Open access • Proceedings Article • DOI:10.1145/2723372.2723713

Learning Generalized Linear Models Over Normalized Data — Source link

Arun Kumar, Jeffrey F. Naughton, Jignesh M. Patel

Institutions: University of Wisconsin-Madison

Published on: 27 May 2015 - International Conference on Management of Data

Topics: Online machine learning, Active learning (machine learning), Semi-supervised learning, Feature engineering and Redundancy (engineering)

Learning Generalized Linear Models Over Normalized Data

Arun Kumar Jeffrey Naughton Jignesh M. Patel

Department of Computer Sciences,
University of Wisconsin-Madison
{arun, naughton, jignesh}@cs.wisc.edu

ABSTRACT

Enterprise data analytics is a booming area in the data management industry. Many companies are racing to develop toolkits that closely integrate statistical and machine learning techniques with data management systems. Almost all such toolkits assume that the input to a learning algorithm is a single table. However, most relational datasets are not stored as single tables due to normalization. Thus, analysts often perform key-foreign key joins before learning on the join output. This strategy of learning *after* joins introduces redundancy avoided by normalization, which could lead to poorer end-to-end performance and maintenance overheads due to data duplication. In this work, we take a step towards enabling and optimizing learning *over* joins for a common class of machine learning techniques called generalized linear models that are solved using gradient descent algorithms in an RDBMS setting. We present alternative approaches to learn over a join that are easy to implement over existing RDBMSs. We introduce a new approach named *factorized learning* that pushes ML computations through joins and avoids redundancy in both I/O and computations. We study the tradeoff space for all our approaches both analytically and empirically. Our results show that factorized learning is often substantially faster than the alternatives, but is not always the fastest, necessitating a cost-based approach. We also discuss extensions of all our approaches to multi-table joins as well as to Hive.

Categories and Subject Descriptors

H.2 [**Information Systems**]: Database Management

Keywords

Analytics; feature engineering; joins; machine learning

1. INTRODUCTION

There is an escalating arms race to bring sophisticated statistical and machine learning (ML) techniques to enterprise applications [3, 5]. A number of projects in both industry and academia aim to integrate ML capabilities with data processing in RDBMSs, Hadoop, and other systems [2, 4, 9, 15, 18, 21, 22, 33, 34]. Almost all such implementations of ML algorithms require that the input dataset be a single table. However, most relational datasets are not as stored single tables due to normalization [27]. Thus, analysts often perform key-foreign key joins of the base tables and *materialize* a single temporary table that is used as the input to the ML algorithm, i.e., they learn *after* joins.

Example: Consider an insurance company analyst modeling customer churn (will a customer leave the company or not) – a standard classification task. She builds a logistic regression model using the large table that stores customer details: `Customers(CustomerID, Churn, Age, Income, ..., EmployerID)`. Note that one of the features, `EmployerID`, is the ID of the customer's employer. It is a foreign key that refers to a separate table that stores details about companies and other organizations: `Employers(EmployerID, Revenue, NumEmployees, ...)`. She joins the two tables on the `EmployerID` as part of her "feature engineering" because she thinks the features of the employer might be helpful in predicting how likely a customer is to churn. For example, she might have a hunch that customers employed by large corporations are less likely to churn. She writes the output of the join as a single temporary table and feeds it to an ML toolkit that implements logistic regression.

Similar examples arise in a number of other application domains, e.g., detecting malicious users by joining data about user accounts with account activities, predicting census mail response rates by joining data about census districts with individual households, recommending products by joining data about past ratings with users and products, etc.

Learning after joins imposes an artificial barrier between the ML-based analysis and the base relations, resulting in several practical issues. First, the table obtained after the join can be much larger than the base tables themselves because the join introduces redundancy that was originally removed by database normalization [8, 27]. This results in unnecessary overheads for storage and performance as well as waste of time performing extra computations on data with redundancy. Second, as the base tables evolve, maintaining the materialized output of the join could become an overhead. Finally, analysts often perform exploratory analysis of different subsets of features and data [20, 33]. Materializing temporary tables after joins for learning on each subset could slow the analyst and inhibit exploration [7]. Learning *over* joins, i.e., pushing ML computations through joins to the base tables, mitigates such drawbacks.

SIGMOD'15, May 31 – June 04, 2015, Melbourne, VIC, Australia.
Copyright 2015 ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2723713.

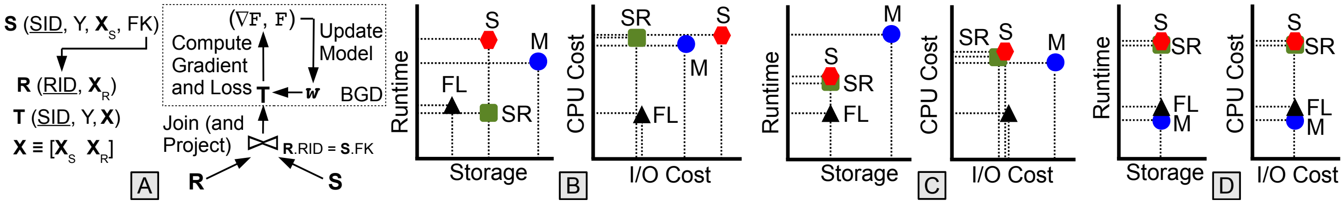


Figure 1: Learning over a join: (A) Schema and logical workflow. Feature vectors from S (e.g., Customers) and R (e.g., Employers) are concatenated and used for BGD. The loss (F) and gradient (∇F) for BGD can be computed together during a pass over the data. Approaches compared: Materialize (M), Stream (S), Stream-Reuse (SR), and Factorized Learning (FL). High-level qualitative comparison of storage-runtime tradeoffs and CPU-I/O cost tradeoffs for runtimes of the four approaches (S is assumed to be larger than R, and the plots are not to scale) (B) When the hash table on R does not fit in buffer memory, S, SR, and M require extra storage space for temporary tables or partitions. But, SR could be faster than FL due to lower I/O costs. (C) When the hash table on R fits in buffer memory, but S does not, SR becomes similar to S and neither need extra storage space, but both could be slower than FL. (D) When all data fit comfortably in buffer memory, none of the approaches need extra storage space, and M could be faster than FL.

From a technical perspective, the issues that arise from the redundancy present in a denormalized relation (used for learning after joins) are well known in the context of traditional relational data management [27]. But the implications of this type of redundancy in the context of ML algorithms are much less well understood. Thus, an important challenge to be addressed is if it is possible to devise approaches that learn over joins and avoid introducing such redundancy without sacrificing either the model quality, learning efficiency, or scalability compared to the currently standard approach of learning after joins.

As a first step, in this paper, we show that, for a large generic class of ML techniques called Generalized Linear Models (GLMs), it is possible to learn over joins and avoid redundancy without sacrificing quality and scalability, while actually improving performance. Furthermore, all our approaches to learn GLMs over joins are simple and easy to implement using existing RDBMS abstractions, which makes them more easily deployable than approaches that require deep changes to the code of an RDBMS. We focus on GLMs because they include many popular classification and regression techniques [17, 24]. We use standard gradient methods to learn GLMs: Batch Gradient Descent (BGD), Conjugate Gradient (CGD), and (L)BFGS [26]. For clarity of exposition, we use only BGD, but our results are also applicable to these other gradient methods. BGD is a numerical optimization algorithm that minimizes an objective function by performing multiple passes (*iterations*) over the data.

Figure 1(A) gives a high-level overview of our problem. We call the approach of materializing **T** before BGD as *Materialize*. We focus on the hybrid hash algorithm for the join operation [31]. We assume that **R** is smaller in size than **S** and estimate the I/O and CPU costs of all our approaches in a manner similar to [31]. We propose three alternative approaches to run BGD over a join in a single-node RDBMS setting – *Stream*, *Stream-Reuse* and *Factorized Learning*. Each approach avoids some forms of redundancy. *Stream* avoids writing **T** and could save on I/O. *Stream-Reuse* also exploits the fact that BGD is iterative and avoids repartitioning of the base relations after the first iteration. But, neither approach avoids redundancy in the computations for BGD. Thus, we design the *Factorized Learning* (in short, *Factorize*) approach that avoids computational redundancy as well. *Factorize* achieves this by interleaving the computations and I/O of the join operation and BGD. None of our approaches compromise on model quality. Furthermore, they are all easy to implement in an RDBMS using the abstraction of user-defined aggregate functions (UDAFs), which provides scalability and ease of deployment [13, 16].

The performance picture, however, is more complex. Figures 1(B-D) give a high-level qualitative overview of the tradeoff space for all our approaches in terms of the storage space needed and the runtimes (split into I/O and CPU costs). Both our analytical and experimental results show that *Factorize* is often the fastest approach, but which approach is the fastest depends on a combination of factors such as buffer memory, input table dimensions, and number of iterations. Thus, a cost model such as ours is required to select the fastest approach for a given instance of our problem. Furthermore, we identify that *Factorize* might face a scalability bottleneck since it maintains an aggregation state whose size is linear in the number of tuples in **R**. We propose three extensions to mitigate this bottleneck and find that none of them dominate the others in terms of runtime, which again necessitates our cost model.

We extend all our approaches to multi-table joins, specifically, the case in which **S** has multiple foreign keys. Such a scenario arises in applications such as recommendation systems in which a table of ratings refers to both the user and product tables [28]. We show that optimally extending *Factorize* to multi-table joins involves solving a problem that is NP-Hard. We propose a simple, but effective, greedy heuristic to tackle this problem. Finally, we extend all our approaches to the shared-nothing parallel setting and implement them on Hive. We find near-linear speedups and scaleups for all our approaches.

In summary, our work makes the following contributions:

- To the best of our knowledge, this is the first paper to study the problem of learning over joins of large relations without materializing the join output. Focusing on GLMs solved using BGD, we explain the tradeoff space in terms of I/O and CPU costs and propose alternative approaches to learn over joins.
- We propose the *Factorize* approach that pushes BGD computations through a join, while being amenable to a simple implementation in existing RDBMSs.

ML Technique	F_e (a, b) (For Loss)	G (a, b) (For Gradient)
Logistic Regression (**LR**)	$\log(1 + e^{-ab})$	$\frac{-a}{1 + e^{ab}}$
Least-Squares Regression (**LSR**), Lasso, and Ridge	$(a - b)^2$	$2(b - a)$
Linear Support Vector Machine (**LSVM**)	$\max\{0, 1 - ab\}$	$-a\delta_{ab<1}$

Table 1: GLMs and their functions.

- We compare the performance of all our approaches both analytically and empirically using implementations on PostgreSQL. Our results show that *Factorize* is often, but not always, the fastest approach. A combination of factors such as the buffer memory, the dimensions of the input tables, and the number of BGD iterations determines which approach is the fastest. We also validate the accuracy of our analytical models.
- We extend all our approaches to multi-table joins. We also demonstrate how to parallelize them using implementations on Hive.

Outline. In Section 2, we present a brief background on GLMs and BGD and some preliminaries for our problem. In Section 3, we explain our cost model and simple approaches to learn over joins. In Section 4, we present the new approach of *Factorized Learning* and its extensions. In Section 5, we discuss our experimental setup and results. We discuss related work in Section 6 and conclude in Section 7.

2. BACKGROUND AND PRELIMINARIES

We provide a brief introduction to GLMs and BGD. For a deeper description, we refer the reader to [17, 24, 26].

Generalized Linear Models (GLMs). Consider a dataset of n examples, each of which includes a d-dimensional numeric *feature vector*, $\mathbf{x}_i$, and a numeric *target*, y_i ($i = 1$ to n). For regression, $y_i \in \mathbb{R}$, while for (binary) classification, $y_i \in \{-1, 1\}$. Loosely, GLMs assume that the data points can be separated into its target classes (for classification), or approximated (for regression), by a hyperplane. The idea is to compute such a hyperplane $\mathbf{w} \in \mathbb{R}^d$ by defining an optimization problem using the given dataset. We are given a *linearly-separable* objective function that computes the *loss* of a given model $\mathbf{w} \in \mathbb{R}^d$ on the data: $F(\mathbf{w}) = \sum_{i=1}^{n} F_e(y_i, \mathbf{w}^T\mathbf{x}_i)$. The goal of an ML algorithm is to minimize the loss function, i.e., find a vector $\mathbf{w}^* \in \mathbb{R}^d$, s.t., $\mathbf{w}^* = \arg\min_{\mathbf{w}} F(\mathbf{w})$. Table 1 lists examples of some popular GLM techniques and their respective loss functions. The loss functions of GLMs are *convex* (bowl-shaped), which means any local minimum is a global minimum, and standard gradient descent algorithms can be used to solve them.[1]

Batch Gradient Descent (BGD). BGD is a simple algorithm to solve GLMs using iterative numerical optimization. BGD initializes the model $\mathbf{w}$ to some $\mathbf{w}_0$, computes the gradient $\nabla F(\mathbf{w})$ on the given dataset, and updates the model as $\mathbf{w} \leftarrow \mathbf{w} - \alpha\nabla F(\mathbf{w})$, where $\alpha > 0$ is the stepsize parameter. The method is outlined in Algorithm 1. Like F, the gradient is also linearly separable: $\nabla F(\mathbf{w}) = \sum_{i=1}^{n} G(y_i, \mathbf{w}^T\mathbf{x}_i)\mathbf{x}_i$. Since the gradient is the direction of steepest ascent of F, BGD is also known as the method of steepest descent [26]. Table 1 also lists the gradient functions of the GLMs. We shall use F and $F(\mathbf{w})$ interchangeably.

```
Algorithm 1 Batch Gradient Descent (BGD)
Inputs: ${\mathbf{x}_i, y_i}_{i=1}^n$ (Data), $\mathbf{w}_0$ (Initial model)
1: $k \leftarrow 0$, $r_{prev} \leftarrow$ null, $r_{curr} \leftarrow$ null, $\mathbf{g}_k \leftarrow$ null
2: while (Stop ($k$, $r_{prev}$, $r_{curr}$, $\mathbf{g}_k$) = False) do
3:     $r_{prev} \leftarrow r_{curr}$
4:     $(\mathbf{g}_k,\ r_{curr}) \leftarrow (\nabla F_{k+1},\ F_{k+1})$      ▷ 1 pass over data
5:     $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k - \alpha_k \mathbf{g}_k$      ▷ Pick $\alpha_k$ by line search
6:     $k \leftarrow k + 1$
7: end while
```

BGD updates the model repeatedly, i.e., over many *iterations* (or *epochs*), each of which requires (at least) one pass over the data. The loss value typically drops over iterations. The algorithm is typically stopped after a pre-defined number of iterations, or when it *converges* (e.g., the drop in the loss value across iterations, or the norm of the gradient, falls below a given threshold). The stepsize parameter (α) is typically tuned using a line search method that potentially computes the loss many times (similar to step 4) [26].

