), the variables C, D are no longer independent and we cannot factorize the computation over C and D. AC/DC exploits the conditional independence enabled by B, but computes no aggregate over B if this is not required in the problem.

4.2 Shared Computation of Aggregates

Section 4.1 explains how to factorize the computation of one aggregate in Σ, $\boldsymbol{c}$, and s_Y over the join of database relations. In this section we show how to share the computation across these aggregates.

Example 4.4. Let us consider the factorized expression of the sum aggregates SUM(C) and SUM(E) over Δ:

$$\text{SUM}(C) = \sum_{a \in Q_A} 1 \times \left(\sum_{b \in Q_B^a} 1 \times \left(\sum_{c \in Q_C^{a,b}} c \times V_D(b) \right) \right) \times \sum_{e \in Q_E^a} 1 \quad (10)$$

$$\text{SUM}(E) = \sum_{a \in Q_A} 1 \times \left(\sum_{b \in Q_B^a} 1 \times \left(\sum_{c \in Q_C^{a,b}} 1 \times V_D(b) \right) \right) \times \sum_{e \in Q_E^a} e \quad (11)$$

We can share computation across the expressions (8) to (11) since they are similar. For instance, given an assignment b for B, all these aggregates need $V_D(b)$. Similarly, for a given assignment a for A, the aggregates (9) and (11) can share the computation of the sum aggregate over Q_E^a. For assignments $a \in Q_A$ and $b \in Q_B^a$, (9) and (10) can share the computation of the sum aggregate over $Q_C^{a,b}$.

AC/DC computes all aggregates together over a single variable order. It then shares as much computation as possible and significantly improves the data locality of the aggregate computation. AC/DC thus decidedly sacrifices the goal of achieving the lowest known complexity for individual aggregates for the sake of sharing as much computation as possible across these aggregates.

Aggregate Decomposition and Registration. For a model of degree *degree* and a set of variables $\{A_l\}_{l \in [n]}$, we have aggregates of the form SUM($\prod_{l \in [n]} A_l^{d_l}$), possibly with a group-by clause, such that $0 \leq \sum_{l \in [n]} d_l \leq 2 \cdot degree$, $d_l \geq 0$, and all categorical variables are turned into group-by variables. The reason for $2 \cdot degree$ is due to the gradient of the objective function (1), which here uses a square loss function $\mathcal{L}$ and pairs any two features of degree up to *degree*. Each aggregate is thus defined uniquely by a monomial $\prod_{l \in [n]} A_l^{d_l}$; we may discard the variables with exponent 0. For instance, the monomial for SUM($C * E$) is CE. The monomial for SUM($C * E$) GROUP BY A is **A**CE.

Aggregates can be decomposed into shareable components. Consider a variable order $\Delta = A(\Delta_1, \dots, \Delta_k)$, with root A and subtrees Δ_1 to Δ_k. We can decompose any aggregate α to be computed over Δ into $k + 1$ aggregates such that aggregate 0 is for A and aggregate $j \in [k]$ is for $root(\Delta_j)$. Then α is computed as the product of its $k + 1$ components. Each of these aggregates is defined by the projection

