Learning Over Joins

by

Arun Kumar

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2016

Date of final oral examination: 07/21/2016

The dissertation is approved by the following members of the Final Oral Committee:

Jeffrey Naughton, Professor Emeritus, Computer Sciences, UW-Madison; Google
Jignesh M. Patel, Professor, Computer Sciences, UW-Madison
C. David Page Jr., Professor, Biostatistics and Medical Informatics, UW-Madison
Christopher Ré, Assistant Professor, Computer Science, Stanford University
Stephen J. Wright, Professor, Computer Sciences, UW-Madison
Xiaojin Zhu, Associate Professor, Computer Sciences, UW-Madison

© Copyright by Arun Kumar 2016
All Rights Reserved

ACKNOWLEDGMENTS

No amount of words can fully express my infinite gratitude to my co-advisors, Jeff Naughton and Jignesh Patel. This dissertation would not have been possible without their unwavering support for my crazy ideas to explore topics that were outside of their core interests. Jeff's incredible research wisdom and uncanny ability to understand his students' situations have helped me innumerable times, as have Jignesh's infectious go-getter spirit and remarkable grasp on grounding research with practical relevance. These are all attributes that I hope to emulate in my career. I think the most rewarding gift an advisor can give their student is the freedom to pursue their interests and collaborations, while staying closely engaged by giving honest advice and critical feedback. I am very fortunate that Jeff and Jignesh trusted me enough to give me this gift.

I am deeply grateful to Chris Ré for getting me started in data management research as a part of his research group. His patience and confidence in me early on were instrumental in getting me to continue in research. His outstanding ability to weave elegant theoretical insights with solid systems work across multiple areas is a skill that I hope to emulate.

I thank Steve Wright and Jerry Zhu for their collaborations and for serving on my committee. Their deep insights about optimization and machine learning, their patience in explaining new concepts to me, and their honest feedback on my ideas were crucial for this research. I thank David Page for serving on my committee and for his insightful feedback on my talks.

I am also deeply grateful to David DeWitt and the Microsoft Jim Gray Systems Lab for funding my dissertation research and for giving me access to Microsoft's resources without any strings attached. I thank David and the other members of the Lab for their continual feedback on my papers and talks, which is an integral part of the remarkable close-knit community environment of the Lab. I thank Robert McCann from Microsoft for our periodic insightful discussions on research and practice, for his feedback on my papers, and for helping to set up a productive research collaboration with Microsoft.

I was fortunate to be able to mentor a great set of students as part of my dissertation research: Lingjiao Chen, Zhiwei Fan, Mona Jalal, Fengan Li, Boqun Yan, and Fujie Zhan. It is a rewarding experience to work with such bright students and watch them mature as researchers. This is a key reason for me to want to continue in academic research. I am thankful to Jeff and Jignesh for giving me the opportunity to advise these students.

I am thankful to all of my other co-authors and research collaborators throughout my graduate school life. An incomplete list includes Mike Cafarella, Joe Hellerstein, Ben Recht, Aaron Feng, Pradap Konda, Feng Niu, and Ce Zhang. I am grateful to my friends,

Bruhathi Sundarmurthy, Wentao Wu, and the other students of the Database Group, for their continual feedback on my ideas, papers and talks.

Finally, I am indebted beyond measure to my family, especially my father, Kumar, my brother, Balaji, and my sister-in-law, Indira, for their love, support, and advice throughout my graduate school life, especially during the tough times. I am grateful to my other close friends, who were always there to support and help me. An incomplete list includes Levent, Thanu, and Vijay. Last but definitely not the least, I am grateful to my wonderful fiancé, Wade, for his love and support during the crucial final stages of my dissertation research and my job search. I look forward to an exciting journey with him as I move forward to a career in academia.

Overall, I am deeply fortunate to have had, and continue to have, such a great set of mentors and such a wonderful set of loved ones. I will never forget the support of all of these people with my dissertation research and I know I can count on them in any of my future endeavors too.

Most of this dissertation research was funded by a 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.

ABSTRACT

Advanced analytics using machine learning (ML) is increasingly critical for a wide variety of data-driven applications that underpin the modern world. Many real-world datasets have multiple tables with *key-foreign key* relationships, but most ML toolkits force data scientists to join them into a single table before using ML. This process of "learning *after* joins" introduces redundancy in the data, which results in storage and runtime inefficiencies, as well as data maintenance headaches for data scientists. To mitigate these issues, this dissertation introduces the paradigm of "learning *over* joins," which includes two orthogonal techniques: *avoiding joins physically* and *avoiding joins logically*. The former shows how to push ML computations through joins to the base tables, which improves runtime performance without affecting ML accuracy. The latter shows that in many cases, it is possible, somewhat surprisingly, to ignore entire base tables without affecting ML accuracy significantly. Overall, our techniques help improve the usability and runtime performance of ML over multi-table datasets, sometimes by orders of magnitude, without degrading accuracy significantly. Our work forces one to rethink a prevalent practice in advanced analytics and opens up new connections between data management systems, database dependency theory, and machine learning.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

Advanced analytics using machine learning (ML) is critical for a wide variety of data-driven applications ranging from insurance and healthcare to recommendation systems and Web search [Gartner; SAS, b]. This has led to a growing interest in both the data management industry and academia to more closely integrate ML and data processing systems [Oracle; Ghoting et al., 2011; Hellerstein et al., 2012; Kumar et al., 2013]. However, almost all ML toolkits assume that the input to an ML algorithm is a single table even though many real-world datasets are stored as multiple tables connected by *key-foreign key* (KFK) dependencies. Thus, data scientists are forced to perform key-foreign key *joins* to gather features from all base tables and *materialize* the join output as a single table, which is then input to their ML toolkit. This strategy of "learning after joins" introduces redundancy avoided by database normalization, which could lead to extra storage requirements, poorer end-to-end runtime performance, and data maintenance headaches for data scientists due to data duplication [Ramakrishnan and Gehrke, 2003]. This could even become a "show-stopper" issue for advanced analytics: from conversations with a major Web company, we learned that the joins to materialize the single table before an ML task brought down their shared cluster due to the blow up in the data size! Furthermore, the increase in the number of tables and features might make it harder for data scientists to explore the data, especially during feature selection, which is almost always performed in conjunction with learning [Guyon et al., 2006; Zhang et al., 2014].

The thesis of this dissertation is that for a wide variety of ML algorithms, it is often possible to mitigate the above issues by "learning over joins" instead, which involves two novel and orthogonal techniques: (1) "avoiding joins physically," in which ML computations are pushed down through joins, and (2) "avoiding joins logically," in which entire base tables can be, somewhat surprisingly, ignored outright. Using a combination of new algorithms, system design, theoretical analysis, and empirical analysis, this dissertation shows how these two techniques can help improve the usability and runtime performance of ML over multi-table data, sometimes by orders of magnitude, without degrading ML accuracy significantly.

1.1 Example

We start with a detailed real-world example of our problem and introduce relevant terminology from both the ML and database literatures. Figure 1.1 depicts this example

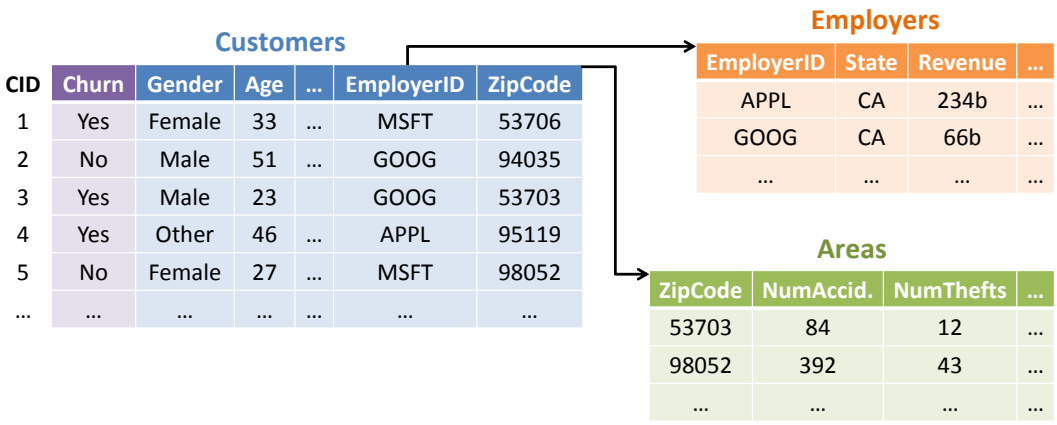


Figure 1.1: Example scenario for ML over multi-table data.

pictorially. An ML task that is ubiquitous in applications such as insurance, retail, and telecommunications is predicting customer *churn*. Churn is the process of a customer leaving a company and moving to its competitor. Companies want to prevent churn because it affects their bottomline. As part of their preventive strategy, they use ML *classification* models to predict which customers are likely to churn so that they could offer them incentives to stay. This involves using a *training dataset* of past customers that have churned or stayed. Let the schema for this dataset be as follows: `Customers(CustomerID, Churn, Gender, Age, ..., EmployerID, ZipCode)`. The attributes of the customer are the *features* for an ML classification *model* such as Logistic Regression or Naive Bayes [Mitchell, 1997]. `Churn` is the *target* for the ML model, also called the *class label*. `EmployerID` is the ID of the customer's employer and it is a *foreign key* that refers to a separate table with details about companies and other organizations. Its schema is: `Employers(EmployerID, State, Revenue, ...)`. Similarly, `ZipCode` is another foreign key that refers to a separate table with details about the areas where customers live, e.g., accident and crime statistics. Its schema is: `Areas(ZipCode, NumAccidents, NumThefts, ...)`. Note that `EmployerID` (resp. `ZipCode`) is the *primary key* (in short, *key*) of `Employers` (resp. `Areas`), i.e., it uniquely identifies each *tuple* (record) in that table.

Given such multi-table data, data scientists almost always compute the relational *join* of all tables – `Customers` $\bowtie$ `Employers` $\bowtie$ `Areas` – before ML to obtain more features for their ML model as part of their *feature engineering*. They might have a hunch that the features of the customers' employers and/or areas might help in predicting customer churn; perhaps customers employed by large corporations and living in low-crime areas might be less likely to churn. The join output is *materialized*, i.e., a single table containing features from

all tables is created. The joins are typically performed in a data processing system such as an RDBMS, Hive/Hadoop, or Spark, or in in-memory toolkits such as R (using the "merge" function) [R]. The materialized table is used as the input to train an ML model.

More Examples ML over KFK joins of multi-table datasets is not specific to insurance or retail or telecommunications; it arises in practically every data-driven application with structured data. We provide a few more real-world examples from other domains.

- **Recommendation Systems.** Predict the rating of a movie/product by a user using a table with past ratings joined with tables about users and movies/products.
- **Hospitality.** Predict the ranking of a hotel in a search listing using a table with past search listings joined with tables about hotels and search events.
- **Web Security.** Predict the duration of a sign-in event using a table with past sign-in information joined with tables about webpages and user accounts.
- **Bioinformatics.** Predict if a gene is responsive to a chemical using a table with gene-chemical interaction information joined with tables about genes and chemicals.

1.2 Technical Contributions

This dissertation introduces the paradigm of "learning over joins" by asking a seemingly simple but fundamental question: *Do we really "need" the KFK joins before ML?* As we will show, this question has several interesting facets, and answering them opens up new connections between data management systems, database dependency theory, and machine learning. We split our technical contributions into three parts.