On large data, it is likely that computing F and ∇F dominates the runtime of BGD [12, 13]. Fortunately, both F and ∇F can be computed scalably in a manner similar to distributive aggregates like `SUM` in SQL. Thus, it is easy to implement BGD using the abstraction of a user-defined aggregate function (UDAF) that is available in almost all RDBMSs [13, 16]. However, unlike `SUM`, BGD performs a "multi-column" or vector aggregation since all feature values of an example are needed to compute its contribution to the gradient. For simplicity of exposition, we assume that feature vectors are instead stored as arrays in a single column.

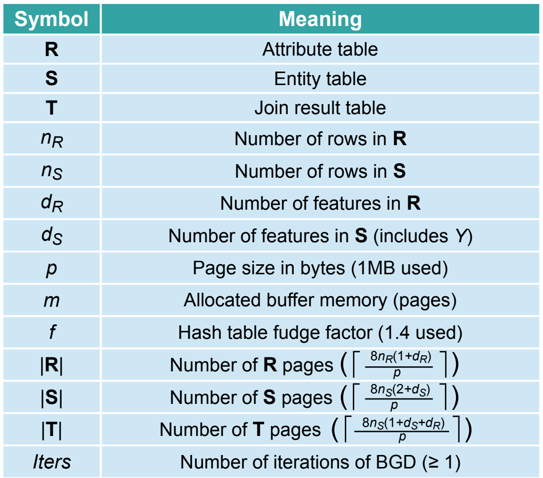
Joins Before Learning. From our conversations with analysts at companies across various domains – insurance, consulting, Web search, security, and e-commerce – we have learned that analysts often perform joins to replace foreign key references with actual feature values as part of their feature engineering effort.[2] In this work, we focus chiefly on a two-table join. We term the main table with the entities to learn on as the *entity table* (denoted $\mathbf{S}$). We term the other table as the *attribute table* (denoted $\mathbf{R}$). A column in $\mathbf{S}$ is a foreign key that refers to $\mathbf{R}$.

Problem Statement. Suppose there are n_S examples (tuples) in $\mathbf{S}$, and n_R tuples in $\mathbf{R}$. Assume that the feature vectors are split across $\mathbf{S}$ and $\mathbf{R}$, with $d_S - 1$ features in $\mathbf{X}_S$ and $d_R = d - d_S + 1$ in $\mathbf{X}_R$. Thus, the "width" of $\mathbf{S}$ is $2 + d_S$, including the ID, foreign key, and target. The width of $\mathbf{R}$ is $1 + d_R$, including the ID. Typically, we have $n_S \gg n_R$, similar to how fact tables have more tuples than dimension tables in OLAP [16, 27]. We now state our problem formally (illustrated in Figure 1(A)).

Given two relations $\mathbf{S}$ $(\underline{SID}, Y, \mathbf{X}_S, FK)$ *and* $\mathbf{R}$ $(\underline{RID}, \mathbf{X}_R)$ *with a key-foreign key relationship* ($\mathbf{S}.FK$ *refers to* $\mathbf{R}.RID$), *where* $\mathbf{X}_S$ *and* $\mathbf{X}_R$ *are feature vectors and* Y *is the target, learn a GLM using BGD over the result of the projected*

[1] Typically, a convex penalty term called a *regularizer* is added to the loss to constrain $\|\mathbf{w}\|$ [17].

[2] An alternative is to simply ignore the foreign key, or treat it as a large, sparse categorical feature. Such feature engineering judgements are largely analyst-specific [7, 20, 33]. Our work simply aims to make feature engineering easier.



Symbol	Meaning
R	Attribute table
S	Entity table
T	Join result table
n_R	Number of rows in **R**
n_S	Number of rows in **S**
d_R	Number of features in **R**
d_S	Number of features in **S** (includes *Y*)
p	Page size in bytes (1MB used)
m	Allocated buffer memory (pages)
f	Hash table fudge factor (1.4 used)
|**R**|	Number of **R** pages ($\lceil \frac{8n_R(1+d_R)}{p} \rceil$)
|**S**|	Number of **S** pages ($\lceil \frac{8n_S(2+d_S)}{p} \rceil$)
|**T**|	Number of **T** pages ($\lceil \frac{8n_S(1+d_S+d_R)}{p} \rceil$)
Iters	Number of iterations of BGD (≥ 1)

Table 2: Notation for objects and parameters used in the cost models. I/O costs are counted in number of pages. Dividing by the disk throughput yields the estimated runtimes. NB: As a simplifying assumption, we use an 8B representation for all values: IDs, target, and features (categorical features are assumed be have been converted to numeric ones [17]).

equi-join $\mathbf{T}(\underline{SID}, Y, [\mathbf{X}_S\ \ \mathbf{X}_R]) \leftarrow \pi(\mathbf{R} \bowtie_{RID=FK} \mathbf{S})$ *such that the feature vector of a tuple in* **T** *is the concatenation of the feature vectors from the joining tuples of* **S** *and* **R**.

3. LEARNING OVER JOINS

We now discuss alternative approaches to run BGD over a table that is logically the output of a key-foreign key join.

3.1 Assumptions and Cost Model

For the rest of the paper, we focus only on the data-intensive computation in step 4 of Algorithm 1 – computing $(\nabla F,\ F)$. The data-agnostic computations of updating $\mathbf{w}$ are identical across all approaches proposed here, and typically take only a few seconds.[3] Tables 2 and 3 summarize our notation for the objects and parameters.

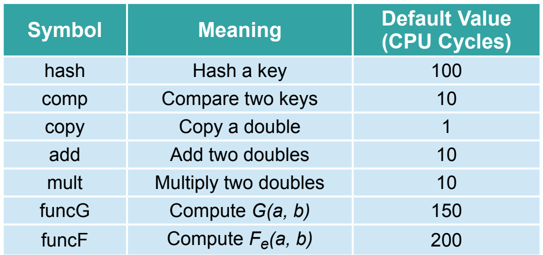
We focus on the classical hybrid hash join algorithm (considering other join algorithms is part of future work), which requires $(m-1) > \sqrt{\lceil f|\mathbf{R}| \rceil}$ [31]. We also focus primarily on the case $n_S > n_R$ and $|\mathbf{S}| \geq |\mathbf{R}|$. We discuss the cases $n_S \leq n_R$ or $|\mathbf{S}| < |\mathbf{R}|$ in the appendix.

3.2 BGD After a Join: Materialize (M)

Materialize (M) is the current popular approach for handling ML over normalized datasets. Essentially, we write a new table and use it for BGD.

1. Apply hybrid hash join to obtain and write **T**.
2. Read **T** to compute $(\nabla F, F)$ for each iteration.

Following the style of the discussion of the hybrid hash join algorithm in [31], we now introduce some notation. The number of partitions of **R** is $B = \left\lceil \frac{\lceil f|\mathbf{R}| \rceil - (m-2)}{(m-2)-1} \right\rceil$. Partition sizes are $|\mathbf{R}_0| = \left\lfloor \frac{(m-2)-B}{f} \right\rfloor$, and $|\mathbf{R}_i| = \left\lceil \frac{|\mathbf{R}|-|\mathbf{R}_0|}{B} \right\rceil (1 \leq i \leq B)$, with the ratio $q = \frac{|\mathbf{R}_0|}{|\mathbf{R}|}$, where $\mathbf{R}_0$ is the first partition and $\mathbf{R}_i$ are the other partitions as per the hybrid hash join algorithm [31]. We provide the detailed I/O and CPU costs of Materialize here. The costs of the other approaches in this section can be derived from these, but due to space constraints, we present them in the appendix.



Symbol	Meaning	Default Value (CPU Cycles)
hash	Hash a key	100
comp	Compare two keys	10
copy	Copy a double	1
add	Add two doubles	10
mult	Multiply two doubles	10
funcG	Compute *G(a, b)*	150
funcF	Compute F_e*(a, b)*	200

Table 3: Notation for the CPU cost model. The approximate default values for CPU cycles for each unit of the cost model were estimated empirically on the machine on which the experiments were run. Dividing by the CPU clock frequency yields the estimated runtimes. For G and F_e, we assume LR. LSR and LSVM are slightly faster.

I/O Cost. If $(m-1) \leq \lceil f|\mathbf{R}| \rceil$, we partition the tables:

```
  (|R|+|S|)          //First read
+ 2.(|R|+|S|).(1-q)  //Write, read temp partitions
+ |T|                //Write output
+ |T|                //Read for first iteration
+ (Iters-1).|T|      //Remaining iterations
- min{|T|,[(m-2)-f|R_i|]} //Cache T for iter 1
- min{|T|,(m-1)}.(Iters-1)  //MRU for rest
```

If $(m-1) > \lceil f|\mathbf{R}| \rceil$, we need not partition the tables:

```
  (|R|+|S|)
+ (Iters+1).|T|
- min{|T|,[(m-2)-f|R|]}
- min{|T|,(m-1)}.(Iters-1)
```

CPU Cost

```
  (nR+nS).hash          //Partition R and S
+ nR.(1+dR).copy        //Construct hash on R
+ nR.(1+dR).(1-q).copy  //R output partitions
+ nS.(2+dS).(1-q).copy  //S output partitions
+ (nR+nS).(1-q).hash    //Hash on R and S partitions
+ nS.comp.f             //Probe for all of S
+ nS.(1+dS+dR).copy     //T output partitions
+ Iters.[               //Compute gradient and loss
    nS.d.(mult+add)     //w.xi for all i
  + nS.(funcG+funcF)    //Apply G and F_e
  + nS.d.(mult+add)     //Scale and add
  + nS.add              //Add for total loss
  ]
```

3.3 BGD Over a Join: Stream (S)

This approach performs the join *lazily* for each iteration.

1. Apply hybrid hash join to obtain **T**, but instead of writing **T**, compute $(\nabla F, F)$ on the fly.
2. Repeat step 1 for each iteration.

[3] CGD and (L)BFGS differ from BGD only in these data-agnostic computations, which are easily implemented in, say, Python, or R [12]. If a line search is used to tune α, we need to compute only F, but largely the same tradeoffs apply.

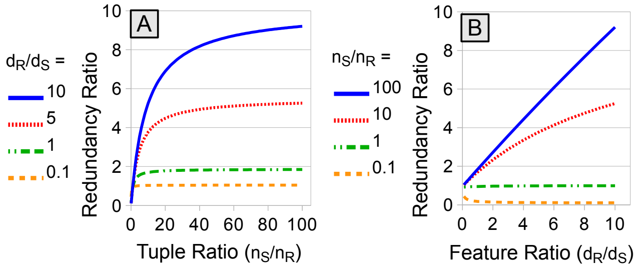


Figure 2: Redundancy ratio against the two dimension ratios (for $d_S = 20$). (A) Fix $\frac{d_R}{d_S}$ and vary $\frac{n_S}{n_R}$. (B) Fix $\frac{n_S}{n_R}$ and vary $\frac{d_R}{d_S}$.

The I/O cost of Stream is simply the cost of the hybrid hash join multiplied by the number of iterations. Its CPU cost is a combination of the join and BGD.