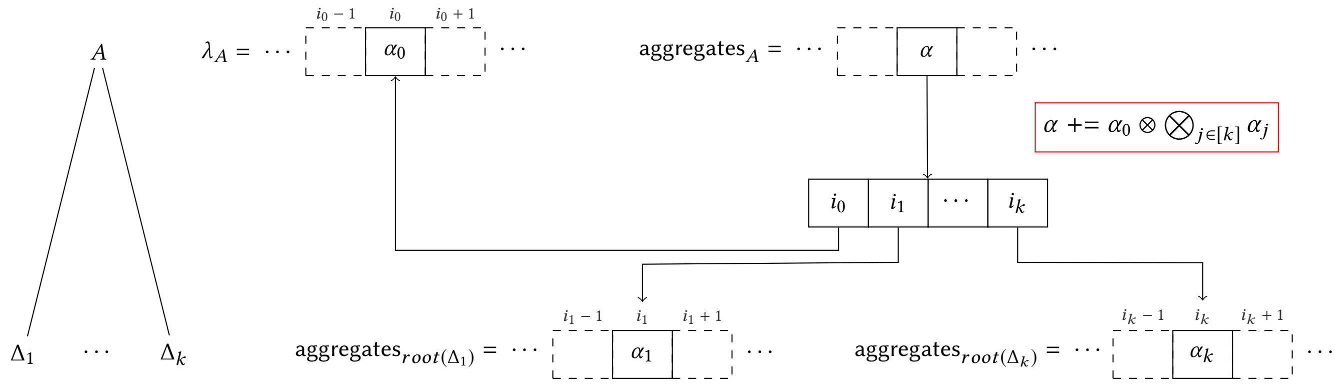

A
Δ_1
$\cdots$
Δ_k
$\lambda_A = \cdots$
$i_0 - 1$
i_0
$i_0 + 1$
α_0
$\text{aggregates}_A = \cdots$
α
$\alpha \mathrel{+}= \alpha_0 \otimes \bigotimes_{j \in [k]} \alpha_j$
i_0 i_1 $\cdots$ i_k
$\text{aggregates}_{root(\Delta_1)} = \cdots$
$i_1 - 1$
i_1
$i_1 + 1$
α_1
$\text{aggregates}_{root(\Delta_k)} = \cdots$
$i_k - 1$
i_k
$i_k + 1$
α_k

Figure 2: Index structure provided by the aggregate register for a particular aggregate α that is computed over the variable order $\Delta = A(\Delta_1, \ldots, \Delta_k)$. The computation of α is expressed as the sum of the Cartesian products of its aggregate components provided by the indices $i_0, \ldots, i_k$.

of the monomial of α onto A or $vars(\Delta_j)$. The aggregate j is then pushed down the variable order and computed over the subtree Δ_j. If the projection of the monomial is empty, then the aggregate to be pushed down is SUM(1), which computes the size of the join defined by Δ_j. If several aggregates push the same aggregate to the subtree Δ_j, this is computed only once for all of them.

The decomposed aggregates form a hierarchy whose structure is that of the underlying variable order Δ. The aggregates at a variable X are denoted by aggregates_X. All aggregates are to be computed at the root of Δ, then fewer are computed at each of its children and so on. This structure is the same regardless of the input data and can be constructed before data processing. We therefore construct at compile time for each variable X in Δ an aggregate register $\mathcal{R}_X$ that is an array of all aggregates to be computed over the subtree of Δ rooted at X. This register is used as an index structure to facilitate the computation of the actual aggregates. More precisely, an entry for an aggregate α in the register of X is labeled by the monomial of α and holds an array of indices of the components of α located in the registers at the children of X in Δ and in the local register Λ_X of X. Figure 2 depicts this construction.

The hierarchy of registers in Figure 3(b) forms an index structure that is used by AC/DC to compute the aggregates. This index structure is stored as one contiguous array in memory, where the entry for an aggregate α in the register comes with an auxiliary array with the indices of α's aggregate components. The aggregates are ordered in the register so that we increase sequential access, and thus cache locality, when updating them.

Example 4.5. Let us compute a regression model of degree 1 over a dataset defined by the join of the relations $R(A, B, C)$, $S(B, D)$, and $T(A, E)$. We assume that B and E are categorical features, and all other variables are continuous. The quantities $(\Sigma, \boldsymbol{c}, s_Y)$ require the computation of the following aggregates: SUM(1), SUM(X) for each variable X, and SUM($X * Y$) for each pair of variables X and Y.

Figure 3(a) depicts a variable order Δ for the natural join of three relations, and Figure 3(b) illustrates the aggregate register that assigns a list of aggregates to each variable in Δ. The aggregates are identified by their respective monomials (the names in the register entries). The categorical variables are shown in bold. Since they are treated as group-by variables, we do not need aggregates whose monomials include categorical variables with exponents higher than 1. Any such aggregate is equivalent to the aggregate whose monomial includes the categorical variable with degree 1 only.

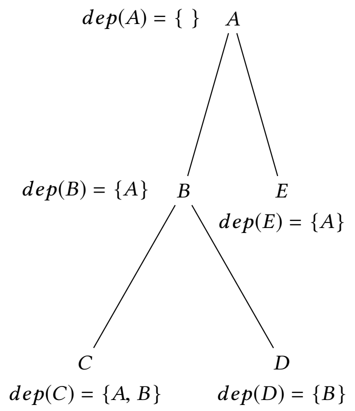
The register $\mathcal{R}_A$ for the root A of Δ has all aggregates needed to compute the model. The register $\mathcal{R}_B$ has all aggregates from $\mathcal{R}_A$ defined over the variables in the subtree of Δ rooted at B. The variables C, D, and E are leaf nodes in Δ, so the monomials for the aggregates in the registers $\mathcal{R}_C$, $\mathcal{R}_D$, and $\mathcal{R}_E$ are the respective variables only. We use two additional registers Λ_A and Λ_B, which hold the aggregates corresponding to projections of the monomials of the aggregates in $\mathcal{R}_A$, and respectively $\mathcal{R}_B$, onto A, respectively B. For a leaf node X, the registers Λ_X and $\mathcal{R}_X$ are the same.