Orion: Avoiding Joins Physically In this project, we interpret the question posed above as follows: *Is it possible to "physically" avoid performing the KFK joins before ML?* The motivation is simple: RDBMS optimizers have long pushed relational operators such as selections and aggregates down through joins to the base tables to physically avoid joins and potentially improve performance [Ramakrishnan and Gehrke, 2003; Yan and Larson, 1995; Chaudhuri and Shim, 1994]. Surprisingly, this idea had not been explored for ML algorithms. This project is the first work to fill this critical research gap. In our customer churn example (Figure 1.1), this means that we can learn on `Customers`, `Employers`, and `Areas` directly instead of joining them. We focus on a large class of ML models known as Generalized Linear Models (GLMs), solved using an optimization algorithm known as Batch Gradient Descent (BGD). GLMs include such popular ML models as logistic and linear regression, while BGD is similar to many other optimization methods. Systematically

eliminating the redundancy caused by joins in both I/O and computations, we devise several approaches to learn over joins, including *factorized learning*, which decomposes BGD computations and pushes them down through the joins to the base tables. Factorized learning avoids redundancy in both I/O and computations, and it does not affect accuracy. But it introduces non-trivial performance trade-offs, which we analyze in depth using cost models. Using extensive analytical and empirical studies on the PostgreSQL RDBMS, we establish that factorized learning is often the fastest approach to learn over joins, and that our cost model accurately predicts the corner cases where it may not be the fastest. We also extend all our approaches to handle multiple joins as well as to distributed data processing systems. This project is the subject of Chapter 3 and is joint work with Jeffrey Naughton and Jignesh Patel. A paper on this work appeared in the ACM SIGMOD conference in 2015 [Kumar et al., 2015c].

Extensions and Generalization of Orion Having introduced factorized learning for GLMs, we ask: *Is factorized learning extensible to other ML models, and if so, is there a way to extend it generically?* In joint work with several BS and MS students at UW-Madison – Lingjiao Chen, Zhiwei Fan, Mona Jalal, Fengan Li, Boqun Yan, and Fujie Zhan – as well as with Jeffrey Naughton, Jignesh Patel, and Stephen Wright, we show that factorized learning is extensible to several other classes of ML models. First, we extend it to *probabilistic classifiers* such as Naive Bayes and Decision Trees and build the Santoku system that provides implementations of factorized learning (and factorized "scoring") in R. This work was presented as a demonstration in the VLDB conference in 2015 [Kumar et al., 2015a]. Second, we extend factorized learning to two other popular optimization algorithms for GLMs with data access patterns that are different from that of BGD: *Stochastic Gradient Descent* (SGD) and *Stochastic Coordinate Descent* (SCD). Third, we extend it to three popular *clustering* algorithms: K-Means, Hierarchical Agglomerative Clustering, and DBSCAN. We also introduce the approach of "compressed clustering" that extends a popular compression scheme and integrates it with clustering algorithms in order to exploit the redundancy introduced by joins even when the normalized input schema is not known. Finally, we generalize factorized learning to arbitrary ML computations expressible in the formal language of *linear algebra*. We introduce a framework of algebraic rewrite rules to automatically transform ML computations over a matrix that is the output of joins into computations over the normalized input matrices. A brief description of these projects is the subject of Chapter 4. Papers on all of these projects are under submission.

Hamlet: Avoiding Joins Logically Not satisfied with avoiding joins physically, we reinterpret our original question in a more radical way: *What if we avoid a KFK join "logically"*

as well, i.e., ignore a table entirely? In our customer churn example, this means that perhaps `Employers` and/or `Areas` is ignored entirely. At first glance, this question might seem surprising. Dealing with fewer tables (and thus, fewer features) could make ML and feature selection faster and potentially make analysis easier. But what about accuracy? Our first insight is that a KFK dependency means that the foreign key encodes all "information" about the features brought in by the join; formally, the latter features are *redundant*, which motivates us to consider avoiding them and using the foreign key as a representative. But experiments on real data (for classification) show that avoiding joins reduces accuracy drastically in some cases. Thus, we dive deeper and perform a learning theoretical analysis of the effects of avoiding KFK joins on the bias-variance trade-off of ML [Shalev-Shwartz and Ben-David, 2014]. Our analysis shows that avoiding joins might not increase the bias of the final model (after feature selection), but it might increase the variance and thus, decrease the overall accuracy. We confirm our analysis with a comprehensive simulation study. This is a new runtime-accuracy trade-off in ML and we would like to help data scientists understand and exploit it easily. Thus, we devise decision rules to predict when avoiding a join is unlikely to decrease accuracy significantly. We call this process "avoiding joins safely." Our desiderata for such rules are genericity, simplicity, speed, flexibility, and conservatism. An empirical validation with real data shows that our rules work well in practice: in some cases, we see speedups of over two orders of magnitude, while the accuracy is similar. This project is the subject of Chapter 5 and is joint work with Jeffrey Naughton, Jignesh Patel, and Xiaojin Zhu. A paper on this work appeared in the ACM SIGMOD conference in 2016 [Kumar et al., 2016].

1.3 Summary and Impact

Almost all machine learning (ML) toolkits assume that the input dataset to an ML algorithm is a single table but many real-world datasets are multi-table, connected by key-foreign key (KFK) relationships. This forces data scientists to learn *after* joins, which causes runtime and storage inefficiencies. This dissertation mitigates this issue by introducing the paradigm of learning *over* joins, which consists of two orthogonal techniques: avoiding joins *physically* and avoiding joins *logically*. The first technique shows how, for a wide variety of ML models, ML computations can be pushed down through joins to the base tables, which could lower runtimes but does not affect accuracy. The second technique shows that in many cases, the tables brought in by KFK joins can be ignored outright to lower runtimes even further but without lowering accuracy significantly. Our work fills critical technical gaps in advanced analytics and opens up new connections between data management and ML.

Throughout this dissertation research, we interacted with data scientists and software engineers at various enterprise and Web companies to understand the state-of-the-art practice and make it easier for them to adopt our ideas and systems. In particular, at the time of writing this document, both of our techniques – avoiding joins physically and logically – are being explored for use in production by LogicBlox for their retail customers and by Microsoft for their internal usage. Data scientists at Facebook have informed us of their use of avoiding joins logically in their recommendation tasks. Finally, IBM Research Almaden have expressed interest in exploring the integration of our work on generalizing factorized learning using linear algebra into SystemML [Ghoting et al., 2011].

2 Preliminaries

We now present the formal problem setup for learning over joins and introduce some notation that will be used in the rest of this dissertation. We then provide some background on the relevant ML and database concepts.

2.1 Problem Setup and Notation

We call the main table with the entities to model using ML as the *entity table*, denoted $\mathbf{S}$. There are k other tables called *attribute tables*, denoted $\mathbf{R}_i$, for $i = 1$ to k (if $k = 1$, we drop the subscript). The schema of $\mathbf{R}_i$ is $\mathbf{R}_i(\underline{\text{RID}_i}, \mathbf{X}_{R_i})$, where RID_i is its key and $\mathbf{X}_{R_i}$ is a vector (sequence) of features. We abuse the notation slightly to also treat $\mathbf{X}$ as a *set* since the order among features in $\mathbf{X}$ is immaterial in our setting. The schema of $\mathbf{S}$ is $\mathbf{S}(\underline{\text{SID}}, Y, \mathbf{X}_S, \text{FK}_1, \ldots, \text{FK}_k)$, where Y is the target for learning, $\mathbf{X}_S$ is a vector of features, and FK_i is a foreign key that refers to $\mathbf{R}_i$. In database parlance, this is known as a "star" schema.[1] Let $\mathbf{T}$ denote the output of the projected equi-join: $\mathbf{T} \leftarrow \pi(\mathbf{S} \bowtie_{\text{FK}_1=\text{RID}_1} \mathbf{R}_1 \ldots \bowtie_{\text{FK}_k=\text{RID}_k} \mathbf{R}_k)$. In general, its schema is $\mathbf{T}(\underline{\text{SID}}, Y, \mathbf{X}_S, \mathbf{X}_{R_1}, \ldots, \mathbf{X}_{R_k})$. Herein lies a subtle but key distinction: in the HAMLET project, we recognize that the foreign keys FK_i need not just be physical connectors between the tables but features themselves. Thus, the schema of $\mathbf{T}$ could also be $\mathbf{T}(\underline{\text{SID}}, Y, \mathbf{X}_S, \text{FK}_1, \ldots, \text{FK}_k, \mathbf{X}_{R_1}, \ldots, \mathbf{X}_{R_k})$. We will discuss this difference in more detail in Chapter 5.

Example We use our running example of predicting customer churn (Figure 1.1). $\mathbf{S}$ is the `Customers` table, $\mathbf{R}_1$ is the `Employers` table, $\mathbf{R}_2$ is the `Areas` table, Y is `Churn`, $\mathbf{X}_S$ is {`Age`, `Gender`, ...}, FK_1 is `Customers.EmployerID`, RID_1 is `Employers.EmployerID`, and $\mathbf{X}_{R_1}$ is {`State`, `Revenue`, ...}, FK_2 is `Customers.ZipCode`, RID_2 is `Employers.EmployerID`, and $\mathbf{X}_{R_1}$ is {`NumAccidents`, `NumThefts`, ...}.

[1] The key-foreign key dependencies in the star schema ensure that the *distinctness* of the entities in $\mathbf{S}$ are preserved after the joins, which is necessary from a statistical correctness perspective [Hastie et al., 2003]. Otherwise, the same entity might end up with multiple examples – a scenario that requires more complex ML models known as statistical relational learning [Getoor and Taskar, 2007]. We leave this to future work.

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 2.1: GLMs and their functions.

2.2 Background for Orion

We provide a brief introduction to GLMs and BGD. For a deeper description, we refer the reader to Hastie et al. [2003], Mitchell [1997], and Nocedal and Wright [2006].

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 2.1 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.[2]

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 [Nocedal and Wright, 2006]. Table 2.1 also lists the gradient functions of the GLMs. We shall use F and $F(\mathbf{w})$ interchangeably.

[2]Typically, a convex penalty term called a *regularizer* is added to the loss to constrain $\|\mathbf{w}\|_1$ or $\|\mathbf{w}\|_2^2$ [Hastie et al., 2003].

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: $\quad r_{prev} \leftarrow r_{curr}$
4: $\quad (\mathbf{g}_k, r_{curr}) \leftarrow (\nabla F_{k+1}, F_{k+1})$ ▷ 1 pass over data
5: $\quad \mathbf{w}_{k+1} \leftarrow \mathbf{w}_k - \alpha_k \mathbf{g}_k$ ▷ Pick α_k by line search
6: $\quad k \leftarrow k + 1$
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 in a manner similar to step 4 [Nocedal and Wright, 2006].

On large data, it is likely that computing F and ∇F dominates the runtime of BGD [Das et al., 2010; Feng et al., 2012]. 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 [Feng et al., 2012; Gray et al., 1997]. 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.

2.3 Background for Hamlet

Feature Selection Feature selection methods are almost always used along with an ML classifier to help improve accuracy [Guyon et al., 2006]. While our work is orthogonal to feature selection methods, we briefly discuss a few popular ones for concreteness sake. At a high level, there are three types of feature selection methods: *wrappers*, *filters*, and *embedded* methods [Kohavi and John, 1997; Guyon et al., 2006].