Discussion of Tradeoffs. The I/O and storage tradeoffs between Materialize and Stream (Figure 1(B)) arise because it is likely that many tuples of **S** join with a single tuple of **R** (e.g., many customers might have the same employer). Thus, $|\mathbf{T}|$ is usually larger than $|\mathbf{S}|+|\mathbf{R}|$. Obviously, the gap depends upon the dataset sizes. More precisely, we define the *redundancy ratio (r)* as the ratio of the size of **T** to that of **S** and **R**:

$$r = \frac{n_S(1 + d_S + d_R)}{n_S(2 + d_S) + n_R(1 + d_R)} = \frac{\frac{n_S}{n_R}(1 + \frac{d_R}{d_S} + \frac{1}{d_S})}{\frac{n_S}{n_R}(1 + \frac{2}{d_S}) + \frac{d_R}{d_S} + \frac{1}{d_S}}$$

This ratio is useful because it gives us an idea of the factor of speedups that are potentially possible by learning over joins. Since it depends on the dimensions of the inputs, we plot the redundancy ratio for different values of the *tuple ratio* ($\frac{n_S}{n_R}$) and (inverse) *feature ratio* ($\frac{d_R}{d_S}$), while fixing d_S. Figure 2 presents the plots. Typically, both dimension ratios are > 1, which mostly yields $r > 1$. But when the tuple ratio is < 1, $r < 1$ (see Figure 2(A)). This is because the join here becomes selective (when $n_S < n_R$). However, when the tuple ratio > 1, we see that r increases with the tuple ratio. It converges to $\frac{1+\frac{d_R}{d_S}+\frac{1}{d_S}}{1+\frac{2}{d_S}} \approx 1 + \frac{d_R}{d_S}$. Similarly, as shown in Figure 2(B), the redundancy ratio increases with the feature ratio, and converges to the tuple ratio $\frac{n_S}{n_R}$.

3.4 An Improvement: Stream-Reuse (SR)

We now present a simple modification to Stream – the Stream-Reuse approach – that can significantly improve performance.

1. Apply hybrid hash join to obtain **T**, but instead of writing **T**, run the first iteration of BGD on the fly.
2. Maintain the temporary partitions of **S** and **R** on disk.
3. For the remaining iterations, reuse the partitions of **S** and and **R** for the hybrid hash join, similar to step 1.

The I/O cost of Stream-Reuse gets rid of the rewriting (and rereading) of partitions at every iteration, but the CPU cost is reduced only slightly. Stream-Reuse makes the join "iteration-aware" – we need to divide the implementation of the hybrid hash join in to two steps so as to reuse the partitions across iterations. An easier way to implement (without changing the RDBMS code) is to manually handle pre-partitioning at the logical query layer after consulting the optimizer about the number of partitions. Although the latter is a minor approximation to SR, the difference in performance (estimated using our analytical cost models) is mostly negligible.

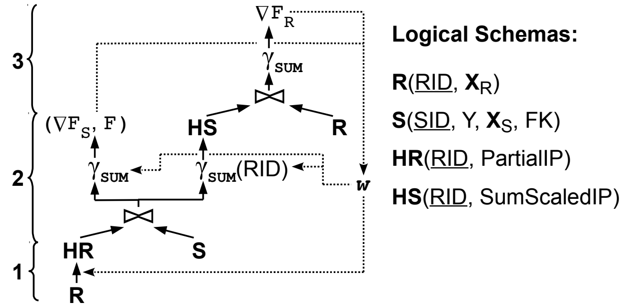


Figure 3: Logical workflow of factorized learning, consisting of three steps as numbered. HR and HS are logical intermediate relations. PartialIP refers to the partial inner products from R. SumScaledIP refers to the grouped sums of the scalar output of $G()$ applied to the full inner products on the concatenated feature vectors. Here, γ_{SUM} denotes a SUM aggregation and $\gamma_{SUM}(RID)$ denotes a SUM aggregation with a GROUP BY on RID.

4. FACTORIZED LEARNING

We now present a new technique that interleaves the I/O and CPU processing of the join and BGD. The basic idea is to avoid the redundancy introduced by the join by dividing the computations of both F and ∇F and "pushing them through the join". We call our technique *factorized learning* (Factorize, or FL for short), borrowing the terminology from "factorized" databases [8]. An overview of the logical computations in FL is presented in Figure 3.

The key insight in FL is as follows: given a feature vector $\mathbf{x} \in \mathbf{T}$, we have $\mathbf{w}^T\mathbf{x} = \mathbf{w}_S^T\mathbf{x}_S + \mathbf{w}_R^T\mathbf{x}_R$. Since the join duplicates $\mathbf{x}_R$ from **R** when constructing **T**, the main goal of FL is to avoid redundant inner product computations as well as I/O over those feature vectors from **R**. FL achieves this goal with the following three steps (numbered in Figure 3).

1. Compute and save the partial inner products $\mathbf{w}_R^T\mathbf{x}_R$ for each tuple in **R** in a new table **HR** under the PartialIP column (part 1 in Figure 3).
2. Recall that the computation of F and ∇F are clubbed together, and that $\nabla F \equiv [\nabla F_S \ \nabla F_R]$. This step computes F and ∇F_S together. Essentially, we join **HR** and **S** on RID and complete the computation of the full inner products on the fly, and follow that up by applying both $F_e()$ and $G()$ on each example. By aggregating both these quantities as it performs the join, FL completes the computation of $F = \sum F_e(y, \mathbf{w}^T\mathbf{x})$ and $\nabla F_S = \sum G(y, \mathbf{w}^T\mathbf{x})\mathbf{x}_S$. Simultaneously, FL also performs a GROUP BY on RID and sums up $G(y, \mathbf{w}^T\mathbf{x})$, which is saved in a new table **HS** under the SumScaledIP column (part 2 in Figure 3).
3. Compute $\nabla F_R = \sum G(y, \mathbf{w}^T\mathbf{x})\mathbf{x}_R$ by joining **HS** with **R** on RID and scaling the partial feature vectors $\mathbf{x}_R$ with SumScaledIP (part 3 in Figure 3).

Example: Consider logistic regression (LR). In step 2, as the full inner product $\mathbf{w}^T\mathbf{x}$ is computed by joining **HR** and

S, FL computes $log(1 + exp(-y\mathbf{w}^T\mathbf{x}))$ and adds it into F. Immediately, FL also computes $\frac{-y}{1+exp(y\mathbf{w}^T\mathbf{x})} = g$ (say), and adds it into SumScaledIP for that RID. It also computes $g\mathbf{x}_S$ and adds it into ∇F_S.

Overall, FL computes $(\nabla F, F)$ without any redundancy in the computations. FL reduces the CPU cost of floating point operations for computing inner products from $O(n_S(d_S + d_R))$ to $O(n_S d_S + n_R d_R)$. The reduction roughly equals the redundancy ratio (Section 3.3). Once $(\nabla F, F)$ is computed, **w** is updated and the whole process is repeated for the next iteration of BGD. Note that to compute only F, step 3 of FL can be skipped. FL gives the same results as Materialize and Stream. We provide the proof in the appendix.

PROPOSITION 4.1. *The output* $(\nabla F, F)$ *of FL is identical to the output* $(\nabla F, F)$ *of both Materialize and Stream.*[4]

While it is straightforward to translate the logical plan shown in Figure 3 into SQL queries, we implement it using a slightly different scheme. Our goal is to take advantage of some physical properties of this problem that will help us improve performance by avoiding some table management overheads imposed by the RDBMS. Basically, since **HR** and **HS** are both small 2-column tables keyed by RID, we maintain them together in a simple in-memory associative array **H** with the bucket for each RID being a pair of double precision numbers (for PartialIP and SumScaledIP). Thus, we perform random reads and writes on **H**, and replace both the joins (from parts 2 and 3 in Figure 3) with simple aggregation queries with *user-defined aggregation functions* (UDAFs) that perform simple scans over the base tables. Overall, our implementation of FL works as follows, with each step corresponding to its respective part in Figure 3:

1. Read **R**, hash on RID and construct **H** in memory with partial inner products ($\mathbf{w}_R^T\mathbf{x}_R$) saved in PartialIP. It can be expressed as the following SQL query: `SELECT UDAF1(R.RID,R.xR,wR) FROM R`.
2. Read **S**, probe into **H** using the condition $FK = RID$, complete the inner products by adding PartialIP from **H** with partial inner products on each tuple ($\mathbf{w}_S^T\mathbf{x}_S$), update F in the aggregation state by adding into it (essentially, a `SUM`), update ∇F_S in the aggregation state by adding into it (a `SUM` over vectors), and add the value of $G(\mathbf{w}^T\mathbf{x})$ into SumScaledIP (essentially, a `SUM` with a `GROUP BY` on RID). As a query: `SELECT UDAF2(S.FK,S.y,S.xS,wS) FROM S`.
3. Read **R**, probe into **H** on RID, compute partial gradients on each example and update ∇F_R in memory (essentially, a `SUM` over vectors). As a query: `SELECT UDAF3(R.RID,R.xR) FROM R`.
4. Repeat steps 1-3 for each remaining iteration.

I/O Cost

```
Iters.[
    (|R|+|S|+|R|)          //Read for each iter
  - min{|R|,(m-1)-|H|} ]  //Cache R for second pass
- (Iters - 1).[
    min{|R|,(m-1)-|H|}    //Cache R for next iter
  + min{|S|,max{0,(m-1)-|H|-|R|}} ] //Cache S too
```

CPU Cost

```
Iters.[
    nR.hash              //Hash R for stats
  + nR.dR.(mult+add)     //Partial w.xi for col 1
  + nR.copy              //Update column 1 of H
  + nS.(hash+comp.f)     //Probe for all of S
  + nS.(dS-1).(mult+add) //Partial w.xi for col 2
  + nS.(add+funcG+funcF) //Full w.xi and functions
  + nS.(dS-1).(mult+add) //Partial scale and add
  + nS.add               //Add for total loss
  + (nS-nR).add          //Compute column 2 of H
  + nR.copy              //Update column 2 of H
  + nR.(hash+comp.f)     //Probe for all of R
  + nR.dR.(mult+add)     //Partial scale and add
  ]
```

The above costs present an interesting insight into FL. While it avoids computational redundancy in computing $(\nabla F, F)$, FL performs extra computations to manage **H**. Thus, FL introduces a non-obvious computational tradeoff. Similarly, FL requires an extra scan of **R** per iteration, which introduces a non-obvious I/O tradeoff. As we will show later in Section 5, these tradeoffs determine which approach will be fastest on a given instance of the problem.

As an implementation detail, we found that, while it is easy to manage **H** as an associative array, the caching of **R** for the second pass is not straightforward to implement. This is because the pages of **R** might be evicted when **S** is read, unless we manually keep them in memory. We found that the performance benefit due to this is usually under 10% and thus, we ignore it. But if an RDBMS offers an easy way to "pin" pages of a table to buffer memory, we can use it in FL.

Finally, we note that FL requires **H** to be maintained in buffer memory, which requires $m - 1 > |\mathbf{H}|$. But note that $|\mathbf{H}| = \left\lceil \frac{f.n_R.(1+2).8}{p} \right\rceil$, which is only $O(n_R)$. Thus, in many cases, **H** will probably easily fit in buffer memory. Nevertheless, to handle cases when **H** does not fit in buffer memory, we present a few extensions to FL.

4.1 Scaling FL along n_R

We explore three extensions to FL to mitigate its scalability bottleneck: FLSQL, FLSQL+, and FL-Partition (FLP).

4.1.1 FLSQL

FLSQL applies the traditional approach of optimizing SQL aggregates (e.g., `SUM`) over joins using logical query rewriting [10,32]. Instead of maintaining **H** in memory, it directly converts the logical plan of Figure 3 into SQL queries by managing **HR** and **HS** as separate tables:

1. Read **R**, and write **HR** with partial inner products.
2. Join **HR** and **S** and aggregate (`SUM`) the result to compute $(\nabla F_S, F)$.
3. Join **HR** and **S** and write **HS** after a `GROUP BY` on RID. **HS** contains sums of scaled inner products.
4. Join **HS** and **R** and aggregate (`SUM`) the result to compute ∇F_R.
5. Repeat steps 1-4 for each remaining iteration.

Note that both **HR** and **HS** are of size $O(n_R)$. Also, note that we have to read **S** twice – once for computing an aggregate and the other for creating a new table.

[4] The proof assumes exact arithmetic. Finite-precision arithmetic may introduce minor errors. We leave a numerical analysis of FL to future work.

4.1.2 FLSQL+

FLSQL+ uses the observation that since $n_S \gg n_R$ typically (and perhaps $d_S < d_R$), it might be faster to write a wider table in step 2 of FLSQL instead of reading $\mathbf{S}$ twice:

1. Read $\mathbf{R}$, and write $\mathbf{HR}$ with partial inner products.
2. Join $\mathbf{HR}$ and $\mathbf{S}$ and write $\mathbf{HS}+$ after a `GROUP BY` on RID. $\mathbf{HS}+$ contains both sums of scaled inner products and partial gradient vectors.
3. Join $\mathbf{HS}+$ and $\mathbf{R}$ and aggregate (`SUM`) the result to compute F and ∇F.
4. Repeat steps 1-3 for each iteration..

Note that $\mathbf{HR}$ is of size $O(n_R)$ but $\mathbf{HS}+$ is of size $O(n_R d_S)$, since it includes the partial gradients too. Thus, whether this is faster or not depends on the dataset dimensions.