A path between two register entries in Figure 3(b) indicates that the aggregate in the register above uses the result of the aggregate in the register below. For instance, each aggregate in $\mathcal{R}_B$ is computed by the product of one aggregate from Λ_B, $\mathcal{R}_C$, and $\mathcal{R}_D$. The fan-in of a register entry thus denotes the amount of sharing of its aggregate: All aggregates from registers above with incoming edges to this aggregate share its computation. For instance, the aggregates with monomials AB, AC, and AD from $\mathcal{R}_A$ share the computation of the aggregate with monomial A from Λ_A as well as the count aggregate from $\mathcal{R}_E$. Their computation uses a sequential pass over the register $\mathcal{R}_B$. This improves performance and access locality as $\mathcal{R}_B$ can be stored in cache and accessed to compute all these aggregates.

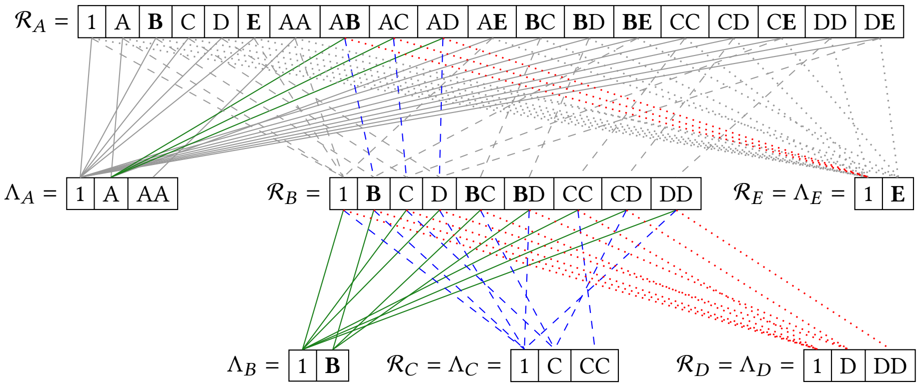
Aggregate Computation. Once the aggregate registers are in place, we can ingest the input database and compute the aggregates over the join of the database relations following the factorized structure given by a variable order. The algorithm in Figure 1 does precisely this. Section 4.1 explained the factorized computation of a single aggregate over the join. We explain here the case of several aggregates organized into the aggregate registers. This is stated by the pseudocode in the red boxes.

Each aggregate is uniformly stored as a map from tuples over their categorical variables to payloads that represent the sums over the projection of its monomial on all continuous variables. If the aggregate has no categorical variables, the key is the empty tuple.

For each possible A-value a, we first compute the array λ_A that consists of the projections of the monomials of the aggregates onto A. If A is categorical, then we only need to compute the 0 and 1



(a) Variable Order Δ.



$\mathcal{R}_A$ = 1 A **B** C D **E** AA A**B** AC AD A**E** **BC** **B**D **BE** CC CD C**E** DD D**E**
Λ_A = 1 A AA
$\mathcal{R}_B$ = 1 **B** C D **B**C **B**D CC CD DD
$\mathcal{R}_E = \Lambda_E$ = 1 **E**
Λ_B = 1 **B**
$\mathcal{R}_C = \Lambda_C$ = 1 C CC
$\mathcal{R}_D = \Lambda_D$ = 1 D DD

(b) Aggregate Registers.

Figure 3: (a) Variable order Δ for the natural join of the relations R(A,B,C), S(B,D), and T(A,E); (b) Aggregate registers for the aggregates needed to compute a linear regression model with degree 1 over Δ. Categorical variables are shown in bold.

powers of a. If A is continuous, we need to compute all powers of A from 0 to $2 \cdot degree$. If A is not a feature used in the model, then we only compute a trivial count aggregate.