A wrapper uses the classifier as a black box to heuristically search for a more accurate subset. Sequential greedy search is a popular wrapper; it has two variants: *forward selection* and *backward selection*. Given a feature set $\mathbf{X}$, forward (resp. backward) selection computes the *error* of an ML model for different subsets of $\mathbf{X}$ of increasing (resp. decreasing) size starting with the empty set (resp. full set $\mathbf{X}$) by adding (resp. eliminating) one feature at

a time. The error can be the holdout validation error or the k-fold cross-validation error. For our purposes, the simpler holdout validation method described in Hastie et al. [2003] suffices: the labeled data is split 50%:25%:25% with the first part used for training, the second part used for the validation error during greedy search, and the last part used for the holdout test error, which is the final indicator of the chosen subset's accuracy.

Filters apply a specified scoring function to each feature $F \in \mathbf{X}$ using the labeled data but independent of any classifier. The top-k features are then chosen, with k picked either manually or tuned automatically using the validation error for a given classifier (we use the latter). Popular scoring functions include *mutual information* $I(F;Y)$ and *information gain ratio* $IGR(F;Y)$. Intuitively, $I(F;Y)$ tells us how much the knowledge of F reduces the *entropy* of Y, while $IGR(F;Y)$ normalizes it by the feature's entropy. The formal definition is as follows.

Definition 2.1. Mutual information. *Given two random variables* A *and* B *with domains* $\mathcal{D}_A$ *and* $\mathcal{D}_B$*, their* mutual information *is* $I(A;B) = H(B) - H(B|A)$*, where* $H(B)$ *is the entropy of* B*. Thus, we have:*

$$I(A;B) = \sum_{a \in \mathcal{D}_A} \sum_{b \in \mathcal{D}_B} P(a,b) \log \frac{P(a,b)}{P(a)P(b)}$$

Embedded methods are "wired" in to the classifier. A common example is L1 or L2 norm *regularization* for linear and logistic regression. These methods perform implicit feature selection by modifying the regression coefficients directly instead of searching for subsets, e.g., L1 norm makes some coefficients vanish, which is akin to dropping the corresponding features [Hastie et al., 2003].

Naive Bayes Probabilistic classifiers assume that data examples arise from some hidden "true" joint probability distribution $P^*(Y, \mathbf{X})$, where Y is a random variable that represents the target (class label), and $\mathbf{X}$ represents the feature vector (with $|\mathbf{X}| = d$ features). Let $\mathcal{D}_Y$ denote the domain of Y, and similarly, $\mathcal{D}_F, \forall F \in \mathbf{X}$. Learning (training) simply becomes computing an estimate P of P^* (or an approximation of it). Given a new example with $\mathbf{X} = \boldsymbol{x}$, the classifier can be used to predict the target using the *maximum a posteriori* (MAP) estimate: $\mathrm{argmax}_{c \in \mathcal{D}_Y} P(Y = c | \mathbf{X} = \boldsymbol{x})$. Using Bayes Rule, we can rewrite $P(Y|\mathbf{X})$ as follows:

$$P(Y|\mathbf{X}) = \frac{P(\mathbf{X}|Y)P(Y)}{P(\mathbf{X})} = \frac{P(\mathbf{X}, Y)}{P(\mathbf{X})} \tag{2.1}$$

In general, $P(\mathbf{X})$ is ignored and substituted with a normalizing constant. $P(\mathbf{X}, Y)$ is estimated by counting the frequencies of all combinations of feature values and class labels in the labeled dataset. Since the number of probabilities to estimate in $P(Y, \mathbf{X})$

grows exponentially in d, the Naive Bayes model mitigates that issue by assuming that the features in $\mathbf{X}$ are conditionally independent, given Y [Pearl, 1988; Mitchell, 1997]. Thus, $P(\mathbf{X}, Y)$ is assumed to factorize as follows:

$$P(\mathbf{X}, Y) = P(Y)P(\mathbf{X}|Y) \approx P(Y)\Pi_{F \in \mathbf{X}} P(F|Y) \tag{2.2}$$

Thus, Naive Bayes needs to estimate only $O(d)$ probabilities, viz., $P(Y)$, and $P(F|Y), \forall F \in \mathbf{X}$. Some (F, Y) combinations might not occur in the training data. To avoid assigning them zero probability, *Laplacian smoothing* is performed by introducing a "dummy" extra count of 1 for all F values when computing $P(F|Y)$ [Mitchell, 1997].

Prediction with Naive Bayes requires looking up the conditional probabilities for a given feature vector and multiplying them to compute the MAP estimate. To avoid numerical underflows, real implementations add logarithms of probabilities instead.

3 Orion: Avoiding Joins Physically

In this chapter, we dive deeper into our technique of avoiding joins physically. Recall that learning *after* joins imposes an artificial barrier between the ML-based analysis and the base tables, resulting in several practical issues. First, the join output table might be much larger than the base table, which causes 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, data scientists often perform exploratory analysis of different subsets of features and data [Zhang et al., 2014; Konda et al., 2013]. Materializing temporary tables after joins for learning on each subset could slow the data scientist and inhibit exploration [Anderson et al., 2013]. Learning *over* joins (physically), i.e., pushing ML computations through joins to the base tables, mitigates such drawbacks.

From a technical perspective, the issues caused by the redundancy present in denormalized datasets are well known in the context of traditional relational data management [Ramakrishnan and Gehrke, 2003]. But the implications of this type of redundancy in the context of ML 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 accuracy, learning efficiency, or scalability compared to the currently standard approach of learning after joins.

As a first step, in this project, we show that, for a large generic class of ML models called Generalized Linear Models (GLMs), it is possible to learn *over* joins and avoid redundancy without sacrificing accuracy 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 [Hastie et al., 2003; Mitchell, 1997]. We use standard gradient methods to learn GLMs: Batch Gradient Descent (BGD), Conjugate Gradient (CGD), and (L)BFGS [Nocedal and Wright, 2006]. 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. More information about GLMs and BGD is provided in Chapter 2.

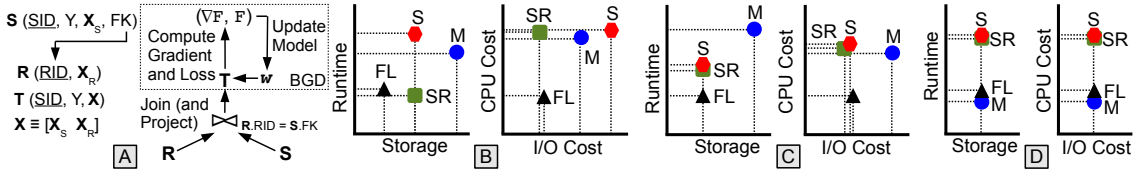


Figure 3.1: Learning over a join: (A) Schema and logical workflow. Feature vectors from **S** (e.g., `Customers`) and **R** (e.g., `Employers`) are concatenated and used for BGD. The loss (F) and gradient (∇F) for BGD can be computed together during a pass over the data. Approaches compared: Materialize (M), Stream (S), Stream-Reuse (SR), and Factorized Learning (FL). High-level qualitative comparison of storage-runtime trade-offs and CPU-I/O cost trade-offs 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.

Figure 3.1(A) gives a high-level overview of our problem using a two-table join. We first focus in-depth on a two-table join and generalize to multi-table joins in the end. In our customer churn example, this is the join of `Customers` with, say, `Employers`. We call the approach of materializing **T** before BGD as *Materialize*. We focus on the hybrid hash algorithm for the join operation [Shapiro, 1986]. 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 Shapiro [1986]. 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 accuracy. 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 [Gray et al., 1997; Feng et al., 2012].

The performance picture, however, is more complex. Figures 3.1(B-D) give a high-level qualitative overview of the trade-off 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 [Rendle, 2013]. 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, this project makes the following contributions:

- To the best of our knowledge, this is the first project to study the problem of learning over joins of large tables without materializing the join output. Focusing on GLMs solved using BGD, we explain the trade-off 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.
- 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 3.1, we explain our cost model and simple approaches to learn over joins. In Section 3.2, we present our new approach of *Factorized Learning* and its extensions. In Section 3.3, we discuss our experimental setup and results.

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)
$\|\mathbf{R}\|$	Number of **R** pages $\left(\left\lceil \frac{8n_R(1+d_R)}{p} \right\rceil\right)$
$\|\mathbf{S}\|$	Number of **S** pages $\left(\left\lceil \frac{8n_S(2+d_S)}{p} \right\rceil\right)$
$\|\mathbf{T}\|$	Number of **T** pages $\left(\left\lceil \frac{8n_S(1+d_S+d_R)}{p} \right\rceil\right)$
Iters	Number of iterations of BGD (≥ 1)

Table 3.1: 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 [Hastie et al., 2003].

3.1 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. Specifically, we contrast the current approach of materializing **T** and using it for BGD against our new approaches that avoid materializing **T** and instead, use **R** and **S** directly.

Assumptions and Cost Model

For the rest of this chapter, we focus only on the data-intensive computation in step 4 of the BGD algorithm (Algorithm 1) presented in Chapter 2, viz., the computation of (∇F, F) at each iteration. The data-agnostic computations of updating **w** are identical across all approaches proposed here, and typically take only a few seconds.[1] Tables 3.1 and 3.2

[1]CGD and (L)BFGS differ from BGD only in these data-agnostic computations, which are easily implemented in, say, Python, or R [Das et al., 2010]. If a line search is used to tune α, we need to compute only F, but largely the same trade-offs apply.

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 3.2: 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.

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 $(m-1) > \sqrt{\lceil f|\mathbf{R}|\rceil}$ [Shapiro, 1986]. We also focus primarily on the case $n_S > n_R$ and $|\mathbf{S}| \geqslant |\mathbf{R}|$. We discuss the cases $n_S \leqslant n_R$ or $|\mathbf{S}| < |\mathbf{R}|$ in the appendix.

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 Shapiro [1986], we now introduce some notation. The number of partitions of **R** is $B = \lceil \frac{\lceil f|\mathbf{R}|\rceil - (m-2)}{(m-2)-1} \rceil$. Partition sizes are $|\mathbf{R}_0| = \lfloor \frac{(m-2)-B}{f} \rfloor$, and $|\mathbf{R}_i| = \lceil \frac{|\mathbf{R}|-|\mathbf{R}_0|}{B} \rceil (1 \leqslant i \leqslant B)$, 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 [Shapiro, 1986]. 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 and for the sake of better readability, we present their costs in the appendix.