4.1.3 FL-Partition (FLP)

The basic idea behind FLP is simple – pre-partition $\mathbf{R}$ and $\mathbf{S}$ so that the smaller associative arrays can fit in memory:

1. Partition $\mathbf{R}$ and $\mathbf{S}$ on RID (resp. foreign key) into $\{\mathbf{R}_i\}$ and $\{\mathbf{S}_i\}$ so that each $\mathbf{H}_i$ corresponding to $\mathbf{R}_i$ fits in memory.
2. For each pair $\mathbf{R}_i$ and $\mathbf{S}_i$, apply FL to obtain partial $(\nabla F, F)_i$. Add the results from all partitions to obtain full $(\nabla F, F)$.
3. Repeat steps 1-2 for each remaining iteration, reusing the partitions of $\mathbf{R}$ and $\mathbf{S}$, similar to Stream-Reuse.

Note that we could even partition $\mathbf{S}$ and $\mathbf{R}$ into more than necessary partitions to ensure that $\mathbf{R}_i$ is cached for the second pass, thus improving the performance slightly. All the above extensions preserve the correctness guarantee of FL. We provide the proof in the appendix.

Proposition 4.2. *The output $(\nabla F, F)$ of FLP, FLSQL, and FLSQL+ are all identical to the output $(\nabla F, F)$ of FL.*

4.2 Extensions

We explain how we can extend our approaches to multi-table joins. We then briefly discuss how we can extend our approaches to a shared-nothing parallel setting.

4.2.1 Multi-table Joins

Multi-table key-foreign key joins do arise in some applications of ML. For example, in a movie recommendation system such as Netflix, ratings of movies by users (e.g., 1-5 stars) are typically stored in a table that has foreign key references to two tables – one with user details, and another with movie details. Thus, there is one entity table and many attribute tables (considering other schema scenarios is part of future work). Extending Materialize and Stream (and Stream-Reuse) to multi-table joins is trivial, since data processing systems such as RDBMSs and Hive already support and optimize multi-table joins [1, 29].

Extending FL is also straightforward, provided we have enough memory to store the associative arrays of all attribute relations simultaneously for step 2. But we face a technical challenge when the memory is insufficient. One solution is to adapt the FLP strategy, but partitioning all input relations might be an overkill. Instead, it is possible to improve performance by formulating a standard discrete optimization problem to determine the subset of input relations to partition so that the overall runtime is minimized.

Formally, we are given k attribute tables $\mathbf{R}_i(\underline{RID_i}, \mathbf{X}_i)$, $i = 1$ to k, and the entity table $\mathbf{S}(\underline{SID}, Y, \mathbf{X}_S, FK_1, \ldots, FK_k)$, with k foreign keys. Our approach reads each $\mathbf{R}_i$ and converts it to its associative array $\mathbf{HR}_i$ (step 1 of FL), and then applies a simplified GRACE hash join [31] recursively on the right-deep join tree with $\mathbf{S}$ as the outer table.[5] We have $m < \sum_{i=1}^{k} |\mathbf{HR}_i|$, and thus, we need to partition $\mathbf{S}$ and some (or all) of $\{\mathbf{HR}_i\}$. Let s_i (a positive integer) be the number of partitions of $\mathbf{HR}_i$ (so, $\mathbf{S}$ has $\Pi_{i=1}^{k} s_i$ partitions). We now make three observations. First, minimizing the total cost of partitioning is equivalent to maximizing the total savings from not partitioning. Second, the cost of partitioning $\mathbf{R}_i$, viz., $2|\mathbf{R}_i|$, is independent of s_i, provided the page size p is large enough to perform mostly sequential writes (note that $s_i \leq |\mathbf{R}_i|$). Thus, it only matters if $s_i = 1$ or not. Define a binary variable x_i as $x_i = 1$, if $s_i = 1$, and $x_i = 0$, if $s_i > 1$. Finally, we allocate at least one page of memory for each $\mathbf{R}_i$ to process each partition of $\mathbf{S}$ (this needs $m > k$, which is typically not an issue). We now formally state the problem (FL-MULTJOIN) as follows:

$$\max \sum_{i=1}^{k} x_i |\mathbf{R}_i|, \ \ s.t. \ \sum_{i=1}^{k} x_i (|\mathbf{HR}_i| - 1) \leq m - 1 - k$$

Basically, we count the I/Os saved for those $\mathbf{R}_i$ that do not need to be partitioned since $\mathbf{HR}_i$ fits entirely in memory. We prove the following result:

Theorem 4.1. *FL-MULTJOIN is* NP*-Hard in l, where $l = |\{i | m - k \geq |\mathbf{HR}_i| > 1\}| \leq k$.*

Essentially, this result means that FL-MULTJOIN is trivial if either none of $\mathbf{HR}_i$ fit in memory individually or all fit simultaneously, but is harder if a few (not all) can fit simultaneously. Our proof provides a reduction from the classical 0/1 knapsack problem [14]. Due to space constraints, we present the proof in the appendix. We adopt a standard $O(klog(k))$ time greedy heuristic for the knapsack problem to solve FL-MULTJOIN approximately [11]. Essentially, we sort the list of attribute tables on $\frac{|\mathbf{R}_i|}{(|\mathbf{HR}_i|-1)}$, and pick the associative arrays to fit in memory in decreasing order of that ratio until we run out of memory. We leave more sophisticated heuristics to future work.

4.2.2 Shared-nothing Parallelism

It is trivial to parallelize Materialize and Stream (and Stream-Reuse) since most parallel data processing systems such as parallel RDBMSs and Hive already provide parallel joins. The only requirement is that the aggregations needed for BGD need to be *algebraic* [16]. But since both F and ∇F are just sums across tuples, they are indeed algebraic. Hence, by implementing them using the parallel user-defined aggregate function abstraction provided by Hive [1] (and most parallel RDBMSs), we can directly leverage existing parallelism infrastructure [13].

FL needs a bit more attention. All three of its steps can also be implemented using parallel UDAFs, But merging independent copies of $\mathbf{H}$ requires reinserting the individual entries, rather than just summing them up. Also, since $\mathbf{H}$ is $O(n_R)$ in size, we might exceed available memory when multiple aggregation states are merged. While this issue might

[5] Hybrid hash requires a more complex analysis and we leave it for future work. But note that GRACE and hybrid hash have similar performance in low memory settings [31].

be resolved in Hadoop automatically by spilling to disk, a faster alternative might be to employ the FLP strategy – by using the degree of parallelism and available memory on each node, we pre-partition the inputs and process each partition in parallel separately. Of course, another alternative is to employ FLSQL or FLSQL+, and leave it to Hive to manage the intermediate tables that correspond to **H** in FL. While these strategies might be slightly slower than FLP, they are much easier to implement. We leave a detailed comparison of alternatives that take communication costs into consideration to future work.

5. EXPERIMENTS

We present empirical and analytical results comparing the performance all our approaches to learn over joins against Materialize. Our goal in this section is three-fold: (1) Get a high-level picture of the tradeoff space using our analytical cost models. (2) Drill down deeper into the relative performance of various approaches using our implementations and also validate the accuracy of our cost models. (3) Evaluate the efficiency and effectiveness of each of our extensions.

Data. Unfortunately, publicly-available large real (labeled) datasets for ML tasks are rare [6]. And to the best of our knowledge, there is no publicly-available large real database with the key-foreign key relationship we study. Nevertheless, since this paper focuses on performance at scale, synthetic datasets are a reasonable option, and we use this approach. The ranges of dimensions for our datasets are modeled on the real datasets that we found in practice. Our synthesizer samples examples based on a random class boundary (for binary classification with LR) and adds some random noise. The codes for our synthesizer and all our implementations are available for download from our project webpage.[6]

5.1 High-level Performance Picture

We compare the end-to-end performance of all approaches in a single-node RDBMS setting. Using our analytical cost models, we vary the buffer memory and plot the I/O, CPU, and total runtimes of all approaches to get a high-level picture of the tradeoff space. Figure 4 presents the results.

The plots show interesting high-level differences in the behavior of all the approaches. To help explain their behavior, we classify memory into three major regions: the hash table on **R** does not fit in memory (we call this the *relative-small-memory*, or RSM region), the hash table on **R** does fit in memory but **S** does not (*relative-medium-memory*, or RMM), and when all tables comfortably fit in memory (*relative-large-memory*, or RLM). These three regions roughly correspond to the three parts of the curve for S in Figure 4. Factorize (FL) seems the fastest in all three regions for this particular set of parameter values. At RSM, Stream (S) is slower than Materialize (M) due to repeated repartitioning. Stream-Reuse (SR), which avoids repartitioning, is faster, and is comparable to FL. But at RMM, we see a crossover and S is faster than M since M needs to read a larger table. Since no partitioning is needed, SR is comparable to S, while FL is slightly faster. At RLM, we see another crossover and M becomes slightly faster than S (and SR) again, while FL is even faster. This is because the CPU costs dominate at RLM, and M has lower CPU costs than S (and SR). The I/O-CPU breakdown shows that the overall trends mirror the I/O costs at RSM and RMM but mirrors the CPU costs at RLM. Figure 4(C) shows that FL has a lower CPU cost by a factor that roughly equals the redundancy ratio (≈ 2.1). But as expected, the difference is slightly lower since FL performs extra computations to manage an associative array.

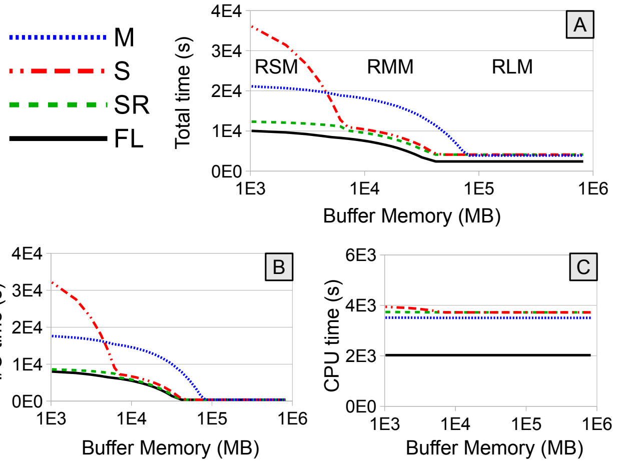


Figure 4: Analytical cost model-based plots for varying the buffer memory (m). (A) Total time, (B) I/O time (with $100MB/s$ I/O rate), and (C) CPU time (with $2.5GHz$ clock). The values fixed are $n_S = 10^8$ (in short, $1E8$), $n_R = 1E7$, $d_S = 40$, $d_R = 60$, and $Iters$ = 20. Note that the x axes are in logscale.

Memory Region	Parameter Varied		
	n_S for n_S / n_R	d_R for d_R / d_S	Iters
RSM	n_R = 5E7, d_S = 40 d_R = 60, Iters = 20	n_S = 5E8, n_R = 5E7 d_S = 40, Iters = 20	n_S = 5E8, n_R = 5E7 d_S = 40, d_R = 60
RMM	n_R = 1E7, d_S = 40 d_R = 60, Iters = 20	n_S = 1E8, n_R = 1E7 d_S = 40, Iters = 20	n_S = 1E8, n_R = 1E7 d_S = 40, d_R = 60
RLM	n_R = 5E6, d_S = 6 d_R = 9, Iters = 20	n_S = 5E7, n_R = 5E6 d_S = 6, Iters = 20	n_S = 5E7, n_R = 5E6 d_S = 6, d_R = 9

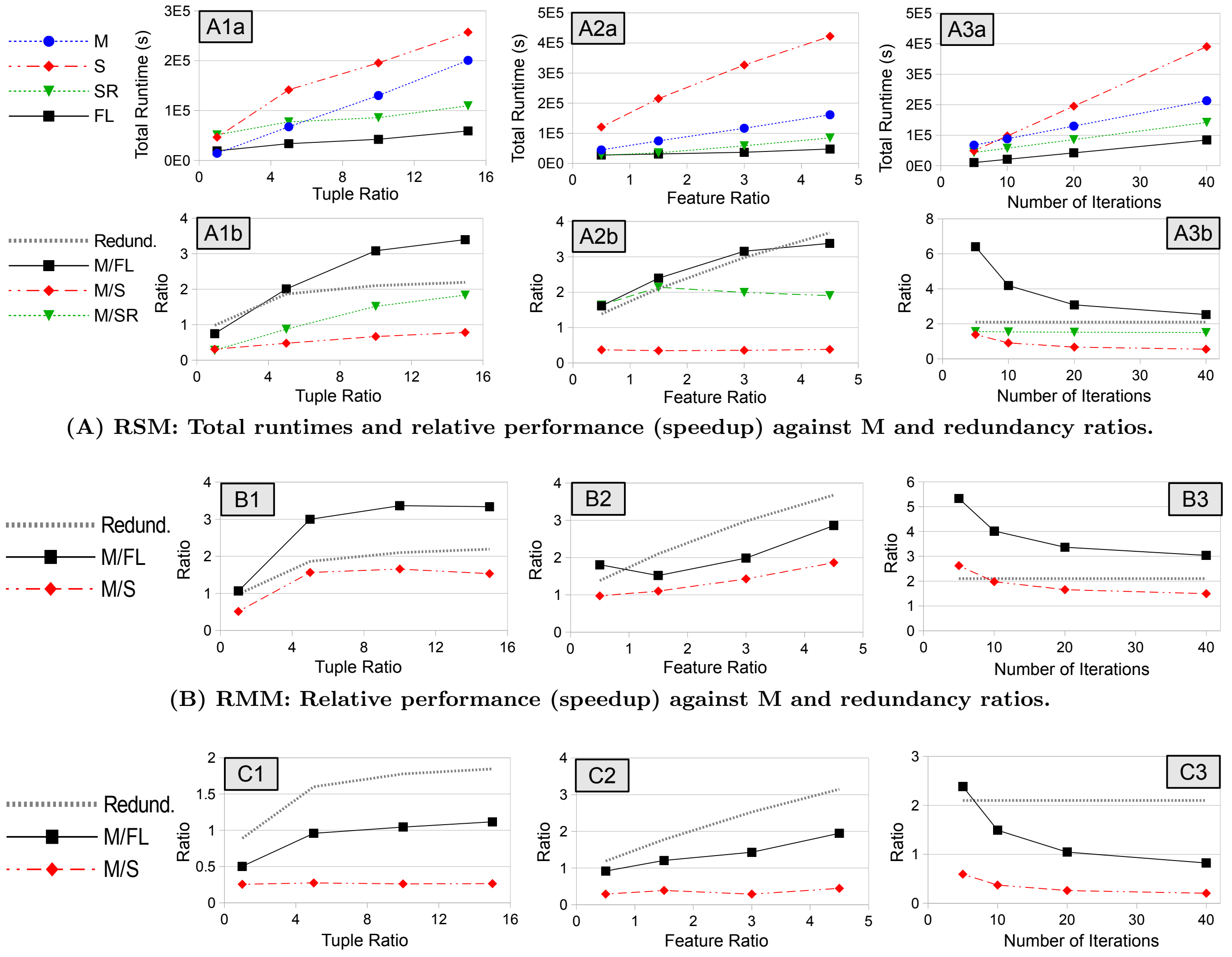
Table 4: Parameters used for the single-node setting results in Figure 5. NB: $xEy \equiv x \times 10^y$.