We update the value of each aggregate α using the index structure depicted in Figure 2 as we traverse the variable order bottom up. Assume we are at a variable A in the variable order. In case A is a leaf, the update is only a specific value in the local register λ_A. In case the variable A has children in the variable order, the aggregate is updated with the Cartesian product of all its component aggregates, i.e., one value from λ_A and one aggregate for each child of A. The update value can be expressed in SQL as follows. Assume the aggregate α has group-by variables C, which are partitioned across A and its k children. Assume also that α's components are α_0 and $(\alpha_j)_{j\in[k]}$. Recall that all aggregates are maps, which we may represent as relations with columns for keys and one column P for payload. Then, the update to α is:

$$\texttt{SELECT } C, (\alpha_0.P * \ldots * \alpha_k.P) \texttt{ AS } P \texttt{ FROM } \alpha_0, \ldots, \alpha_k;$$

Further Considerations. The auxiliary arrays that provide the precomputed indices of aggregate components within registers speed up the computation of the aggregates. Nevertheless, they still represent one extra level of indirection since each update to an aggregate would first need to fetch the indices and then use them to access the aggregate components in registers that may not be necessarily in the cache. We have been experimenting with an aggressive aggregate compilation approach that resolves all these indices at compile time and generates the specific code for each aggregate update. In experiments with linear regression, this compilation leads to a 4× performance improvements. However, the downside is that the AC/DC code gets much larger and the C++ compiler needs much more time to compile it. For higher-degree models, it can get into situations where the C++ compiler crashes. We are currently working on a hybrid approach that partially resolves the indices while maintaining a reasonable code size.

5 THE INNER LOOP OF BGD

As shown in Section 3, the gradient descent solver repeatedly computes $J(\boldsymbol{\theta})$ and $\boldsymbol{\nabla} J(\boldsymbol{\theta})$, which require matrix-vector and vector-vector multiplications over the quantities $(\Sigma, \mathbf{c}, s_Y)$. We discuss here the computation that involves the Σ matrix.

It is possible that several entries $\sigma_{ij} \in \Sigma$ map to the same aggregate query that is computed by AC/DC. Consider, for instance, the following scalar entries in the feature mapping vector h: $h_i(\mathbf{x}) = x_a$, $h_j(\mathbf{x}) = x_b \cdot x_c$, $h_k(\mathbf{x}) = x_b$, $h_l(\mathbf{x}) = x_a \cdot x_c$, $h_m(\mathbf{x}) = x_c$, and $h_n(\mathbf{x}) = x_a \cdot x_b$. By definition of Σ, any pair-wise product of these entries in h corresponds to one entry in Σ. The entries σ_{ij}, σ_{lk}, and σ_{mn} (as well as their symmetric counterparts) all map to the same aggregate $\texttt{SUM}(A \cdot B \cdot C) = \sum_{(\mathbf{x},y)\in Q(D)} x_a \cdot x_b \cdot x_c$. To avoid this redundancy, we use a sparse representation of Σ, which assigns to each distinct aggregate query a list of index pairs (i, j) that contains one pair for each entry $\sigma_{ij} \in \Sigma$ that maps to this query.

AC/DC operates directly over the sparse representation of Σ. Consider the matrix vector product $\mathbf{p} = \Sigma g(\boldsymbol{\theta})$, and let A be the root of the variable order Δ. We compute $\mathbf{p}$ by iterating over all aggregate maps $\alpha \in \text{aggregates}_A$, and for each index pair (i, j) in Σ that is assigned to α, we add to the i's entry in $\mathbf{p}$ with the product of α and j's entry in $g(\boldsymbol{\theta})$. If $i \neq j$, we also add to j's entry in $\mathbf{p}$ with the product of α and i's entry in $g(\boldsymbol{\theta})$.

Regularizer under FDs. Section 3 explains how to rewrite the regularizer for a ridge linear regression model under the FD $A \longrightarrow B$. To compute the regularizer, we need to first construct the relation $\mathbf{R}(B, A)$, and then compute the inverse of the matrix $\mathbf{D} = (\mathbf{I}_A + \mathbf{R}^\top\mathbf{R})$. Each (i, j) entry in $\mathbf{R}^\top\mathbf{R}$ is 1, if the FD maps the A-values a_i and a_j to the same B-value b. Otherwise, the entry (i, j) is zero.

To facilitate the construction of the matrix $\mathbf{D}$, we construct $\mathbf{R}$ as a map that groups the tuples of $\mathbf{R}$ by B-values. Thus, we construct a mapping from B-values to sets of A-values during the computation of the factorized aggregates over the variable order: When the algorithm is at the variable B, we add all possible assignments of A that can be paired with a particular B-value b to the payload of the tuple in the map that has b as key.