I/O Cost If $(m-1) \leqslant \lceil f|\mathbf{R}|\rceil$, 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
  ]
```

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.

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.

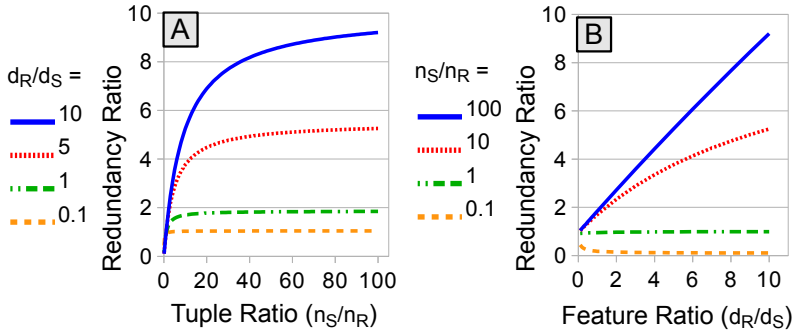


Figure 3.2: Redundancy ratio against the two dimension ratios (for $d_S = 20$). (A) Fix $\frac{d_R}{d_S}$ and vary $\frac{n_S}{n_R}$. (B) Fix $\frac{n_S}{n_R}$ and vary $\frac{d_R}{d_S}$.

Discussion of Trade-offs The I/O and storage trade-offs between Materialize and Stream (Figure 3.1(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 3.2 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 3.2(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 3.2(B), the redundancy ratio increases with the feature ratio, and converges to the tuple ratio $\frac{n_S}{n_R}$.

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.

3.2 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 decomposing the computations of both F and ∇F and "pushing them down through the join." We call our technique *factorized learning* (Factorize, or FL for short), borrowing the terminology from "factorized" databases [Bakibayev et al., 2013]. An overview of the logical computations in FL is presented in Figure 3.3.

The key insight in FL is as follows: given a feature vector $\mathbf{x} \in \mathbf{T}$, $\mathbf{w}^T\mathbf{x} = \mathbf{w}_S^T\mathbf{x}_S + \mathbf{w}_R^T\mathbf{x}_R$. 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 3.3).

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 3.3).
2. Recall that F and ∇F are computed 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, 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).
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 3.3).

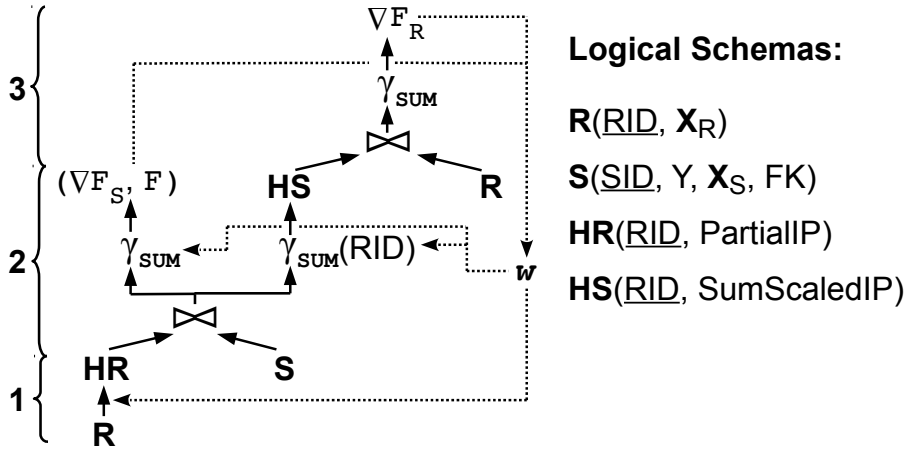


Figure 3.3: Logical workflow of factorized learning, consisting of three steps as numbered. **HR** and **HS** are logical intermediate relations. PartialIP refers to the partial inner products from **R**. SumScaledIP refers to the grouped sums of the scalar output of $G()$ applied to the full inner products 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.

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, $\mathbf{w}$ 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 3.2.1. *The output* $(\nabla F, F)$ *of FL is identical to the output* $(\nabla F, F)$ *of both Materialize and Stream.*[2]

While it is straightforward to translate the logical plan shown in Figure 3.3 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

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

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 3.3) with 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 3.3:

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, this is 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** using RID, compute partial gradients on each example, and update ∇F_R in memory. Essentially, this is 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 trade-off. Similarly, FL requires an extra scan of **R** per iteration, which introduces a non-obvious I/O trade-off. As we will show later in Section 5, these trade-offs 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 $|\mathbf{H}| = \lceil \frac{f.n_R.(1+2).8}{p} \rceil$, which is only $O(n_R)$. Thus, in many cases, **H** will probably easily fit in buffer memory. Nevertheless, to handle cases in which **H** does not fit in buffer memory, we present a few extensions to FL.

Scaling FL along n_R

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

FLSQL

FLSQL applies the traditional approach of optimizing SQL aggregates (e.g., `SUM`) over joins using logical query rewriting [Yan and Larson, 1995; Chaudhuri and Shim, 1994]. Instead of maintaining **H** in memory, it directly converts the logical plan of Figure 3.3 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.

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 **S** twice:

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

Note that **HR** is of size $O(n_R)$ but **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.

FL-Partition (FLP)

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

1. Partition **R** and **S** on RID (resp. FK) 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 **R** and **S**, similar to Stream-Reuse.

Note that we could even partition **S** and **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 3.2.2. *The output* $(\nabla F, F)$ *of FLP, FLSQL, and FLSQL+ are all identical to the output* $(\nabla F, F)$ *of FL.*

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.

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 [Selinger et al., 1979; Apache, a].

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 $\mathbf{R}_i$, converts it to its associative array $\mathbf{HR}_i$ (step 1 of FL), and then applies a simplified GRACE hash join [Shapiro, 1986] recursively on the right-deep join tree with $\mathbf{S}$ as the outer table.[3] 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 $s_i \leqslant |\mathbf{R}_i|$). 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

[3] 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 [Shapiro, 1986].

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:

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

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 3.1. *FL-MULTJOIN is* NP*-Hard in* l*, where* $l = |\{i | m - k \geqslant |\mathbf{HR}_i| > 1\}| \leqslant k$.

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 [Garey and Johnson, 1990]. We present the proof in the appendix. We adopt a standard $O(k \log(k))$ time greedy heuristic for the knapsack problem to solve FL-MULTJOIN approximately [Dantzig, 1957]. 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.

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* [Gray et al., 1997]. 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 [Apache, a] (and most parallel RDBMSs), we can directly leverage existing parallelism infrastructure [Feng et al., 2012].

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 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 $\mathbf{H}$ 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.

3.3 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 trade-off 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 [Agarwal et al., 2014]. 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 work 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 on GitHub: `https://github.com/arunkk09/orion`.

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 trade-off space. Figure 3.4 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 3.4. 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

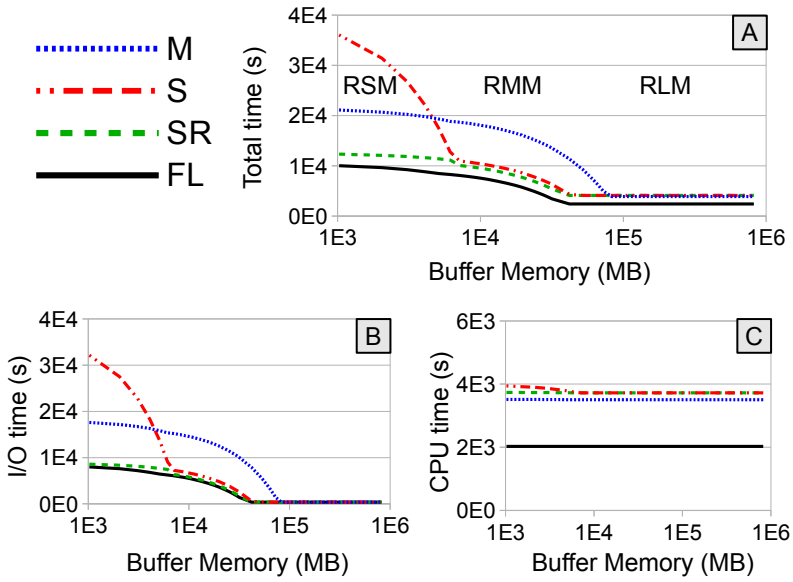


Figure 3.4: Analytical cost model-based plots for varying the buffer memory (m). (A) Total time. (B) I/O time (with 100MB/s I/O rate). (C) CPU time (with 2.5GHz clock). The values fixed are $n_S = 10^8$ (in short, 1E8), n_R = 1E7, $d_S = 40$, $d_R = 60$, and $Iters = 20$. Note that the x axes are in logscale.

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 3.4(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.

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 [Nocedal

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 3.3: Parameters used for the single-node setting in Figure 3.5. NB: $xEy \equiv x \times 10^y$.

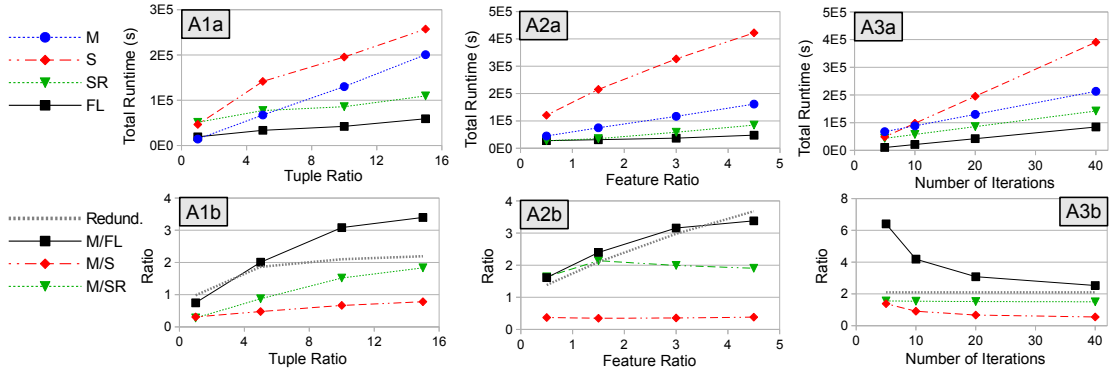
and Wright, 2006; Feng et al., 2012]. Finally, we assess whether our cost models are able to accurately predict the observed performance trends and discuss some practical implications.

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 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.

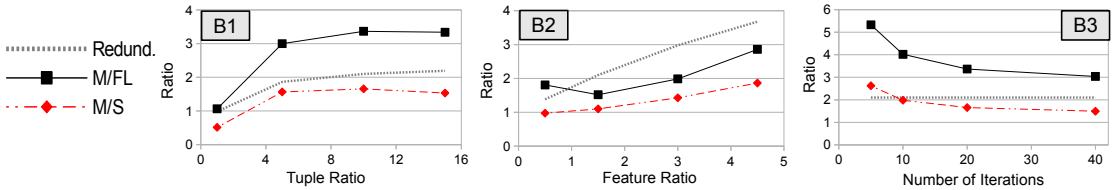
Results

The implementation-based runtimes, speedup ratios, and redundancy ratios for RSM are plotted in Figure 3.5(A). The corresponding parameter values chosen (for those parameters that are not varied) are presented in Table 3.3. We present the corresponding speedup ratios for RMM in Figure 3.5(B) and for RLM in Figure 3.5(C). For the sake of readability, we skip the runtime plots for RMM and RLM here and present them in the appendix instead.

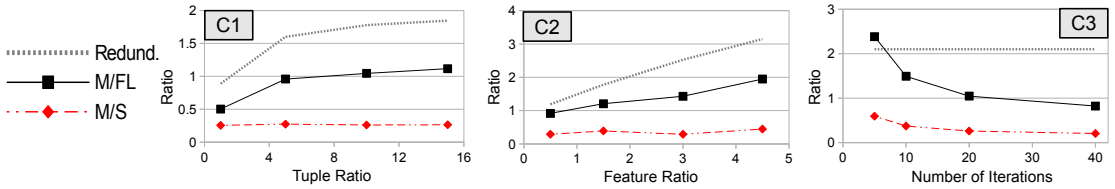
- **RSM:** The plots in Figure 3.5(A) show that S is the slowest in most cases, followed by the state-of-the-art approach, M. SR is significantly faster than M, while FL is the fastest. This trend is seen across a wide range of values for all 3 parameters – tuple ratio, feature ratio, and number of iterations. All the approaches seem to scale almost linearly with each parameter. Figure 3.5(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



(A) RSM: Runtimes and speedups against M and redundancy ratios.



(B) RMM: Speedups against M and redundancy ratios.



(C) RLM: Speedups against M and redundancy ratios.

Figure 3.5: 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 ($\mathtt{Iters}$) – 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 3.3.

mostly higher than the redundancy ratio (r) – this is because the cost of materializing $\mathbf{T}$ is ignored by r. Figure 3.5(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 3.5(B) show that S could become faster than M, as predicted by Figure 3.4. SR (skipped here for brevity) is roughly as fast as S, since no partitioning occurs. FL is the fastest in most cases, but interestingly, Figure 3.5(B2) shows that the speedup of FL over M is lower than the redundancy ratio for larger feature ratios.

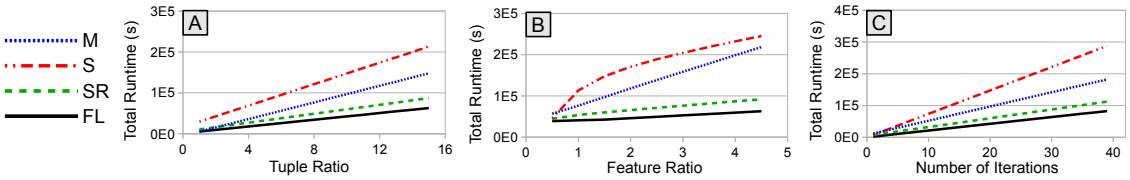


Figure 3.6: Analytical cost model-based plots of performance against each of (A) $\frac{n_S}{n_R}$, (B) $\frac{d_R}{d_S}$, and (C) $Iters$ for the RSM region. The other parameters are fixed as per Table 3.3.

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 3.5(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 3.5(C1,C2) show that M is faster than FL at low dimension ratios (but slower at higher ratios). Figure 3.5(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 for different parameter settings. Thus, we now verify if our models predicted the trends correctly. Figure 3.6 presents the performance results predicted by our cost models for the RSM region. A comparison of the respective plots of Figure 3.5(A) with Figure 3.6 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 for the sake of readability, 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. Table 3.4 presents the discrete prediction accuracy,