5.2 Performance Drill Down

Staying with the single-node RDBMS setting, we now drill down deeper into each memory region and study the effect of the major dataset and algorithm parameters using our implementations. For each memory region, we vary each of three major parameters – tuple ratio ($\frac{n_S}{n_R}$), feature ratio ($\frac{d_R}{d_S}$), and number of BGD iterations ($Iters$) – one at a time, while fixing all the others. We use a decaying stepsize (α) rather than a line search for simplicity of exposition. Thus, $Iters$ is also the actual number of passes over the dataset [13, 26]. Finally, we assess whether our cost models are able to accurately predict the observed performance trends and discuss some practical implications.

Experimental Setup: All four approaches were prototyped on top of PostgreSQL (9.2.1) using UDAFs written in C and control code written in Python. The experiments were run

[6] `http://pages.cs.wisc.edu/~arun/orion`



(A) RSM: Total runtimes and relative performance (speedup) against M and redundancy ratios.

(B) RMM: Relative performance (speedup) against M and redundancy ratios.

(C) RLM: Relative performance (speedup) against M and redundancy ratios.

Figure 5: Implementation-based performance against each of (1) tuple ratio ($\frac{n_S}{n_R}$), (2) feature ratio ($\frac{d_R}{d_S}$), and (3) number of iterations ($Iters$) – separated column-wise – for the (A) RSM, (B) RMM, and (C) RLM memory region – separated row-wise. SR is skipped for RMM and RLM since its runtime is very similar to S. The other parameters are fixed as per Table 4.

on machines with Intel Xeon X5650 2.67GHz CPUs, 24GB RAM, 1TB disk, and Linux 2.6.18-194.3.1.el5. The dataset sizes are chosen to fit each memory region's criteria.

Results: The implementation-based runtimes and speedup (and redundancy) ratios for RSM are plotted in Figure 5(A). The corresponding parameter values chosen (for those parameters that are not varied) are presented in Table 4. We present the corresponding speedup ratios for RMM in Figure 5(B) and for RLM in Figure 5(C). Due to space constraints, we skip the runtime plots for RMM and RLM here but present them in the appendix.

- **RSM:** The plots in Figure 5(A) shows that S is the slowest in most cases, followed by the state-of-the-art approach, M. SR is significantly faster than M, while FL is the fastest. This trend is seen across a wide range of values for all 3 parameters – tuple ratio, feature ratio, and number of iterations. All the approaches seem to scale almost linearly with each parameter. Figure 5(A1a) shows a small region where SR is slower than M. This arises because the cost of partitioning both **R** and **S** is slightly more than the cost of materializing **T** at low tuple ratios. In many cases, the performance of SR is comparable to FL, even though the latter performs an extra read of **R** to compute ∇F. The speedup plots show that the speedup of FL over M is mostly higher than the redundancy ratio (r) – this is because the cost of materializing **T** is ignored by r. Figure 5(A3b) confirms this reason as it shows the speedup dropping with iterations since the cost of materialization gets amortized. In contrast, the speedups of SR over M are mostly lower than r.
- **RMM:** The plots in Figure 5(B) show that S could become faster than M (as predicted by Figure 4). SR (skipped here for brevity) is roughly as fast as S, since

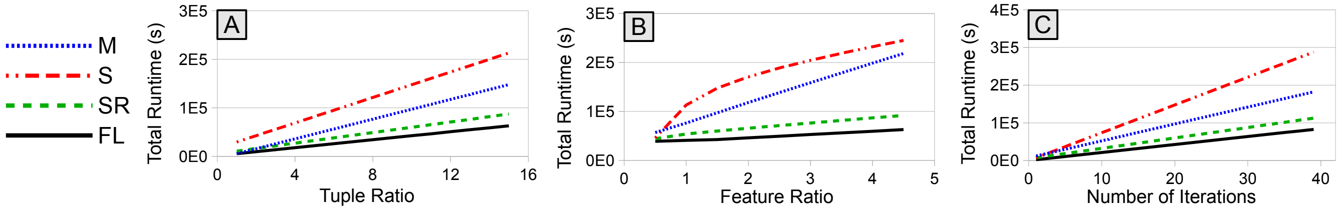


Figure 6: Analytical cost model-based plots of performance against each of (A) $\frac{n_S}{n_R}$, (B) $\frac{d_R}{d_S}$, and (C) $Iters$ for the RSM region. The other parameters are fixed as per Table 4.

no partitioning occurs. FL is the fastest in most cases, but interestingly, Figure 5(B2) shows that the speedup of FL over M is lower than the redundancy ratio for larger feature ratios. This is because $|\mathbf{R}|$ increases and FL reads it twice, which means its relative runtime increases faster than the redundancy ratio.

- **RLM:** The plots in Figure 5(C) show that M is faster than S again. Interestingly, the speedup of FL over M is smaller than r in most cases. In fact, Figures 5(C1,C2) show that M is faster than FL at low dimension ratios (but slower at higher ratios). Figure 5(C3) shows that the amortization of materialization cost could pay off for large values of $Iters$. The lower speedup of FL occurs because all the data fit in memory and the runtime depends mainly on the CPU costs. Thus, the relative cost of managing $\mathbf{H}$ in FL for each iteration (note that M needs to write $\mathbf{T}$ only once) against the cost of BGD's computations for each iteration determines the relative performance. This is also why S (and SR) are much slower than M. Since the CPU cost of BGD increases with the dimension ratios, FL, which reduces the computations for BGD, is faster than M at higher values of both ratios.

Cost Model Accuracy. Our main goal for our analytical models was to understand the fine-grained behavior of each approach and to enable us to quickly explore the relative performance trends of them all for different parameter settings. Thus, we now verify if our models predicted the trends correctly. Figure 6 presents the performance results predicted by our cost models for the RSM region. A comparison of the respective plots of Figure 5(A) with Figure 6 shows that the runtime trends of each approach are largely predicted correctly, irrespective of what parameter is varied. As predicted by our models, crossovers occur between M and S at low $Iters$, between M and SR at low tuple ratios, and between M and FL at low tuple ratios. We found similar trends for RMM and RLM as well but due to space constraints, we present their plots in the appendix.

Since no single approach dominates all others, we also verify if our cost model can predict the fastest approach correctly. We found that, overall, it correctly predicts the fastest approach in 95% of the cases shown in Figure 5. The accuracy of the predicted runtimes, however, varies across approaches and memory regions. For example, the standard R^2 score for FL on RMM is 0.77, while that for SR on RSM is 0.27. Similarly the median percentage error for FL on RSM is 14%, while that for S on RLM is 73%. Due to space constraints, we present more details in the appendix. We think it is interesting future work to improve the absolute accuracy of our cost model, say, by making our models more fine-grained, and by performing a more careful calibration.

Discussion. We briefly discuss a few practical aspects of our proposed approaches.

Application in a System: Recent systems such as Columbus [20, 33] and MLBase [21] provide a high-level language that includes both relational and ML operations. Such systems optimize the execution of logical ML computations by choosing among alternative physical plans using cost models. While we leave it to future work, we think it is possible to apply our ideas for learning over joins by integrating our cost models with their optimizers. An alternative way is to create hybrid approaches, say, like the hybrid of SR and FL that we present in the appendix. The disadvantage is that it is more complex to implement. It is also not clear if it is possible to create a "super-hybrid" that combines FL with both SR and M. We leave a detailed study of hybrid approaches to future work.

Convergence: The number of iterations ($Iters$) parameter might be unknown a priori if we use a convergence criterion for BGD such as the relative decrease in loss. To the best of our knowledge, there is no proven technique to predict the number of iterations for a given quality criterion for any gradient method. While Figures 5(A3b,B3,C3) show that FL is faster irrespective of $Iters$, crossovers might occur for other parameter values. In cases where crossovers are possible, we think a "dynamic" optimizer that tracks the costs of many approaches might be able switch to a faster approach after a particular iteration.

Summary. Our results with both implementations and cost models show that Factorize can be significantly faster than the state-of-the-art Materialize, but is not always the fastest. Stream is often, but not always, faster than Materialize, while Stream-Reuse is faster than Stream and sometimes comparable to Factorize. A combination of the buffer memory, dataset dimensions, and the number of BGD iterations affects which approach is the fastest. Our cost models largely predict the trends correctly and achieve high accuracy in predicting the fastest approach. Thus, they could be used by an optimizer to handle learning over joins.

5.3 Evaluation of Extensions

We now focus on evaluating the efficiency and effectiveness of each of our extensions – scaling FL to large values of n_R, multi-table joins for FL, and shared-nothing parallelism.

5.3.1 Scaling FL along n_R

Using our analytical cost models, we now compare the performance of our three extensions to FL when $\mathbf{H}$ cannot fit in memory. Note that $|\mathbf{H}| \approx 34n_R$ bytes, which implies that for $m = 24\text{GB}$, FL can scale up to $n_R \approx 750E6$. Attribute tables seldom have so many tuples (unlike entity tables).

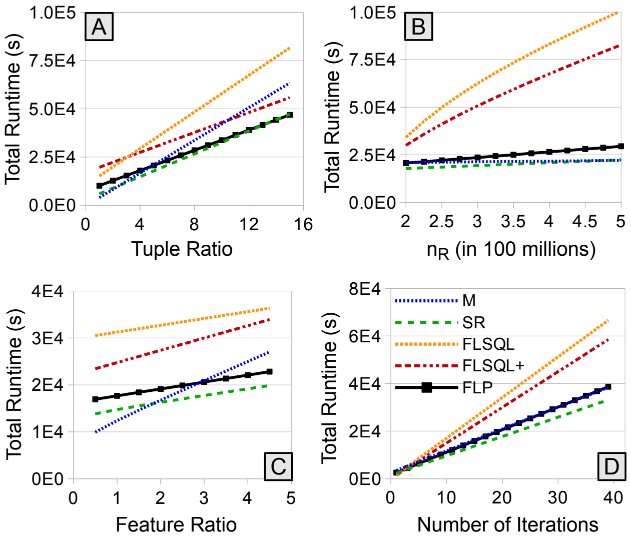


Figure 7: Analytical plots for when m is insufficient for FL. We assume $m = 4$GB, and plot the runtime against each of n_S, d_R, $Iters$, and n_R, while fixing the others. Wherever they are fixed, we set $(n_S, n_R, d_S, d_R, Iters) = (1E9, 2E8, 2, 6, 20)$.

Set	I/O Cost	M	FL		
			PA	GH	OP
1	Partitioning	1,384	171	31	28
	Iters = 10	4,745	2,098	1,959	1,955
2	Partitioning	3,039	338	164	145
	Iters = 10	10,239	3,941	3,766	3,748

Table 5: I/O costs (in 1000s of 1 MB pages) for multi-table joins. Set 1 has $k = 5$, i.e., 5 attribute tables, while Set 2 has $k = 10$. We set $n_S = 2E8$, $d_S = 10$, while n_i and d_i $(i > 0)$ range from $1E7$ to $6E7$ and 35 to 120 respectively. We set $m = 4$GB.

Hence, we use a smaller value of $m = 4$GB. We also vary n_R for this comparison. Figure 7 presents the results.

The first observation is that FLSQL is the slowest in most cases, while FLSQL+ is slightly faster. But, as suspected, there is a crossover between these two at low tuple ratios. More suprisingly, FLP and SR have similar performance across a wide range of all parameters (within this low memory setting), with SR being slightly faster at high feature ratios, while M becomes faster at low feature ratios.