We can use this representation of **R** to iterate over the payload set, and for any two A-values a_i, a_j in the payload set, we increment the corresponding index in **D** by one. We store this matrix as a sparse matrix in the format used by the Eigen library for linear algebra, and then use Eigen's Sparse Cholesky Decomposition to compute the inverse of **D**, and ultimately the solution for the regularizer.

6 EXPERIMENTS

We report on the performance of learning regression models and factorization machines over a real dataset used in retail applications.

Systems. We consider two variants of our system: The plain AC/DC and its extension AC/DC+FD that exploits functional dependencies. We also report on five competitors: F learns linear regression models and one-hot encodes the categorical features [21]; MADlib [12] 1.8 uses *ols* to compute the closed-form solution of polynomial regression models (MADlib also supports generalized linear models, but this is consistently slower than *ols* for our experiments and we do not report it here); R [19] 3.0.2 uses *lm* (linear model) based on QR-decomposition [10]; TensorFlow [1] 1.6 uses the LinearRegressor estimator with *ftrl* optimization [15]; and libFM [20] 1.4.2 supports factorization machines.

The competitors come with strong limitations. MADlib inherits the limitation of at most 1600 columns per relation from its PostgreSQL host. The MADlib one-hot encoder transforms a categorical variable with n distinct values into n columns. Therefore, the number of distinct values across all categorical variables plus the number of continuous variables in the input data cannot exceed 1600. R limits the number of values in their data frames to $2^{31} - 1$. There exist R packages, e.g., *ff*, which work around this limitation by storing data structures on disk and mapping only chunks of data in main memory. The biglm package can compute the regression model by processing one ff-chunk at a time. Chunking the data, however, can lead to rank deficiencies within chunks (feature interactions missing from chunks), which causes biglm to fail. Biglm fails in all our experiments due to this limitation, and, thus, we are unable to benchmark against it. LibFM requires as input a zero-suppressed encoding of the join result. Computing this representation is an expensive intermediary step between exporting the query result from the database system and importing the data. To compute the model, we used its more stable MCMC variant with a fixed number of runs (300); its SGD implementation requires a fixed learning rate α and does not converge. AC/DC uses the adaptive learning rate from Algorithm 1 and runs until the parameters have converged with high accuracy (for FaMa, it uses 300 runs).

TensorFlow uses a user-defined iterator interface to load a batch of tuples from the training dataset at a time. This iterator defines a mapping from input tuples to (potentially one-hot encoded) features and is called directly by the learning algorithm. Learning over batches requires a random shuffling of the input data, which nevertheless requires loading the entire dataset into memory in TensorFlow. This failed for our experiments and we therefore report its performance without shuffling the input data. We benchmark TensorFlow for LR only as it does not provide functionality to create all pairwise interaction terms for PR and FaMa, third-party implementations of these models relied on python packages that failed to load our datasets. The optimal batch size for our experiments is 100,000 tuples. Smaller batch sizes require loading too many batches, very large batches cannot fit into memory. Since TensorFlow requires a fixed number of iterations, we decided to report the times to do one epoch over the dataset (i.e. computing 840 batches). This means that the algorithm learned over each input tuple once, in practice it is necessary to optimize with several epochs to get a good model.

Experimental Setup. All experiments were performed on an Intel(R) Core(TM) i7-4770 3.40GHz/64bit/32GB with Linux 3.13.0 and g++4.8.4. We report wall-clock times by running each system once and then reporting the average of four subsequent runs with warm cache. We do not report the times to load the database into memory for the join as they can differ substantially between the systems and are orthogonal to this work. All relations are given sorted by their join attributes.

Dataset. We experimented with a real-world dataset in the retail domain for forecasting user demands and sales. It has five relations: `Inventory` (storing information about the inventory units for products (sku) in a store (locn), at a given date), `Census` (storing demographics information per zipcode such as population, median age, repartition per ethnicities, house units and how many are occupied, number of children per household, number of males, females, and families), `Location` (storing the zipcode for each store and distances to several other stores), `Item` (storing the price and category, subcategory, and categoryCluster for each products), and `Weather` (storing weather conditions such as mean temperature, and whether it rains, snows, or thunders for each store at different dates). The natural join of these five relations is acyclic and has 43 variables. We compute the join over the variable order: `(locn (zip (`*vars*`(Census),`*vars*`(Location)), date(sku (`*vars*`(Item)),`*vars*`(Weather))))`. The following 8 variables are categorical: locn, zip, category, subcategory, categoryCluster, snow, rain, thunder. The variables sku and date are not features in our models. We design 4 fragments of our dataset with an increasing number of categorical features. v_1 is a partition of the entire dataset that was specifically tailored to work within the limitations of R. It includes all categorical variables as features except for locn and zip. v_2 computes the same model as v_1 but over all rows in the data (5× larger than v_1). v_3 adds zip as a categorical feature to v_2; v_2 and v_3 were designed to work within the limitations of MADlib. v_1 to v_3 have no functional dependency. Finally, v_4 has all categorical variables and the functional dependency locn→zip.