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 3.4: 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 3.5: 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 3.6: Mean / median percentage error for predicting runtimes.

i.e., how often our cost model predicted the fastest approach correctly for each experiment and memory region corresponding to the results of Figure 3.5. This quantity matters because a typical cost-based optimizer uses a cost model only to predict the fastest approach. Table 3.5 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 3.6 presents the mean and median percentage error in the predictions.

We find that, overall, our cost model correctly predicts the fastest approach in 95% of the cases shown in Figure 3.5. 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%. 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 [Zhang et al., 2014; Konda et al., 2013] and MLBase [Kraska et al., 2013] 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 accuracy criterion for any gradient method. While Figures 3.5(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.

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.

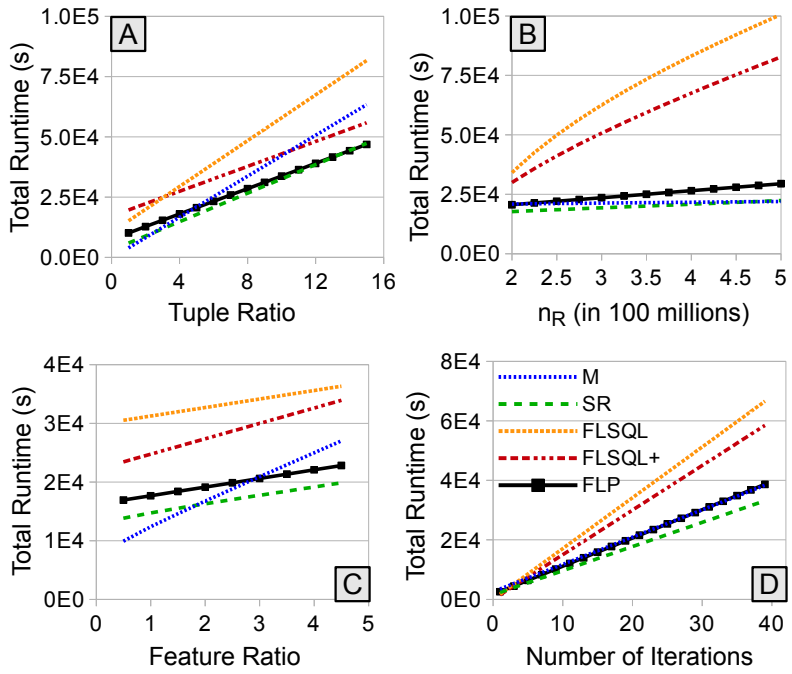


Figure 3.7: Analytical plots for when m is insufficient for FL. We assume m = 4GB, and plot the runtime against each of n_S, d_R, Iters, and n_R, while fixing the others. Wherever they are fixed, we set $(n_S, n_R, d_S, d_R, \text{Iters}) = (1E9, 2E8, 2, 6, 20)$.

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 = 24GB, FL can scale up to $n_R \approx 750E6$. Attribute tables seldom have so many tuples (unlike entity tables). Hence, we use a smaller value of m = 4GB. We also vary n_R for this comparison. Figure 3.7 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 surprisingly, 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.

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 $O(k\log(k))$ time greedy

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 3.7: 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 $n_S = 2E8$ and $d_S = 10$, while n_i and d_i ($i > 0$) range from 1E7 to 6E7 and 35 to 120 respectively. We set $m = 4GB$.

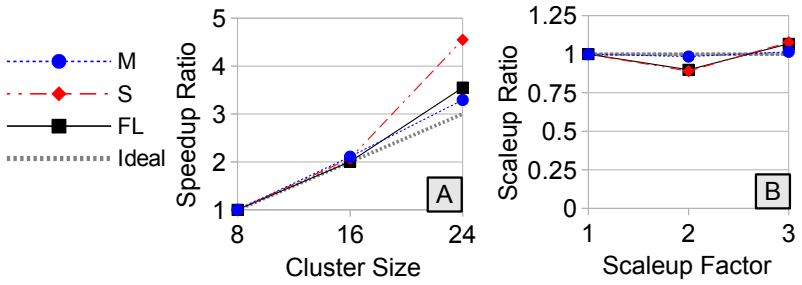


Figure 3.8: Parallelism with Hive. (A) Speedup against cluster size (number of worker nodes) for $(n_S, n_R, d_S, d_R, Iters) = (15E8, 5E6, 40, 120, 20)$. Each approach is compared to itself, e.g., FL on 24 nodes is 3.5x faster than FL on 8 nodes. The runtimes on 24 nodes were 7.4h for S, 9.5h for FL, and 23.5h for M. (B) Scaleup as both the cluster and dataset sizes 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.

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 3.7 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.

Shared-nothing Parallelism

We compare our Hive implementations of M, S, and FL.[4] Our goal is to verify if similar runtime trade-offs 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.[5] Figure 3.8 presents the results.

Figure 3.8(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 trade-offs 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 predict the trends. We consider this as an interesting avenue for future work. Figure 3.8(B) shows that all approaches achieve near-linear scaleups. Overall, we see that similar runtime trade-offs as for the RDBMS setting apply, and that our approaches achieve near-linear speedups and scaleups.

3.4 Conclusion: Avoiding Joins Physically

Key-foreign key joins are often required prior to applying ML on multi-table 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 joins in order to avoid such redundancy. Focusing on generalized linear

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

[5]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.

models solved using batch gradient descent, we propose several alternative approaches to learn over joins that are also easy to implement over existing data processing systems. We introduce a new approach named factorized learning that pushes the ML computations through joins and avoids redundancy in both I/O and computations without affecting ML accuracy. 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. This work opens up a new problem space for studying the interplay between ML and relational operations, especially joins, in the context of feature engineering.

Most of the content of this chapter is from our paper titled "Learning Generalized Linear Models Over Normalized Data" that appeared in the ACM SIGMOD 2015 conference. The code for our system is open source and is available on GitHub: `https://github.com/arunkk09/orion`.

A sum of sums is the same sum,
Factorizing avoids repeating some.
So, learn after joins no more,
Do learn over joins herefore!

---

4 Extensions and Generalization of Orion

In this chapter, we extend the idea of avoiding joins physically, specifically the technique of factorized learning to several other classes of popular ML models: probabilistic classifiers, GLMs with non-batch optimization methods, and clustering algorithms. We also generalize it using linear algebra. These are natural follow-on projects to Project Orion in which we introduced the idea of learning over joins for GLMs with BGD. These projects establish the wide applicability and generality of our idea.

4.1 Extension: Probabilistic Classifiers Over Joins and Santoku

In this project, we extend the idea of factorized learning (FL) to probabilistic classifiers such as Naive Bayes, Tree-Augmented Naive Bayes, and Decision Trees. We also introduce the technique of *factorized scoring* (FS), which "factorizes" the computations of scoring, i.e., computing the accuracy of a learned ML model on a test set. We build a toolkit named Santoku that provides implementations of factorized learning and scoring for probabilistic classifiers and logistic regression to make it easier for data scientists to adopt our ideas in practice. We now explain briefly how factorized learning works for Naive Bayes with an example.

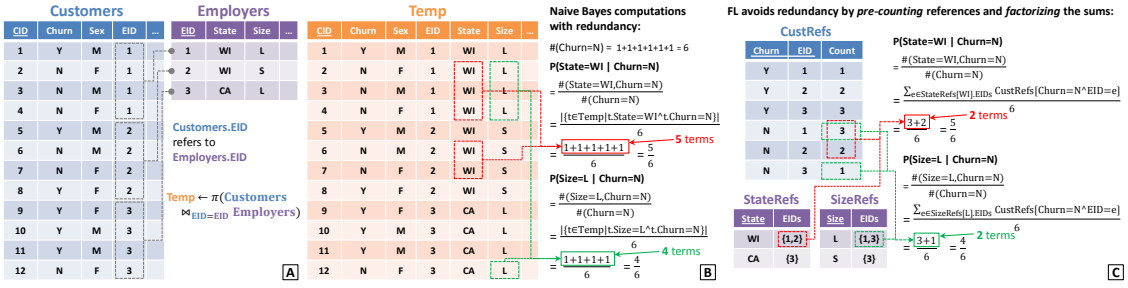


Figure 4.1: Illustration of Factorized Learning for Naive Bayes. (A) The base tables `Customers` (the "entity table" as defined in Kumar et al. [2015c]) and `Employers` (an "attribute table" as defined in Kumar et al. [2015c]). The target feature is `Churn` in `Customers`. (B) The denormalized table `Temp`. Naive Bayes computations using `Temp` have redundancy, as shown here for the conditional probability calculations for `State` and `Size`. (C) FL avoids computational redundancy by pre-counting references, which are stored in `CustRefs`, and by decomposing ("factorizing") the sums using `StateRefs` and `SizeRefs`.

Figure 4.1(A) shows a simple instance of our customer churn example. The output of the join, `Temp`, has redundancy in the features from `Employers`, e.g., values of `State` and `Size` get repeated more often. This results in redundancy in the computations for Naive Bayes when it counts occurrences to estimate the conditional probabilities for those features. Figure 4.1(B) illustrates the additions needed for both `State=WI` and `Size=L` when operating over `Temp`. In contrast, FL avoids redundant computations by pre-computing the number of foreign key references, and by factoring them into the counting. Figure 4.1(C) illustrates the reference counts that are temporarily stored in `CustRefs`, which is obtained, in SQL terms, using a `GROUP BY` on `Churn` and `EID` along with a `COUNT`. The list of `EID` values for `State=WI` (and `Size=L`) are also obtained. Thus, we can reduce the sums for those features into smaller sums, viz., 2 terms instead of 5 for `State=WI`, and 2 instead of 4 for `Size=L`. Usually, the learned ML models are also "scored," i.e., their prediction accuracy is validated with a set of test examples. For Naive Bayes, this requires us to compute the *maximum a posteriori* (MAP) estimate on a given test feature vector $\mathbf{x}$ as follows: $\mathrm{argmax}_{y \in \mathcal{D}_Y} P(Y = y) \Pi_{F \in \mathbf{X}} P(F = \mathbf{x}_{(F)} | Y = y)$ [Mitchell, 1997]. Essentially, scoring involves a multiplication of conditional probabilities. Thus, we can exploit the redundancy in scoring as well, not just learning, by factorizing the computations on a test set and pushing them through the joins. We call this technique *factorized scoring*, and we use it in Santoku when the models need to be validated.

In addition to providing a library of implementations of FL and FS, Santoku handles the following three tasks. We explain each task in a bit more detail.

1. Determining if FL/FS would be faster on a given input using cost models.
2. Allowing the user to specify functional dependencies (FDs) on a given single (denormalized) table and using the FDs to make it possible to apply FL/FS.
3. Helping data scientists compare feature subsets from different tables to apply our idea of avoiding joins logically (explained in Chapter 5).

While FL avoids redundant computations, it performs extra work for "book-keeping." As we showed in Project Orion, this means that FL could be slower than using the denormalized dataset for some inputs depending on various parameters of the data, system, and ML model [Kumar et al., 2015c]. It will be helpful for analysts if a system could *automatically* decide which tables to join and which to apply FL on in order to optimize the performance of the ML model over normalized data. Santoku *provides such an optimization capability by using a simple cost model and a cost-based optimizer.*

A closely related scenario is learning over a table with functional dependencies (FDs) between features. For example, we can view $\mathbf{T}$ as having the following FD: $\mathrm{FK} \rightarrow \mathbf{X}_R$. This

FD is a result of the key-foreign key join.[1] In general, there could be many such FDs in a denormalized table. From speaking to analysts at various companies, we learned that they ignore such FDs altogether because their ML toolkits cannot handle them. While they may not use the database terminology (FD), analysts recognize that such "functional relationships" can exist among features. One can use the FDs to normalize the single table, and then apply FL to different degrees. But once again, these approaches could be slower than using the single table on some inputs. Santoku *enables analysts to integrate such FD-based functional relationships into some popular ML models and automatically optimizes performance.*