5.3.2 *Multi-table Joins for FL*

We compare three alternative approaches to solve FL-MULTJOIN using our analytical cost model – Partition-All (PA), a baseline heuristic that partitions all $\mathbf{R}_i$ in $O(k)$ time, Optimal (OP), which solves the problem exactly in $O(2^k)$ time, and the $O(klog(k))$ time greedy heuristic (GH). We perform this experiment for two different sets of inputs: one with $k = 5$ and the other with $k = 10$, with a range of different sizes for all the tables. We report the I/O costs for the plan output by each approach. Table 5 presents the results.

For Set 1 $(k = 5)$, PA has a partitioning cost that is nearly 6 times that of OP. But GH is only about 12% higher than OP, and both GH and OP partition only one of the five attribute tables. As expected, PA closes the gap on total cost as we start running iterations for BGD. At $Iters = 10$, PA has only 7% higher cost than OP, whereas M is 140% higher. A similar trend is seen for Set 2, with the major difference being that the contribution of the partitioning cost to the total cost becomes higher for both GH and OP, since they partition six out of the ten attribute tables.

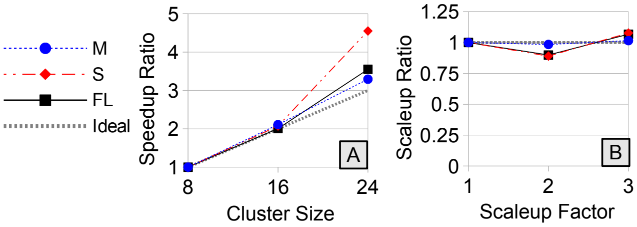


Figure 8: Parallelism with Hive. (A) Speedup against cluster size (number of worker nodes) for $(n_S, n_R, d_S, d_R, Iters) = (15E8, 5E6, 40, 120, 20)$. Each approach is compared to itself, e.g., FL on 24 nodes is 3.5x faster than FL on 8 nodes. The runtimes on 24 nodes were 7.4h for S, 9.5h for FL, and 23.5h for M. (B) Scaleup as both the cluster and dataset size are scaled. The inputs are the same as for (A) for 8 nodes, while n_S is scaled. Thus, the size of T varies from 0.6TB to 1.8TB.

5.3.3 *Shared-nothing Parallelism*

We compare our Hive implementations of M, S, and FL.[7] Our goal is to verify if similar runtime tradeoffs as for the RDBMS setting apply here, and also measure the speedups and scaleups. The setup is a 25-node Hadoop cluster, where each node has two 2.1GHz Intel Xeon processors, 88GB RAM, 2.4TB disk space, and runs Windows Server 2012. The datasets synthesized are written to HDFS with a replication factor of three.[8] Figure 8 presents the results.

Figure 8(A) shows that all three approaches achieve near-linear speedups. The speedups of S and FL are slightly super-linear primarily because more of the data fits in the aggregate memory of a larger cluster, which makes an iterative algorithm such as BGD faster. Interestingly, S is comparable to FL on 8 nodes (S takes 33.5h, and FL, 33.8h), and is faster than FL on 24 nodes (7.4h for S, and 9.5h for FL). Using the Hive query plans and logs, we found that FL spends more time (16%) on Hive startup overheads than S (7%) since it needs more MapReduce jobs. Nevertheless, both S and FL are significantly faster than M, which takes 76.7h on 8 nodes and 23.5h on 24 nodes. We also verified that FL could be faster than S on different inputs. For example, for $(n_S, n_R, d_S, d_R) = (1E9, 100, 20, 2000)$ on 8 nodes, FL is 4.2x faster than S. Thus, while the exact crossover points are different, the runtime tradeoffs are similar to the RDBMS setting in that FL dominates M and S as the redundancy ratio increases. Of course, the runtimes could be better on a different system, and a more complex cost model that includes communication and startup costs might be able to

[7] Due to engineering issues in how we can use Hive's APIs, we use FLSQL+ instead of FLP for FL in some cases.

[8] Although the aggregate RAM was slightly more than the raw data size, not all of the data fit in memory. This is due to implementation overheads in Hive and Hadoop that resulted in Hive using an aggregate of 1.2TB for a replicated hash table created by its broadcast hash join implementation.

predict the trends. We consider this as an interesting avenue for future work. Figure 8(B) shows that all approaches achieve near-linear scaleups. Overall, we see that similar runtime tradeoffs as for the RDBMS setting apply, and that our approaches achieve near-linear speedups and scaleups.

6. RELATED WORK

We now discuss how our problem and our approaches are related to prior work in the literature.

Factorized computation: The abstraction of factorized databases was proposed recently to improve the efficiency of RDBMSs [8]. The basic idea is to succintly represent relations with join dependencies using algebraically equivalent forms that store less data physically. By exploiting the distributivity of Cartesian product over a union of sets, they enable faster computation of relational operations over such databases. However, as they admit, their techniques apply only to in-memory datasets. Our work can be seen as a special case that only has key-foreign key joins. We extend the general idea of factorized computation to ML algorithms over joins. Furthermore, our work considers datasets that may not fit in memory. We explore the whole tradeoff space and propose new approaches that were either not considered for, or are inapplicable to, relational operations. For example, the iterative nature of BGD for learning GLMs enables us to design the Stream-Reuse approach. Recent work [28] has also shown that combining features from multiple tables can improve ML model quality, and that factorizing computations can improve efficiency. They focus on a specific ML algorithm named factorization machine that is used for a recommendation systems application. Using a simple abstraction called "block-structured dataset" that encodes the redundancy information in the data, they reduce computations. However, as they admit, their techniques apply only to in-memory datasets. We observe that designing block-structured datasets essentially requires joins of the base tables – a problem not recognized in [28]. Hence, we take a first-principles approach towards the problem of learning over joins. By harnessing prior work from the database literature, and avoiding explicit encoding of redundancy information, we handle datasets that may not fit in memory. We devise multiple approaches that apply to a large class of ML models (GLMs) and also extend them to a parallel setting. Finally, our empirical results show that factorizing computation for ML may not always be the fastest approach, necessitating a cost model such as ours.

Query optimization: As noted in [8], factorized computations generalize prior work on optimizing SQL aggregates over joins [10, 32]. While BGD over joins is similar at a high level to executing a `SUM` over joins, there are major differences that necessitate new techniques. First, BGD aggregates feature vectors, not single features. Thus, BGD requires more complex statistics and rearrangement of computations as achieved by factorized learning. Second, BGD is iterative, resulting in novel interplays with the join algorithm, e.g., the Stream-Reuse approach. Finally, there exist non-commutative algorithms such as Stochastic Gradient Descent (SGD) that might require new and more complex tradeoffs. While we leave SGD for future work, this paper provides a framework for designing and evalutating solutions for that problem. Learning over joins is also similar in spirit to multi-query optimization (MQO) in which the system optimizes the execution of a bunch of queries that are presented together [30]. Our work deals with join queries that have sequential dependencies due to the iterative nature of BGD, not a bunch of queries presented together.

Analytics systems: There are many commercial and open-source toolkits that provide scalable ML and data mining algorithms [2, 18]. However, they all focus on implementations of individual algorithms, not on pushing ML algorithms through joins. There is increasing research and industrial interest in building systems that achieve closer integration of ML with data processing. These include systems that combine linear algebra-based languages with data management platforms [4, 15, 34], systems for Bayesian inference [9], systems for graph-based ML [23], and systems that combine dataflow-based languages for ML with data management platforms [21, 22, 33]. None of these systems address the problem of learning over joins, but we think our work is easily applicable to the last group of systems. We hope our work contributes to more research in this direction. Analytics systems that provide incremental maintenance over evolving data for some ML models have been studied before [19, 25]. However, neither of those papers address learning over joins. It is interesting future work to study the interplay between these two problems.

7. CONCLUSION AND FUTURE WORK

Key-foreign key joins are often required prior to applying ML on real-world datasets. The state-of-the-art approach of materializing the join output before learning introduces redundancy avoided by normalization, which could result in poor end-to-end performance in addition to storage and maintenance overheads. In this work, we study the problem of learning over a join in order to avoid such redundancy. Focusing on generalized linear models solved using batch gradient descent, we propose several alternative approaches to learn over a join that are also easy to implement over existing RDMBSs. We introduce a new approach named factorized learning that pushes the ML computations through a join and avoids redundancy in both I/O and computations. Using analytical cost models and real implementations on PostgreSQL, we show that factorized learning is often substantially faster than the alternatives, but is not always the fastest, necessitating a cost-based approach. We also extend all our approaches to multi-table joins as well as a shared nothing parallel setting such as Hive.

With increasing research and industrial interest in closely integrating advanced analytics with data processing, we think our work lays a foundation for more research on integrating ML with relational operations in the context of feature engineering. As for future work, we are working on extending factorized learning to other popular algorithms to solve GLMs such as stochastic gradient descent and coordinate descent methods. Another area is pushing other popular ML techniques such as non-linear SVMs, decision trees, neural networks, and clustering algorithms through joins. Since the data access behavior of these techniques might differ from that of GLMs with BGD, it is not clear if it is straightforward to extend our ideas to these techniques. We also intend to formally analyze the effects of the redundancy introduced by database dependencies on ML algorithms. Finally, we intend to expand the scope of the problem to consider aggregate queries as well as the task of feature selection.

8. REFERENCES

[1] Apache Hive. hive.apache.org.

[2] Apache Mahout. mahout.apache.org.

[3] IBM Report. www-01.ibm.com/software/data/bigdata/.

[4] Oracle R Enterprise.

[5] SAS Report on Analytics. sas.com/reg/wp/corp/23876.

[6] A. Agarwal et al. A Reliable Effective Terascale Linear Learning System. *JMLR*, 15:1111–1133, 2014.

[7] M. Anderson et al. Brainwash: A Data System for Feature Engineering. In *CIDR*, 2013.

[8] N. Bakibayev et al. Aggregation and Ordering in Factorised Databases. In *VLDB*, 2013.

[9] Z. Cai et al. Simulation of Database-valued Markov Chains Using SimSQL. In *SIGMOD*, 2013.

[10] S. Chaudhuri and K. Shim. Including Group-By in Query Optimization. In *VLDB*, 1994.

[11] G. B. Dantzig. Discrete-Variable Extremum Problems. *Operations Research*, 5(2):pp. 266–277, 1957.

[12] S. Das et al. Ricardo: Integrating R and Hadoop. In *SIGMOD*, 2010.

[13] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a Unified Architecture for in-RDBMS Analytics. In *SIGMOD*, 2012.

[14] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[15] A. Ghoting et al. SystemML: Declarative Machine Learning on MapReduce. In *ICDE*, 2011.

[16] J. Gray et al. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Min. Knowl. Discov.*, 1(1):29–53, Jan. 1997.

[17] T. Hastie et al. *The Elements of Statistical Learning: Data mining, Inference, and Prediction*. Springer-Verlag, 2001.

[18] J. Hellerstein et al. The MADlib Analytics Library or MAD Skills, the SQL. In *VLDB*, 2012.

[19] M. L. Koc and C. Ré. Incrementally Maintaining Classification using an RDBMS. In *VLDB*, 2011.

[20] P. Konda, A. Kumar, C. Ré, and V. Sashikanth. Feature Selection in Enterprise Analytics: A Demonstration using an R-based Data Analytics System. In *VLDB*, 2013.

[21] T. Kraska et al. MLbase: A Distributed Machine-learning System. In *CIDR*, 2013.

[22] A. Kumar et al. Hazy: Making it Easier to Build and Maintain Big-data Analytics. *CACM*, 56(3):40–49, March 2013.

[23] Y. Low et al. GraphLab: A New Framework For Parallel Machine Learning. In *UAI*, 2010.

[24] T. M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[25] M. Nikolic et al. LINVIEW: Incremental View Maintenance for Complex Analytical Queries. In *SIGMOD*, 2014.

[26] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, 2006.

[27] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 2003.

[28] S. Rendle. Scaling Factorization Machines to Relational Data. In *VLDB*, 2013.

[29] P. G. Selinger et al. Access Path Selection in a Relational Database Management System. In *SIGMOD*, 1979.

[30] T. K. Sellis. Multiple-Query Optimization. *ACM TODS*, 13(1):23–52, Mar. 1988.

[31] L. D. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM TODS*, 11(3):239–264, Aug. 1986.

[32] W. P. Yan and P.-Å. Larson. Eager Aggregation and Lazy Aggregation. In *VLDB*, 1995.

[33] C. Zhang, A. Kumar, and C. Ré. Materialization Optimizations for Feature Selection Workloads. In *SIGMOD*, 2014.

[34] Y. Zhang, W. Zhang, and J. Yang. I/O-Efficient Statistical Computing with RIOT. In *ICDE*, 2010.

Acknowledgments

We thank David DeWitt, Stephen J. Wright, Robert McCann, Jiexing Li, Bruhathi Sundarmurthy, Wentao Wu, and the members of the Microsoft Jim Gray Systems Lab for their feedback on this paper. This work is supported by a research grant from the Microsoft Jim Gray Systems Lab. All views expressed in this work are that of the authors and do not necessarily reflect any views of Microsoft.