We learned LR, PR_2, and FaMa_2^8 models that predict the amount of inventory units based on all other features.

Summary of findings. Table 1 shows our findings. AC/DC+FD is the fastest system in our experiments. It is orders of magnitude faster than the competitors or can finish successfully when the others exceed memory/time/internal design limits. The performance gap is attributed to several optimizations: (1) our system is an end-to-end in-database solution as it performs the join together with the aggregates and therefore avoids the costly data export-import step at the interface between database systems and statistical packages (almost 50% time cost for R); (2) it avoids the join materialization (6% time cost for R); (3) it factorizes the computation of the aggregates and the underlying join (20× compression factor); (4) it massively shares the computation of large (up to 46M) sets of distinct, non-zero aggregates (16K× faster due to sharing for PR); (5) it decouples the computation of the aggregates on the input data from the parameter convergence step and thus avoids scanning the join result per iteration (we need on average 400 iterations); (6) it avoids the upfront one-hot encoding that comes with higher asymptotic complexity and prohibitively large covariance matrices by only computing non-identical, non-zero matrix entries (for PR_2 and our dataset v_4, this leads to a 259× reduction factor in the number of aggregates to compute!); (7) it exploits functional dependencies in the input data to reduce the number of features of the model (3.5x improvement factor). None of our competitors employ all of these optimizations. Our earlier prototype F supports (1) to (5), though with limited form of (4). MADlib supports (1) and (2) and does not need (5) as it computes the closed-form solution. R and TensorFlow represent one-hot encoded features as zero-suppressed matrices, but still require a transformation of the data before learning. LibFM also has a sparse representation of the feature vectors, which is given by zero-suppressed representation of the input data.

Categorical features. As we move from v_1/v_2 to v_4, we increase the number of categorical features by approx. 50 times for LR (from 55 to 2.7K) and 65 times for PR_2 and $FaMa_2^8$ (from 2.4K to 154K). For LR, this increase only led to a 7× decrease in performance of **AC/DC** and at least 9× for MADlib (we stopped MADlib after 22 hours). For PR_2, this yields a 13.7× performance decrease for **AC/DC**. This behavior remains the same for **AC/DC**'s aggregate computation step with or without the convergence step, since the latter is dominated by the former by up to three orders of magnitude. This sub-linear behavior is partly explained by the ability of our system to process large sets of aggregates much faster in bulk than individually (it takes 34 seconds for 43 count aggregates but only 1819 seconds for 37M aggregates!), and by the same-order increase in the number of aggregates: 65 (51) times more distinct non-zero aggregates in v_4 vs v_2 for LR (and respectively PR_2 and $FaMa_2^8$).

The performance of TensorFlow is largely invariant to the increase in the number of categorical features, since its internal mapping from tuples in the training dataset to the sparse representation of the features vector remains the same.

Increasing database size. A 5× increase in database size (and join result) from v_1 to v_2 leads to a similar decrease factor in performance for F and AC/DC on all models, since the number of features and aggregates stay roughly the same and the join is acyclic and is processed in linear time. The performance of MADlib and TensorFlow follows the same trend for LR. MADlib runs out of time (22 hours) for both datasets for PR_2 models. R cannot cope with the size increase due to internal design limitations.

One-hot encoding vs sparse representations with group-by aggregates. One-hot encoding categorical features leads to a large number of zero and/or redundant entries in the Σ matrix. For instance, for v_4 and PR_2, the number of features is $m = 154,595$, and then the upper half of Σ would have $m(m+1)/2 \approx 1.19 \times 10^{10}$ entries! Most of these are either zero or repeating. In contrast, AC/DC's sparse representation only considers 46M non-zero and distinct aggregates. The number of aggregates is reduced by 259x!