Finally, we consider the important related task of *feature selection*. It is often a tedious exploratory process in which analysts evaluate smaller feature vectors for their ML model to help improve accuracy, interpretability, etc. [Anderson et al., 2013; Konda et al., 2013]. Ignoring FD-based functional relationships could mean ignoring potentially valuable information about what features are "useful." Santoku *helps analysts exploit FD-based functional relationships for feature selection purposes.* Santoku provides a "feature exploration" option that automatically constructs and evaluates smaller feature vectors by dropping different combinations of sides of FDs. In our customer churn example, one can drop `EmployerID` (perhaps an *uninterpretable* identifier), or other features from **`Employers`**, or both. While this may not "solve" feature selection fully, it provides valuable *automatic insights* using FDs that could help analysts with feature selection.

Santoku is designed as an open-source library usable in R, which is a powerful and popular environment for statistical computing. R provides easy access to a large repository of ML codes. [2] By open-sourcing our API and code, we hope to encourage contributions from the R community that extend Santoku to more ML models. Many data management companies such as EMC, Oracle, and SAP have also released products that scale R scripts *transparently* to larger-than-memory data. We implement Santoku fully in the R language, which enables us to exploit such R-based analytics systems to provide scalability automatically. For users that do not want to write R scripts, Santoku also provides an easy-to-use GUI, as illustrated in Figure 4.2.

System architecture

We now discuss the system architecture of Santoku, presented in Figure 4.3, and explain how it fits into a standard advanced analytics ecosystem.

[1] Key-foreign key dependencies are not FDs, but we can view them as such with some obvious assumptions.
[2] `http://cran.r-project.org/`

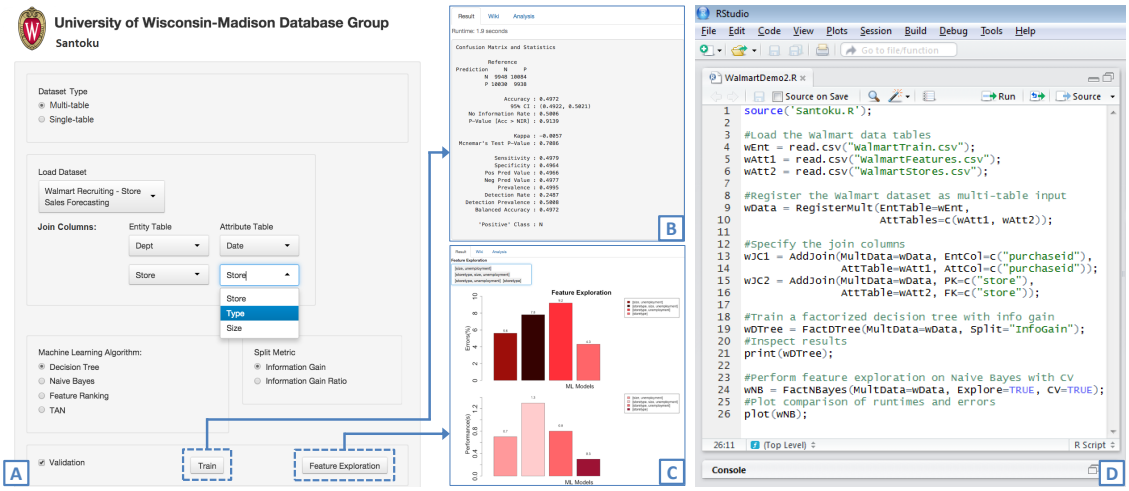


Figure 4.2: Screenshots of SANTOKU: (A) The GUI to load the datasets, specify the database dependencies, and train ML models. (B) Results of training a single model. (C) Results of feature exploration comparing multiple feature vectors. (D) An R script that performs these tasks programmatically from an R console using the SANTOKU API.

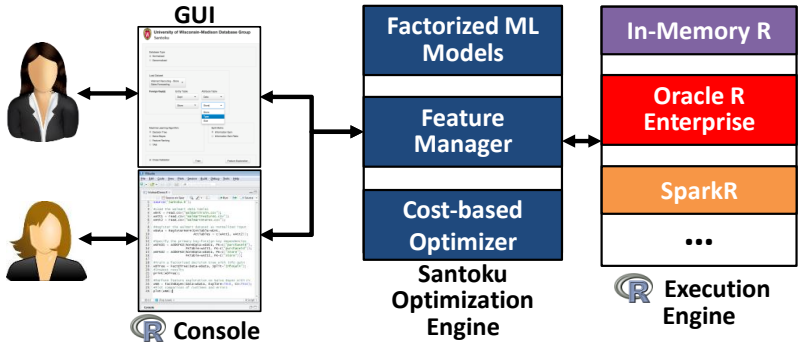


Figure 4.3: High-level architecture. Users interact with SANTOKU either using the GUI or R scripts. SANTOKU optimizes the computations using factorized learning and invokes an underlying R execution engine.

Front-end SANTOKU provides custom front-ends for two kinds of data scientists – those who prefer a graphical user interface (GUI), and those who prefer to write R scripts. The GUI is intuitive and has three major portions (Figure 4.2(A)). The first portion deals with the data: data scientists can specify either a multi-table (normalized) input or a single-table (denormalized) input. For normalized inputs, the data scientist specifies the base tables and the "join columns," i.e., the features on which the tables are joined (the foreign keys and primary keys in database parlance) using menus. For denormalized inputs, the data scientist specifies the single table and any "functional relationships" among the features

(the left and right sides of FDs in database parlance) using menus. The second portion deals with the ML model: they choose a model and its parameters, and can either train a single model or perform feature exploration, possibly with validation. The third portion displays the results: a summary of the execution for training (Figure 4.2(B)), and plots comparing several feature vectors for feature exploration (Figure 4.2(C)). Interestingly, we were able to implement Santoku's GUI in R itself using its graphics and visualization libraries. The GUI is rendered in a browser, which makes it portable. Santoku also provides an intuitive API that can be used in R scripts (Figure 4.2(D)). This enables analysts to exploit Santoku's factorized ML models programmatically. The operations on Santoku's GUI also invoke this API internally.

Santoku Optimization Engine The core part of Santoku is its optimization engine, which has three components. The first component is a library of R codes that implement factorized learning and scoring for a set of popular ML techniques – Naive Bayes, Tree-Augmented Naive Bayes (TAN), feature ranking, and decision trees Mitchell [1997] as well as logistic regression using BGD Kumar et al. [2015c]. We adapted the implementations of these models from standard R packages on CRAN. We expect to add more as our system matures. The second component is the *feature manager*. It manipulates the feature vectors of the datasets. It handles three major tasks: normalization of single tables using FDs, denormalization by joining multiple tables, and constructing the alternative feature vectors for feature exploration. The third component is a cost-based optimizer that uses a cost model to determine whether or not to use factorized learning and scoring on a given input (given by the analyst, or constructed internally as part of feature exploration). The cost model is calibrated based on the R execution engine.

Back-end Since Santoku is implemented in R, it simply "piggybacks" on existing R execution engines, with the standard in-memory R perhaps being the most popular. Several commercial and open-source systems scale R to different data platforms, e.g., Oracle R Enterprise operates over an RDBMS (and Hive), while SparkR operates over the Spark distributed engine. Such systems enable Santoku to automatically scale to large datasets.

Santoku is joint work with a BS student, Boqun Yan, and an MS student, Mona Jalal, along with Jeff Naughton and Jignesh Patel. My contribution was in the conceptualization of the system, parts of the implementation, and advising the other students through the rest of the implementation. Most of the content of this section is from our paper titled "Demonstration of Santoku: Optimizing Machine Learning over Normalized Data" that appeared in the demonstration track of the VLDB conference in 2015 [Kumar et al.,

2015a]. All of our code and the real datasets used in this project are available on GitHub: `https://github.com/arunkk09/santoku`.

4.2 Extension: Other Optimization Methods Over Joins

In Project Orion, we focused on GLMs solved using the optimization method known as Batch Gradient Descent (BGD). In practice, on large datasets, BGD often takes more iterations to converge than *stochastic* optimization methods such as Stochastic Gradient Descent (SGD) and Stochastic Coordinate Descent (SCD) [Nocedal and Wright, 2006]. SGD is increasingly popular for ML over large datasets [Feng et al., 2012; Lin and Kolcz, 2012], while SCD is also popular if the features are dense [Friedman et al., 2010]. Thus, in this project, we study if the idea of learning over joins is extensible to GLMs solved using SGD and SCD. If so, we would like to understand how the trade-off space for SGD and SCD over joins differs from that of BGD over joins.

The first observation is that the data access patterns of SGD and SCD are significantly different from that of BGD and we explain their data access patterns briefly next. Focusing on the materialized scenario with a single table $\mathbf{T}$, BGD performs one sequential scan of $\mathbf{T}$ per iteration to compute the gradient, which is akin to a `SUM` aggregation in SQL. BGD then updates the model $\mathbf{w}$ and moves to the next iteration. In contrast, SGD updates the model after *each* example in $\mathbf{T}$ by approximating the full gradient with the gradient computed on a single example at a time. Typically, SGD samples the data examples *with* replacement but practical implementations on large datasets use sampling *without* replacement. This requires a shuffle of the dataset, typically implemented using a sort in data processing systems such as RDBMSs, e.g., using an `ORDER BY RANDOM()` in PostgreSQL [Feng et al., 2012]. Thus, at each iteration, SGD shuffles $\mathbf{T}$ and performs a sequential scan. SCD, on the other hand, updates $\mathbf{w}$ one co-efficient (coordinate) at a time. That is, SCD computes the portion of the gradient along one feature at a time, while fixing all the other co-efficients. SCD then cycles through all the features iteratively. Thus, SCD has a column-wise access pattern in contrast to BGD's (and SGD's) row-wise access pattern. SCD typically needs to visit the coordinates in random order at each iteration. Next, we explain in brief about how we can learn GLMs using SGD over joins, followed by SCD over joins.

SGD Over Joins We start by noting a key distinction between BGD and SGD over the materialized table $\mathbf{T}$: BGD has computational redundancy because $\mathbf{X}_\mathrm{R}$ values are repeated. However, SGD has no computational redundancy because $\mathbf{w}$ changes after each example. Thus, the only advantage of learning over joins with SGD is in reducing the cost of data access, e.g., I/O. The approach we consider is similar to Stream for BGD over joins: perform

the join lazily by building a hash table over the inner table **R**, and do a sequential pass over the table **S** (after shuffling it). For each tuple in **S**, we look up into the hash table over **R** and obtain the entire feature vector. However, this approach faces a subtle issue when the hash table over **R** does not fit in buffer memory. The regular hash join would end up splitting both **S** and **R** into chunks based on the FK (= RID). This hash partitioning means that the *ordering* of the examples in **S** after the shuffle would get scrambled. Thus, the final w obtained after a pass will deviate from what we would get with the materialized approach. This is a new runtime-accuracy trade-off for SGD over joins. In this project, we explore this trade-off in depth by studying the effects of the relaxed ordering on the convergence of SGD. We implement our approaches on PostgreSQL by combining Project Orion and Feng et al. [2012]. Using both synthetic and real datasets, we find empirically that as long as we shuffle the examples within each chunk of **S**, SGD still converges in roughly the same number of iterations. This means SGD over joins could still provide significant runtime benefits over SGD after joins. One complicating issue that could slow convergence is if all the examples in a chunk have the same class label, SGD takes slower to converge; this is similar to the CA-TX issue noticed in Feng et al. [2012]. Empirically, we find that this is made up for by the runtime savings per iteration. Ongoing work includes more experiments with real and synthetic datasets, a more formal understanding the effects of relaxed ordering on SGD convergence, and experiments with Mini-batch SGD, which is a hybrid of SGD and BGD.