APPENDIX

A. COSTS OF STREAM

I/O Cost. If $(m-1) \leq \lceil f|\mathbf{R}|\rceil$:

```
Iters * [
  (|R| + |S|)               //First read
+ (|R| + |S|).(1 - q)       //Write temp partitions
+ (|R| + |S|).(1 - q)       //Read temp partitions
]
```

If $(m-1) > \lceil f|\mathbf{R}|\rceil$:

```
Iters * [
  (|R| + |S|)
- min{|R|+|S|,(m-1) - f|R|]}.(Iters - 1)
]
```

CPU Cost

```
Iters.[
    (nR+nS).hash             //Partition R and S
  + nR.(1+dR).copy           //Construct hash on R
  + nR.(1+dR).(1-q).copy     //R output partitions
  + nS.(2+dS).(1-q).copy     //S output partitions
  + nR.(1-q).hash            //Hash rest of R
  + nS.(1-q).hash            //Hash rest of S
  + nS.comp.f                //Probe for all of S
  + nS.d.(mult+add)          //Compute w.xi
  + nS.(funcG+funcF)         //Apply functions
  + nS.d.(mult+add)          //scale and add
  + nS.add                   //Add for total loss
]
```

B. COSTS OF STREAM-REUSE

I/O Cost. If $(m-1) \leq \lceil f|\mathbf{R}|\rceil$:

```
  (|R| + |S|)               //First read
+ (|R| + |S|).(1 - q)       //Write temp partitions
+ (|R| + |S|).(1 - q)       //Read of iter 1
+ (Iters - 1).(|R| + |S|)  //Remaining iterations
- (Iters - 1).min{|R|+|S|,[(m-2) - f|R0|]} //Cache
```

If $(m-1) > \lceil f|\mathbf{R}|\rceil$:

```
  (|R| + |S|)
+ (Iters - 1).|S|
- (Iters - 1).min{|S|,[(m-1) - f|R|]}
```

CPU Cost

```
  (nR+nS).hash          //Partition R and S
+ nR.(1+dR).copy        //Construct hash on R
+ nR.(1+dR).(1-q).copy //R output partitions
+ nS.(2+dS).(1-q).copy //S output partitions
+ nR.(1-q).hash         //Hash rest of R
+ nS.(1-q).hash         //Hash rest of S
+ nS.comp.f             //Probe for all of S
+ (Iters-1).[
    nR.hash             //Construct hash on R
  + nR.(1+dR).copy      //Construct hash on R
  + nS.(hash + comp.f) //Probe for all of S
]
+ Iters.[                  //Compute gradient
    nS.d.(mult+add)        //Compute w.xi
  + nS.(funcG+funcF)       //Apply functoins
  + nS.d.(mult+add)        //Scale and add
  + nS.add                 //Add for total loss
]
```

C. PROOF OF PROPOSITION 4.1

Given $\mathbf{S} = \{(sid, fk, y, \mathbf{x}_S)_i\}_{i=1}^{n_S}$, and $\mathbf{R} = \{(rid, \mathbf{x}_R)_i\}_{i=1}^{n_R}$, with $n_S > n_R$, the output of the join-project query $\mathbf{T} \leftarrow \pi(\mathbf{R} \bowtie_{\mathbf{R}.rid=\mathbf{S}.fk} \mathbf{S})$ is $\mathbf{T} = \{(sid, y, \mathbf{x})_i\}_{i=1}^{n_S}$. Note that *sid* is the primary key of $\mathbf{S}$ and $\mathbf{T}$, while *rid* is the primary key of $\mathbf{R}$, and fk is a foreign key in $\mathbf{S}$ that points to *sid* of $\mathbf{R}$. Denote the joining tuples $s \in \mathbf{S}$ and $r \in \mathbf{R}$ that produce a given $t \in \mathbf{T}$ as $S(t)$ and $R(t)$ respectively. Also given is $\mathbf{w} \in \mathbb{R}^d$, which is split as $\mathbf{w} = [\mathbf{w}_S \ \mathbf{w}_R]$, where $|\mathbf{w}_R| = d_R$.

Materialize and Stream both operate on $\mathbf{T}$ (the only difference is *when* the tuples of $\mathbf{T}$ are produced). Thus, they output identical values of $\nabla F = \sum_{t \in \mathbf{T}} G(t.y, \mathbf{w}^T t.\mathbf{x}) t.\mathbf{x}$ and $F = \sum_{t \in \mathbf{T}} F_e(t.y, \mathbf{w}^T t.\mathbf{x})$. Denote their output $(\nabla F^*, F^*)$, with $\nabla F^* = [\nabla F_S^* \ \nabla F_R^*]$, in a manner similar to $\mathbf{w}$.

We first prove that the output F of FL equals F^*. As per the logical workflow of FL (see Figure 3), we have $\mathbf{HR} = \{(rid, pip)_i\}_{i=1}^{n_R}$, s.t. $\forall h \in \mathbf{HR}, \exists r \in \mathbf{R}\ s.t.\ h.rid = r.rid \wedge h.pip = \ \mathbf{w}_R^T r.\mathbf{x}_R$. Also, we have $\mathbf{U} \leftarrow \pi(\mathbf{HR} \bowtie_{\mathbf{HR}.rid=\mathbf{S}.fk} \mathbf{S})$ expressed as $\mathbf{U} = \{(sid, rid, y, \mathbf{x}_S, pip)_i\}_{i=1}^{n_S}$. FL aggregates $\mathbf{U}$ to compute $F = \sum_{u \in \mathbf{U}} F_e(u.y, (\mathbf{w}_S^T u.\mathbf{x}_S + u.pip))$. But $u.pip = HR(u).pip = \mathbf{w}_R^T R(HR(u)).\mathbf{x}_R$ and $\mathbf{w}_S^T u.\mathbf{x}_S = \mathbf{w}_S^T S(u).\mathbf{x}_S$. Due to their join expressions, we also have that $\forall u \in \mathbf{U}$, there is exactly one $t \in \mathbf{T}\ s.t.\ u.sid = t.sid$, which implies $F_e(u.y, (\mathbf{w}_S^T u.\mathbf{x}_S + u.pip)) = F_e(t.y, \mathbf{w}^T t.\mathbf{x})$. That along with the relationship $\mathbf{w}^T t.\mathbf{x} = \mathbf{w}_S^T S(t).\mathbf{x}_S + \mathbf{w}_R^T R(t).\mathbf{x}_R$ implies $F = F^*$.

Next, we prove that the output ∇F_S of FL equals ∇F_S^*. As $\mathbf{S}$ is scanned, FL scales and aggregates the feature vectors to get $\nabla F_S = \sum_{u \in \mathbf{U}} G(u.y, (\mathbf{w}_S^T u.\mathbf{x}_S + u.pip))\mathbf{x}_S$. Applying the same argument as for F, we have that $\forall u \in \mathbf{U}$, there is exactly one $t \in \mathbf{T}\ s.t.\ u.sid = t.sid$, which implies $G(u.y, (\mathbf{w}_S^T u.\mathbf{x}_S\ + u.pip)) = G(t.y, \mathbf{w}^T t.\mathbf{x})$, Thus, we have $\nabla F_S = \nabla F_S^*$.

Finally, we prove that the output ∇F_R of FL equals ∇F_R^*. We have the logical relation $\mathbf{HS} \leftarrow \gamma_{SUM(rid)}(\pi(\mathbf{U}))$ as $\mathbf{HS} = \{(rid, fip)_i\}_{i=1}^{n_R}$, obtained by completing the inner products, applying G, and grouping by *rid*. We have $\forall h \in \mathbf{HS}, h.fip = \sum_{u \in \mathbf{U}: u.rid = h.rid} G(u.y, (\mathbf{w}_S^T u.\mathbf{x}_S + u.pip))$. We then have another relation $\mathbf{V} \leftarrow \pi(\mathbf{HS} \bowtie_{\mathbf{HS}.rid=\mathbf{R}.rid} \mathbf{R})$ as $\mathbf{V} = \{(rid, fip, \mathbf{x}_R)_i\}_{i=1}^{n_R}$. Now, FL simply aggregates $\mathbf{V}$ to compute $\nabla F_R = \sum_{v \in \mathbf{V}} (v.fip) v.\mathbf{x}_R$. Due to their join expressions, we also have that $\forall v \in \mathbf{V}$, there is exactly one $r \in \mathbf{R}\ s.t.\ v.rid = r.rid$. Since *rid* imposes a partition on $\mathbf{T}$, we define the partition corresponding to v as $\mathbf{T}_v = \{t \in T | t.rid = v.rid\}$. Thus, $v.fip = \sum_{t \in \mathbf{T}_v} G(t.y, \mathbf{w}^T t.\mathbf{x})$, which coupled with the distributivity of product over a sum, implies $\nabla F_R = \nabla F_R^*$. □

D. PROOF OF PROPOSITION 4.2

The proof for FLSQL is identical to FL, since FLSQL simply materializes some intermediate relations, while the proofs for FLSQL+ and FLP are along the same lines. For FLP, we also use the fact that addition is associative over $\mathbb{R}$ and $\mathbb{R}^d$, and both F and ∇F are just sums of terms. □

E. PROOF OF THEOREM 4.1

We restate the FL-MULTJOIN problem and the theorem.

Problem. Given m, k, $\{|\mathbf{R}_i|\}_{i=1}^k$, $\{|\mathbf{HR}_i|\}_{i=1}^k$ as inputs and k binary variables $\{x_i\}$ to optimize over:

$$\max \sum_{i=1}^{k} x_i |\mathbf{R}_i|, \ \ s.t. \ \sum_{i=1}^{k} x_i (|\mathbf{HR}_i| - 1) \leq m - 1 - k$$

Theorem. FL-MULTJOIN is NP-Hard in l, where $l = |\{i | m - k \geq |\mathbf{HR}_i| > 1\}| \leq k$.

Proof. We prove by a reduction from the 0/1 knapsack problem, which is proven to be NP-Hard. The 0/1 knapsack problem is stated as follows. Given a weight W and n items with respective weights $\{w_i\}$ and values $\{v_i\}$ as inputs and n binary variables $\{z_i\}$ to optimize over, compute $\max \sum_{i=1}^n z_i v_i,\ s.t.\ \sum_{i=1}^n z_i w_i \leq W$. While not necessary, W, $\{w_i\}$, and $\{v_i\}$ are all generally taken to be positive integers. The reduction is obvious now. Set $k = n$, $m = W + k + 1$, $|\mathbf{R}_i| = v_i$ and $|\mathbf{HR}_i| = 1 + w_i$, $\forall i = 1\ to\ k$. Also, $w_i > W \implies z_i = 0$, while $w_i = 0 \implies z_i = 1$. Thus, the actual size of the knapsack problem is $|\{i | W \geq w_i > 0\}|$, which after reduction becomes $|\{i | m - k \geq |\mathbf{HR}_i| > 1\}| = l$. Thus, FL-MULTJOIN is NP-Hard in l. □

F. ADDITIONAL RUNTIME PLOTS

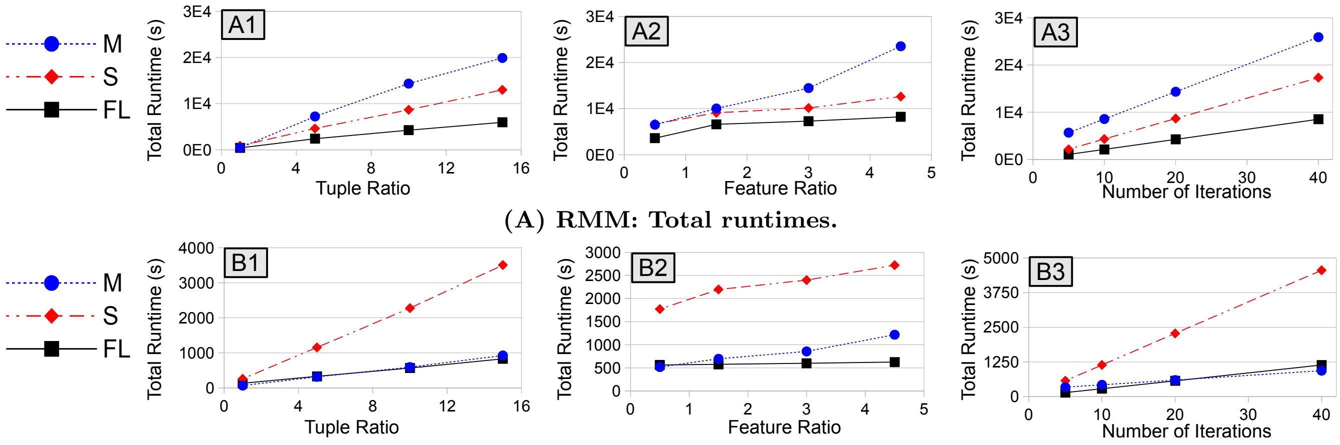
The implementation-based runtime results for RMM and RLM are presented in Figure 9. The analytical cost model-based plots for the same are in Figure 12. Note that the parameters for both these figures are fixed as per Table 4.