Our competitors require the data be one-hot encoded *before* learning. The static one-hot encoding took (in seconds): 28.42 for R on v_1; 9.41 for F on v_1 and v_2; 2 for MADlib on v_1 to v_3; and slightly more than an hour for libFM, due to the expensive zero-suppression step. TensorFlow one-hot encodes on the fly during the learning phase and cannot be reported separately.

Functional dependencies. The FD in our dataset v_4 has a twofold effect on AC/DC (all other systems do not exploit FDs): it effectively reduces the number of features and aggregates, which leads to better performance of the in-database precomputation step; yet it requires a more elaborate convergence step due to the more complex regularizer. For LR, the aggregate step becomes 2.3× faster, while the convergence step increases 13×. Nevertheless, the convergence step takes at most 2% of the overall compute time in this case. For degree-2 models, the FD brings an improvement by a factor of 3.5× for PR_2, and 3.87× for $FaMa_2^8$. This is due to a 10% decrease in the number of categorical features, which leads to a 20% decrease in the number of group-by aggregates.

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

REFERENCES

[1] Martín Abadi et al. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR* abs/1603.04467 (2016). http://arxiv.org/abs/1603.04467

[2] Mahmoud Abo Khamis, Hung Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. In-Database Learning with Sparse Tensors. In *PODS*.

[3] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. In *PODS*. 13–28.

[4] Nurzhan Bakibayev, Tomás Kociský, Dan Olteanu, and Jakub Závodný. 2013. Aggregation and Ordering in Factorised Databases. *PVLDB* 6, 14 (2013), 1990–2001.

[5] Jonathan Barzilai and Jonathan M. Borwein. 1988. Two-point step size gradient methods. *IMA J. Numer. Anal.* 8, 1 (1988), 141–148.

[6] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM Trans. Intell. Syst. Tech.* 2 (2011), 27:1–27:27. Issue 3. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[7] Tarek Elgamal, Shangyu Luo, Matthias Boehm, Alexandre V. Evfimievski, Shirish Tatikonda, Berthold Reinwald, and Prithviraj Sen. 2017. SPOOF: Sum-Product Optimization and Operator Fusion for Large-Scale Machine Learning. In *CIDR*.

[8] Rong-En Fan et al. 2008. LIBLINEAR: A Library for Large Linear Classification. *J. Mach. Learn. Res.* 9 (2008), 1871–1874.

[9] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. 2012. Towards a unified architecture for in-RDBMS analytics. In *SIGMOD*. 325–336.

[10] J. G. F. Francis. 1961. The QR transformation: A unitary analogue to the LR transformation–Part 1. *Comput. J.* 4, 3 (1961), 265–271.

[11] Gaël Guennebaud et al. 2010. Eigen v3. http://eigen.tuxfamily.org. (2010).

[12] Joseph M. Hellerstein and et al. 2012. The MADlib Analytics Library or MAD Skills, the SQL. *PVLDB* 5, 12 (2012), 1700–1711.

[13] Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. 2015. Learning Generalized Linear Models Over Normalized Data. In *SIGMOD*. 1969–1984.

[14] Arun Kumar, Jeffrey F. Naughton, Jignesh M. Patel, and Xiaojin Zhu. 2016. To Join or Not to Join?: Thinking Twice about Joins before Feature Selection. In *SIGMOD*. 19–34.

[15] H. Brendan McMahan and et al. 2013. Ad Click Prediction: A View from the Trenches. In *KDD*. 1222–1230.

[16] Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2013. Skew Strikes Back: New Developments in the Theory of Join Algorithms. In *SIGMOD Rec.* 5–16.

[17] Dan Olteanu and Maximilian Schleich. 2016. Factorized Databases. *SIGMOD Rec.* 45, 2 (2016), 5–16.

[18] Dan Olteanu and Jakub Závodný. 2015. Size Bounds for Factorised Representations of Query Results. *ACM TODS* 40, 1 (2015), 2.

[19] R Core Team. 2013. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, www.r-project.org.

[20] Steffen Rendle. 2012. Factorization Machines with libFM. *ACM Trans. Intell. Syst. Technol.* 3, 3 (2012), 57:1–57:22.