SCD Over Joins Unlike SGD, SCD over **T** has both computational and I/O redundancy. However, the redundancy arises only when a feature (column) from **R** is accessed. Owing to the column-wise access pattern of SCD, we consider implementing it in a column store. We assume that each column of **T** fits in buffer memory (but not necessarily the whole table). We consider both a Stream-style approach and a Factorize-style approach. First, we obtain a tuple-tuple physical mapping based on FK and RID. Given a tuple from **S**, this FK-RID mapping directly tells us which tuple in **R** to look up. The Stream approach uses this mapping to reconstruct the denormalized column for each feature in **R** for every iteration, i.e., it performs the join lazily. Thus, it saves I/O redundancy but it still has computational redundancy. The Factorize approach, however, avoids performing the join even lazily. Instead, it constructs an associative array to pre-compute the contribution to the gradient for each tuple in **R** and then performs a `GROUP BY`-style aggregate on the FK-RID mapping to scale and add the entries in the associative array. Thus, similar to the Factorize technique for BGD, this approach for SCD avoids computational redundancy as well.

Empirically, using synthetic datasets and a file-base implementation[3] of all approaches, we find that when **T** does not fit in memory, both Stream and Factorize provide significant runtime savings. But when **T** does fit in memory, Stream becomes slower, similar to the result for BGD in Project Orion. Interestingly, the runtimes of Factorize are comparable to that of Materialize in this setting. In fact, in some cases, Materialize is slightly faster than Factorize for SCD for the same dataset wherein Materialize was slower than Factorize for BGD. This is because the savings in the computations are relatively less significant for SCD compared to BGD. This underscores the importance of the cost model. Of course, Factorize offers non-runtime advantages too, e.g., lower storage and easier maintenance. Ongoing work includes more experiments with real and synthetic datasets and experiments with Block Coordinate Descent, which is a hybrid of SCD and BGD.

This project is joint work with three BS students, Boqun Yan, Zhiwei Fan, and Fujie Zhan, along with Jeff Naughton, Jignesh Patel, and Steve Wright. Papers on this project are under preparation. My contribution was in the conceptualization of this project and advising the other students throughout the algorithmic and experimental details.

4.3 Extension: Clustering Algorithms Over Joins

In this project, we extend our idea of learning over joins to clustering algorithms, some of which have significantly different data access patterns than GLMs. Clustering is a popular approach to unsupervised ML in which the data examples do not need to be labeled. Clustering algorithms have diverse applications ranging from bioinformatics and medical imaging to recommendation systems and social network analysis [Aggarwal and Reddy, 2013]. However, going one step further, we also consider one limitation of factorized learning in this project: factorized learning needs prior knowledge of the normalized schema, i.e., the database dependencies. From conversations with data scientists at various enterprise and Web companies, we learned that in practice, data scientists sometimes use the denormalized data directly because ML toolkits do not typically track database dependencies, which could limit the practical applicability of factorized learning.

This project aims to mitigate the above issues by taking a first principles approach to optimizing popular clustering algorithms over denormalized data. We ask: *How should we exploit the redundancy in denormalized data to optimize clustering algorithms without modifying the output clusters?* For the sake of tractability, we focus on three popular and representative clustering algorithms that differ substantially in their complexity and data access pattern:

[3]We tried implementing our ideas in the popular column store RDBMS MonetDB first. But we were not able to get its extensibility mechanisms to add user-defined functions for SCD to work.

K-Means, Hierarchical Agglomerative Clustering (HAC), and DBSCAN. We refer the interested reader to Aggarwal and Reddy [2013] for more details about these algorithms. We introduce two alternative approaches to optimize these clustering algorithms over denormalized data: *factorized* clustering and *compressed* clustering. We explain each of these approaches in brief next.

Factorized Clustering Assuming that the normalized schema is available (or is cheap to obtain), we show that it is possible to push the three clustering algorithms down through joins and factorize them. However, the algorithms differ in the precise techniques needed to factorize them and we explain the trade-offs involved. At a high-level, factorizing K-Means involves staging the computations of the distances between data points and centroids, while factorizing HAC involves staging the computations of distances between every pair of data points. However, factorized learning for DBSCAN works in a "lazy" fashion in which computations are factorized incrementally. We also present some modifications to our factorized clustering algorithms to improve performance.

Compressed Clustering We also consider cases in which the normalized schema is either not available or is too expensive to obtain. We consider applying compression to this case, since compression has long been used by RDBMSs to reduce storage, while processing SQL queries on compressed data has been shown to improve performance [Abadi et al., 2006; Chen et al., 2001]. Surprisingly, our initial results with the popular Lempel-Ziv-Welch (LZW) compression technique showed that it is often slower than directly clustering the denormalized dataset. We found that the cause was that LZW "out of the box" uses codes that are too short to capture the redundancy in feature vectors effectively. Thus, we present new optimizations to LZW compression that exploit structural properties of the redundancy present in denormalized datasets and that are tailored towards the row-wise access pattern of the clustering algorithms. We also introduce a heuristic to prevent the algorithm from creating too many short codes to avoid having to decode potentially non-repeating features on the fly. We then integrate the three clustering algorithms with our new compressed representation to avoid explicit decompression costs.

We perform an extensive empirical analysis with both synthetic and real datasets to compare the current state-of-the-art approach of using the denormalized data ("materialized" clustering) against both factorized and compressed clustering. The trends in the results differ across the three clustering algorithms but overall, we observe that for a wide range of data sizes, both factorized and compressed clustering significantly outperform their respective materialized versions when the joins introduce redundancy. Compressed

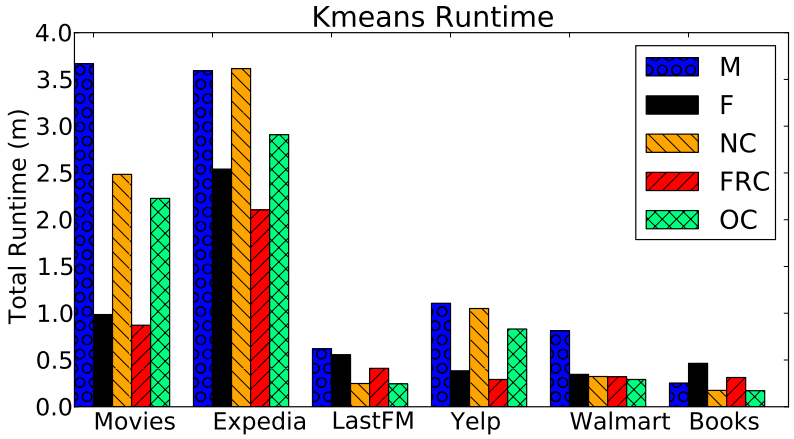


Figure 4.4: Results on real datasets for K-Means. The approaches compared are – M: Materialize (use the denormalized dataset), F: Factorized clustering, FRC: Factorized clustering with recoding (improves F), NC: Naive LZW compression, and OC: Optimized compression (improves NC).

clustering often yields comparable performance to factorized clustering but it is sometimes slower than factorized clustering. However, as the amount of "non-schematic" redundancy in the data (the redundancy present even before joining) increases, we find that compressed clustering becomes faster than factorized clustering. Figure 4.4 presents a snapshot of the results for K-Means on the real datasets from Kumar et al. [2016] (Chapter 5). The speed-up of factorized clustering over materialized clustering was up to 4.2x, while the speed-up for compressed clustering was up to 2.8x. Overall, our experiments validate that irrespective of whether the normalized schema is available or not, it is often possible to significantly improve the performance of clustering algorithms over denormalized data.

This project is joint work with two MS students, Fengan Li and Lingjiao Chen, along with Jeff Naughton and Jignesh Patel. It is under submission to a conference. My contribution was in the conceptualization of this project and advising the other students throughout the algorithmic and experimental details.

4.4 Generalization: Linear Algebra Over Joins

While factorized learning helps avoid redundancy in ML computations by pushing them through joins, it requires a developer to manually rewrite the ML algorithm's code, say, in R (as we did in Project Santoku), or on top of a data processing system (as we did in Project Orion). This raises an important question: *Do we really need to manually reimplement*

each ML algorithm in order to extend the benefits of factorized learning to other ML algorithms, or is there a way to mitigate this development overhead? Suppose there is a formal language that can represent many (if not all) ML algorithms. If we can "factorize" such a language directly, then all the representable ML algorithms will be *automatically* factorized in one go, which mitigates the above development overhead. Thus, our question now becomes: *Is there such a formal language?*

We find our answer in the recent trend to provide more systems support for R. Several projects focus on improving data processing in R [Sridharan and Patel, 2014] and integrating R (or R-like languages) with data processing systems [Zhang et al., 2010; Ghoting et al., 2011; Oracle; Apache, c]. The core of R is *linear algebra*, an elegant formal language to express a wide variety of ML algorithms ranging from classification and regression to clustering and feature extraction [Ghoting et al., 2011]. The ML code is written using basic and derived *operators* in linear algebra, e.g., matrix-vector multiplication. Many database vendors provide lower-level code to translate these operators into queries over an RDBMS, Hive/Hadoop, or Spark. This dramatically reduces the development overhead for implementing many ML algorithms. Alas, such systems still assume that the input *feature matrix* is a single table. When the dataset is multi-table, analysts have to join and materialize a single table for their linear algebra scripts.