G. CASE $|\mathbf{S}| < |\mathbf{R}|$ OR TUPLE RATIO ≤ 1

In this case, an RDBMS optimizer would probably simply choose to build the hash table on $\mathbf{S}$ instead of $\mathbf{R}$. It is straightforward to extend the models of M, S, and, SR to this case. FL, however, is more interesting. Prima facie, it appears that we can switch the access of the tables – construct $\mathbf{H}$ using $\mathbf{S}$, and join that with $\mathbf{R}$, etc. This way, we need only one scan of $\mathbf{R}$ and two of $\mathbf{S}$. The two twists to the FL approach described earlier are: (1) The associative array must store the SID, RID, Y, and PartialIP. (2) We need to doubly index the array since the first join is on RID with $\mathbf{R}$, while the second is on SID with $\mathbf{S}$. Of course, we could use a different data structure as well. Nevertheless, our plots suggest something subtly different.

First, as long as $n_S > n_R$, even if $|\mathbf{S}| < |\mathbf{R}|$, it might still be beneficial to construct $\mathbf{H}$ using $\mathbf{R}$ first as before. This is because the other approaches still perform redundant computations, and FL might still be faster. Figure 10 presents the runtimes for such a setting for varying values of buffer memory as well as the other parameters. The figures confirm our observations above. Of course, it is possible that the above modified version of FL might be faster than the original verion in some cases.

Second, when $n_S \leq n_R$ (irrespective of the sizes of $\mathbf{R}$ and $\mathbf{S}$), there is probably little redundancy in the computations, which means that Materialize is probably the fastest approach. This is because FL performs unnecessary computations on $\mathbf{R}$, viz., with tuples that do not have a joining tuple in $\mathbf{S}$. The above modified version of FL might also be slower than Materialize since the latter obtains a new dataset that probably has almost no redundancy. Figure 11 presents the runtimes for such a setting for varying values of buffer memory as well as the other parameters. The figures confirm our observations above. Of course, in the above, we have implicitly assumed that at most 1 tuple in $\mathbf{S}$ joins with a tuple in $\mathbf{R}$. If we have multiple tuples in $\mathbf{S}$ joining with $\mathbf{R}$, Materialize might still have computational redundancy. In such cases, a more complex hybrid of Materialize and FL might be faster, and we leave the design of such a hybrid approach to future work.



(A) RMM: Total runtimes.

(B) RLM: Total runtimes.

Figure 9: Implementation-based performance against each of (1) tuple ratio ($\frac{n_S}{n_R}$), (2) feature ratio ($\frac{d_R}{d_S}$), and (3) number of iterations ($Iters$) – separated column-wise – for (A) RMM, and (B) RLM – separated row-wise. SR is skipped since its runtime is very similar to S. The other parameters are fixed as per Table 4.

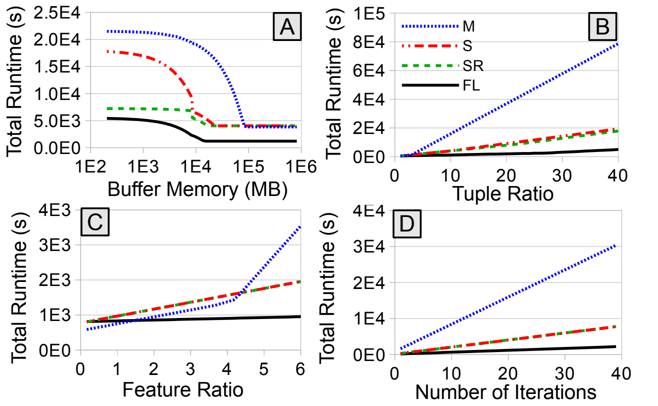


Figure 10: Analytical plots for the case when $|\mathbf{S}| < |\mathbf{R}|$ but $n_S > n_R$. We plot the runtime against each of m, n_S, d_R, $Iters$, and n_R, while fixing the others. Wherever they are fixed, we set $(m, n_S, n_R, d_S, d_R, Iters) = (24GB, 1E8, 1E7, 6, 100, 20)$.

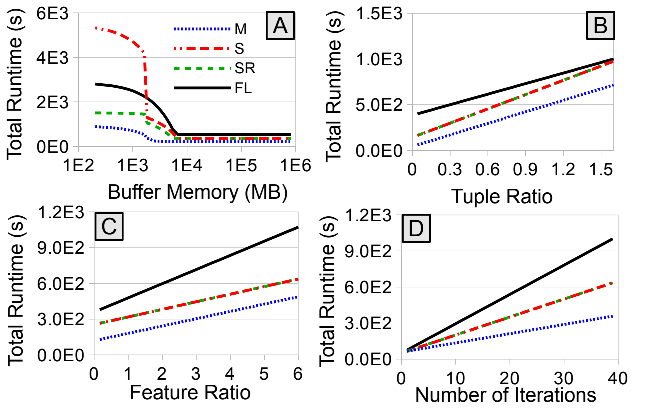


Figure 11: Analytical plots for the case when $n_S \leq n_R$ (mostly). We plot the runtime against each of m, n_S, d_R, $Iters$, and n_R, while fixing the others. Wherever they are fixed, we set $(m, n_S, n_R, d_S, d_R, Iters) = (24GB, 2E7, 5E7, 6, 9, 20)$.

H. MORE COMPLEX APPROACHES

These approaches are more complex to implement since they might require changes to the join implementations.

H.1 Stream-Reuse-Rock (SRR)

1. Similar to Stream-Reuse.
2. Only twist is that for alternate iterations, we flip the order of processing the splits from $0 \rightarrow B$ to $B \rightarrow 0$ and back so as to enable the hash table in cache to be reused across iterations ("rocking the hash-cache").

I/O Cost. If $(m-1) \leq \lceil f|\mathbf{R}|\rceil$:

```
  I/O Cost of Stream-Reuse
+ (Iters - 1).min{|R|+|S|,[(m-2) - f|$R_0$|]}   //Cache
- $\lfloor\frac{Iters}{2}\rfloor$.[|$R_i$| + min{|R|+|S|,(m-2)-f|$R_i$|}]   //H($R_B$)
- $\lfloor\frac{Iters-1}{2}\rfloor$.[|$R_0$| + min{|R|+|S|,(m-2)-f|$R_0$|}] //H($R_0$)
```

If $(m-1) > \lceil f|\mathbf{R}|\rceil$:

```
  I/O Cost of Stream-Reuse
```

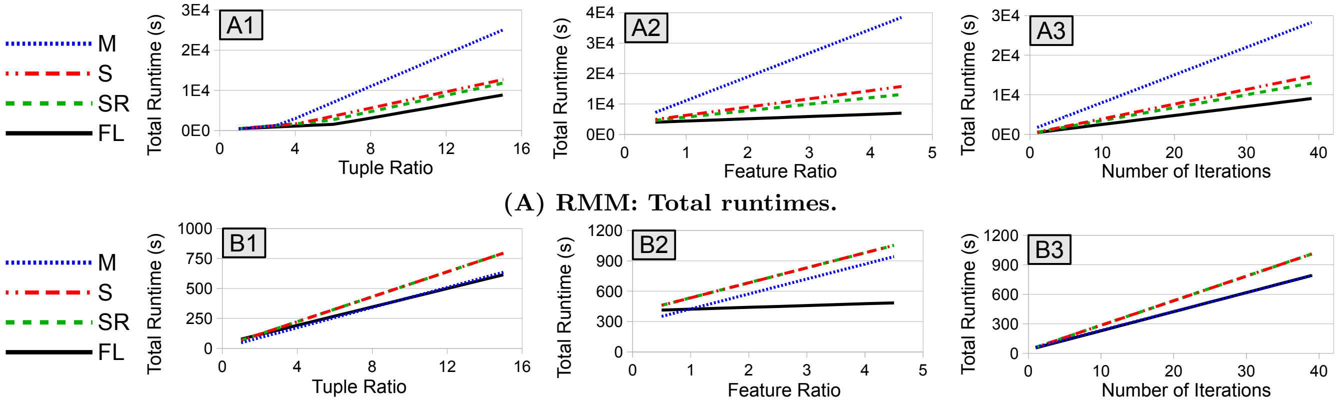
SRR also makes the join implementation "iteration-aware".

CPU Cost

```
  CPU Cost of Stream-Reuse
- $\lfloor\frac{Iters}{2}\rfloor$.[                  //Cache HASH($R_B$)
  nR.$\frac{(1-q)}{B}$.[hash + (1+dR).copy]
]
- $\lfloor\frac{Iters-1}{2}\rfloor$.[                //Cache HASH($R_0$)
  nR.q.[hash + (1+dR).copy]
]
```

H.2 Hybrid of Stream-Reuse-Rock and Factorize (SFH)

1. Let $\mathbf{R}'$ be an augmentation of $\mathbf{R}$ with two columns padded to store statistics. So, $|\mathbf{R}'| = \left\lceil \frac{8n_R.(1+d_R+2)}{p} \right\rceil$.
2. Let $B = \left\lceil \frac{f|\mathbf{R}'|-(m-1)}{(m-1)-1} \right\rceil$. Let $|\mathbf{R}'_0| = \left\lfloor \frac{(m-2)-B}{f} \right\rfloor$, $|\mathbf{R}'_i| = \left\lceil \frac{|\mathbf{R}'|-|\mathbf{R}'_0|}{B} \right\rceil (1 \leq i \leq B)$, and $q = \frac{|\mathbf{R}'_0|}{|\mathbf{R}'|}$.
3. Similar to hybrid hash join on $\mathbf{R}'$ and $\mathbf{S}$.



(A) RMM: Total runtimes.

(B) RLM: Total runtimes.

Figure 12: Analytical plots of runtime against each of (1) $\frac{n_S}{n_R}$, (2) $\frac{d_R}{d_S}$, and (3) $Iters$, for both the (A) RMM, and (B) RLM memory regions. The other parameters are fixed as per Table 4.

4. Only twist is that PartialIP from $\mathbf{R}'_i$ is computed when the $\mathbf{H}$ is constructed on $\mathbf{R}_i$, while SumScaledIP is computed when $\mathbf{S}_i$ is read.
5. Repeat for remaining iterations, reusing the same partitions and rocking the hash-cache as in SRR.

We omit the I/O and CPU costs of SFH here since they are easily derviable from the other approaches.

I. COST MODEL ACCURACY DETAILS

Table 6 presents the discrete prediction accuracy, i.e., how often our cost model predicted the fastest approach correctly for each experiment and memory region corresponding to the results of Figure 5. This quantity matters because a typical cost-based optimizer uses a cost model only to predict the fastest approach. Table 7 presents the standard R^2 score for predicting the absolute runtimes. We split them by approach and memory region because the models are different for each. Similarly, Table 8 presents the mean and median percentage error in the predictions.

Experiment	RSM	RMM	RLM	
Tuple Ratio	75%	75%	100%	83%
Feature Ratio	100%	100%	100%	100%
Iterations	100%	100%	100%	100%
	92%	92%	100%	**95%**

Table 6: Discrete prediction accuracy of cost model.

Approach	RSM	RMM	RLM
Materialize	0.65	0.23	0.65
Stream	0.55	0.88	-1.2
Stream-Reuse	0.27	–	–
Factorize	0.77	0.77	0.67

Table 7: Standard R^2 scores for predicting runtimes.

Approach	RSM	RMM	RLM
Materialize	33% / 30%	33% / 23%	31% / 29%
Stream	33% / 30%	20% / 16%	73% / 75%
Stream-Reuse	42% / 37%	–	–
Factorize	22% / 14%	26% / 19%	25% / 26%

Table 8: Mean / median percentage error for predicting runtimes.

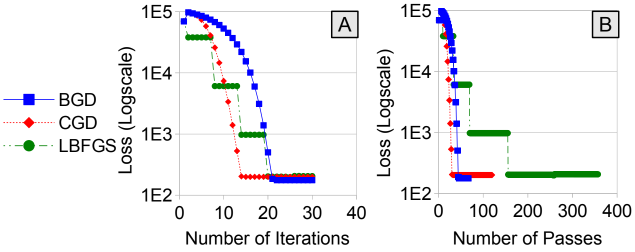


Figure 13: Comparing gradient methods: Batch Gradient Descent (BGD), Conjugate Gradient (CGD), and Limited Memory BFGS (LBFGS) with 5 gradients saved. The parameters are $n_S = 1E5$, $n_R = 1E4$, $d_S = 40$, and $d_R = 60$. (A) Loss after each iteration. (B) Loss after each pass over the data (extra passes needed for line search to tune α).

J. COMPARING GRADIENT METHODS

While our focus in this paper has been on performance at scale for learning over joins, we briefly mention an interesting finding regarding the behavior of different gradient methods on a dataset with the redundancy that we study. This particular experiment is agnostic to the approach we use to learn over the join. Figure 13 plots the loss for three popular gradient methods for LR – BGD, CGD, and LBFGS – against the number of iterations and the number of passes. As expected, BGD takes more iterations to reach a similar loss as CGD. However, the behavior of LBFGS (with five gradients saved to approximate the Hessian) is more perplexing. Typically, LBFGS is known to converge faster than CGD in terms of number of iterations [6, 26] but we observe otherwise on this dataset. We also observe that LBFGS performs many extra passes to tune the stepsize, making it slower than even BGD in this case. However, it is possible that a different stepsize tuning strategy might make LBFGS faster. While we leave a formal explanation to future work, we found that the Hessians obtained had high condition numbers, which we speculate might make LBFGS less appealing [26].