[21] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning Linear Regression Models over Factorized Joins. In *SIGMOD*. 3–18.

		v_1	v_2	v_3	v_4
Join Representation	Listing	774M	3.614G	3.614G	3.614G
(#values)	Factorized	37M	169M	169M	169M
Compression	Fact/List	20.9×	21.4×	21.4×	21.4×
Join Computation (PSQL) for R, TensorFlow, libFM		50.63	216.56	216.56	216.56
Factorized Computation of 43 Counts over Join		8.02	34.15	34.15	34.15
Linear regression					
Features	without FDs	33 + 55	33+55	33+1340	33+2702
(continuous+categorical)	with FDs	same as above, there are no FDs			33+2653
Aggregates	without FDs	595+2,418	595+2,421	595+111,549	595+157,735
(scalar+group-by)	with FDs	same as above, there are no FDs			595+144,589
MADLib (ols)	Learn	1,898.35	8,855.11	$> 79,200.00$	–
R (QR)	Export/Import	308.83	–	–	–
	Learn	490.13	–	–	–
TensorFlow (FTLR)	Export/Import	74.72	372.70	372.70	372.70
(1 epoch, batch size 1K)	Learn	2,762.50	12,710.53	12,724.94	12,708.11
F	Aggregate	93.31	424.81	OOM	OOM
	Converge (runs)	0.01 (359)	0.01 (359)		
AC/DC	Aggregate	25.51	116.64	117.94	895.22
	Converge (runs)	0.02 (343)	0.02 (367)	0.42 (337)	0.66 (365)
AC/DC+FD	Aggregate	same as **AC**			380.31
	Converge (runs)	there are no FDs			8.82 (366)
Speedup of **AC/DC+FD** over	MADlib	74.36×	75.91×	$> 669.14\times$	∞
	R	33.28×	∞	∞	∞
	TensorFlow	113.12×	114.01×	112.49×	34.17×
	F	3.65×	3.64×	∞	∞
	AC/DC	same as **AC/DC**, there are no FDs			2.30×
Polynomial regression degree 2					
Features	without FDs	562+2,363	562+2,366	562+110,209	562+154,033
(continuous+categorical)	with FDs	same as above, there are no FDs			562+140,936
Aggregates	without FDs	158k+742k	158k+746k	158k+65,875k	158k+46,113k
(scalar+group-by)	with FDs	same as above, there are no FDs			158k+36,712k
MADlib (ols)	Learn	$> 79,200.00$	$> 79,200.00$	$> 79,200.00$	–
AC/DC	Aggregate	132.43	517.40	820.57	7,012.84
	Converge (runs)	3.27 (321)	3.62 (365)	349.15 (400)	115.65 (200)
AC/DC+FD	Aggregate	same as **AC/DC**			1,819.80
	Converge (runs)	there are no FDs			219.51 (180)
Speedup of **AC/DC+FD** over	MADlib	$> 583.64\times$	$> 152.01\times$	$> 67.71\times$	∞
	AC/DC	same as **AC/DC**, there are no FDs			3.50×
Factorization machine degree 2 **rank** 8					
Features	without FDs	530+2,363	530+2,366	530+110,209	530+154,033
(continuous+categorical)	with FDs	same as above, there are no FDs			562+140,936
Aggregates	without FDs	140k+740k	140k+744k	140k+65,832k	140k+45,995k
(scalar+group-by)	with FDs	same as above, there are no FDs			140k+36,595k
libFM (MCMC)	Export/Import	412.84	1462.54	3,096.90	3,368.06
	Learn (runs)	19,692.90 (300)	103,225.50 (300)	79,839.13 (300)	87,873.75 (300)
AC/DC	Aggregate	128.97	498.79	772.42	6,869.47
	Converge (runs)	3.03 (300)	3.05 (300)	262.54 (300)	166.60 (300)
AC/DC+FD	Aggregate	same as **AC/DC**			1,672.83
	Converge (runs)	there are no FDs			144.07 (300)
Speedup of **AC/DC+FD** over	libFM	152.70×	209.03×	80.34×	50.33×
	AC/DC	same as **AC/DC**, there are no FDs			3.87×

Table 1: Time performance (seconds) for learning LR, PR, and FaMa models over increasingly larger fragments (v_1 to v_4) of Retailer. (–) means that the system failed to compute due to design limitations. R can only compute the LR model for v_1, the other versions and models exceed the size limit of R's data frames. MADlib cannot compute any model on v_4 since the one-hot encoding requires more than 1600 columns; MADlib takes over 22 hours for PR. R and MADlib do not support FaMa models. TensorFlow learns LR models with batch size 1K and one epoch; it does not support PR and FaMa models. Since libFM requires a fixed number of runs, all experiments for FaMa$^8_2$ are computed with 300 runs.