In this project, we take a step towards *factorizing linear algebra*, i.e., pushing linear algebra operators through joins. We extend linear algebra by introducing a new logical data type, the *normalized matrix*, to represent normalized (multi-table) datasets. We then devise an extensive framework of *algebraic rewrite rules* to convert key basic and derived operators in linear algebra over a regular denormalized matrix into operations over the normalized matrix, i.e., the pre-join input. For example, a matrix-vector multiplication – common in ML algorithms – is rewritten into a set of operations over the normalized matrix that yield the same output.

A key advantage of our framework is *closure* with respect to linear algebra, i.e., a linear algebra script is rewritten only to a different linear algebra script. This could make it easy to integrate our framework with R and R-based systems without needing to modify their internals. For concreteness sake, we implement our framework in standard main-memory R as a library although our framework is generic and applicable to any other linear algebra system such as R-based analytics systems and Matlab. This enables us to piggyback on any scalability, parallelism, or other performance-related improvements made by such systems. Overall, our framework helps obviate the need for developers to rewrite ML code from scratch to integrate factorized learning. To demonstrate our point, we apply our framework to four popular and representative ML algorithms – logistic regression for classification, least squares for regression, K-Means for clustering, and

Op Type	Name	Expression	Output Type
Element-wise Scalar Op	Arithmetic Op ($? = +, -, *, /$)	T ? x or x ? T	Normalized Matrix
	Exponentiation	T^x or x^T	
	Scalar Function f (e.g., log, exp, sin)	f(T)	
Aggregation	Row Summation	rowSums(T)	Column Vector
	Column Summation	colSums(T)	Row Vector
	Summation	sum(T)	Scalar
Multiplication	Left Multiplication	T %*% X	Regular Matrix
	Right Multiplication	X %*% T	
	Crossproduct	crossprod(T)	
	Double Multiplication	T1 %*% T2	
Element-wise Matrix Op	Arithmetic Op ($? = +, -, *, /$)	X ? T or T ? X	

Table 4.1: Operators and functions of linear algebra (using R notation) handled in this project over a normalized matrix T. The parameter X or x is a scalar for Element-wise Scalar Ops, a $(d_S + d_R) \times d_x$ matrix for Left Multiplication, an $n_x \times n_S$ matrix for Right Multiplication, and an $n_s \times (d_S + d_R)$ matrix for Element-wise Matrix Ops. All the operators except Element-wise Matrix Ops are factorizable in general.

Gaussian non-negative matrix factorization (GNMF) for feature extraction – and show how these algorithms are automatically factorized over normalized data.

From a technical perspective, we organize a large subset of linear algebra operators that arise commonly in ML into four groups as shown in Table 4.1. Element-wise scalar operators such as scalar-matrix multiplication are trivial to rewrite, while the rewrites for aggregation operators are reminiscent of query optimization rules for SQL aggregates over joins [Chaudhuri and Shim, 1994; Yan and Larson, 1995]. Matrix multiplication operators, which are crucial for ML, require more complex rewrites and allow for alternative rewrites with different runtime performance. We discuss the trade-offs involved and explain how to pick a rewrite. Multiplication of two normalized matrices requires predicting the size of an intermediate sparse matrix, which is reminiscent of the classical problem of cardinality estimation. We provide a formal characterization of this operation and propose an effective heuristic. Finally, element-wise matrix operators (e.g., matrix addition) do not usually have redundancy, which means rewriting cannot improve the performance. Fortunately, they seldom occur in popular ML algorithms [Mitchell, 1997].

In order to handle cases in which the input normalized matrix is *transposed*, we devise new rewrite rules for the first three groups of operators. Interestingly, in most cases, it turns

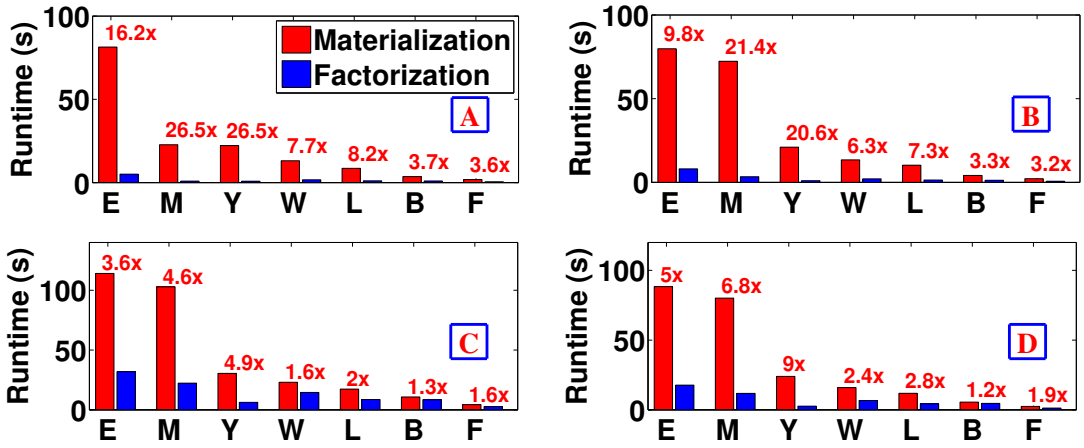


Figure 4.5: Performance on real datasets for (A) Linear Regression, (B) Logistic Regression, (C) K-Means, and (D) GNMF. E, M, Y, W, L, B, and F correspond to the Expedia, Movies, Yelp, Walmart, LastFM, Books, and Flights dataset respectively. The number of iterations/centroids/topics is 20/5/5.

out that rewrite rules can be easily adapted to transposed inputs due to the "push down" properties of the transpose operator with respect to some other linear algebra operators. However, when a normalized matrix is multiplied with a transposed normalized matrix, multiple rewrites are possible and it is non-obvious which rewrite should be picked. This issue is similar to the one faced when multiplying two normalized matrices. Finally, we extend all our rules to multi-table joins rather than just a two-table join. Interestingly, the rewrite rules for multi-table joins lead to an instance of the multiplication of two normalized matrices.

We perform an extensive empirical analysis of our system with both synthetic and real datasets. Across a wide range of data sizes, we find that our factorized operators and our factorized versions of the ML algorithms are almost always significantly faster than their corresponding materialized versions. To predict cases of slow-down, we devise simple but effective heuristic decision rules. Finally, we evaluate the performance of the factorized ML algorithms on seven real-world normalized datasets from Kumar et al. [2016] (Chapter 5). Figure 4.5 presents the results. Overall, the speed-ups ranged from 1.2x to 26.5x on the real datasets. This shows that our framework not only lowers development overhead, but that it can also yield significant runtime speed-ups.

This project is joint work with an MS student, Lingjiao Chen, along with Jeff Naughton and Jignesh Patel. It is under submission to a conference. My contribution was in the conceptualization of this project and advising the other student throughout the algorithmic and experimental details.

5 Hamlet: Avoiding Joins Logically

In this chapter, we dive deeper into our technique of avoiding joins logically. Recall that irrespective of whether joins are performed physically, all features from all base tables are used for ML. Almost always, data scientists apply a feature selection method, either explicitly or implicitly [Guyon et al., 2006], over this entire set of features in conjunction with their chosen ML model. Feature selection helps improve ML accuracy and is widely considered critical for ML-based analytics [Guyon et al., 2006; SAS, a; Zhang et al., 2014; Konda et al., 2013]. While this process certainly "works," it can be both painful and wasteful because the increase in the number of features might make it harder for data scientists to explore the data and also increases the runtime of ML and feature selection methods. In some cases, the joins might also be expensive and introduce data redundancy, causing even more efficiency issues, as shown by our work in Project Orion (Chapter 3).

In this project, we help mitigate the above issues by studying a rather radical idea: *What if we ignore a KFK join, i.e., a base table, entirely?* In other words, is it possible to ignore all "foreign" features (the features from the table referred to by the foreign key) in the first place without significantly reducing ML accuracy? We call this process "avoiding the join." At first glance, this seems preposterous: how can we be confident that ignoring some features is unlikely to reduce accuracy significantly without even running the feature selection method over the data (which requires the join)? The key turns out to be a rather simple observation: the KFK dependencies present in the *schema* enable us to avoid joins. Simply put, in an information theoretic sense [Guyon et al., 2006], a foreign key encodes "all information" about all the foreign features brought in by a KFK join, which allows us to use it as a "representative" for the foreign features. Thus, this observation seems to make things stunningly simple: ignore all KFK joins and use foreign keys as representatives of foreign features!

Alas, the real world is not as simple as described above. Unfortunately, the information theoretic perspective is not sufficiently perspicacious to fully answer our core question of when it is "safe" to avoid a join, i.e., when ML accuracy is unlikely to be affected significantly. The finite nature of training datasets in the real world makes it necessary to analyze our problem using the standard ML notions of bias and variance [Shalev-Shwartz and Ben-David, 2014]. This requires detailed, yet subtle, theoretical analysis (which we perform) of the effects of KFK joins on ML. It turns out that foreign features, which are safe

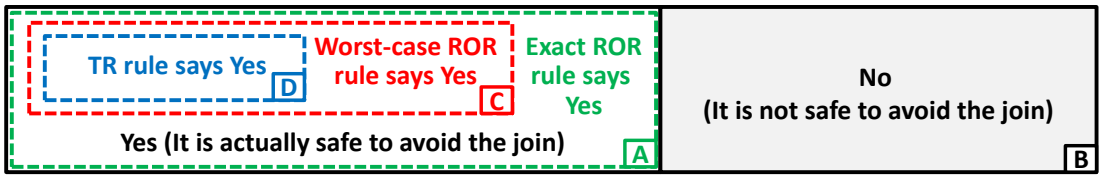


Figure 5.1: Illustrating the relationship between the decision rules to tell which joins are "safe to avoid."

to ignore from an information theoretic perspective, could be indispensable when both bias and variance are considered. This brings us back to square one with our conundrum: *given a KFK join, how to tell if it is safe to avoid or not?*

Answering the above core question could yield at least four benefits:

1. It can help improve the performance of ML tasks without losing much accuracy.
2. In applications in which data scientists explore features by being in the loop [Kumar et al., 2015b; Zhang et al., 2014], having fewer tables and features might make exploration easier.
3. It might help reduce the costs of data acquisition in applications where new tables (e.g., weather data) are purchased and joined with existing data. If we can show that such joins might not really help accuracy, data scientists can reconsider such purchases.
4. In some applications, data scientists have dozens of tables in the input and prefer to join only a few "most helpful" tables (colloquially called *source selection*). Answering our question might help data scientists assess which tables matter less for accuracy.

In this project, we show that it is possible to answer our core question by designing practical heuristics that are motivated by our theoretical understanding. Thus, apart from establishing new connections between joins and ML, our work can help make feature selection over normalized data easier and faster. Indeed, the data management community is increasingly recognizing the need for more of such formal and systems support to make feature selection easier and faster [Anderson et al., 2013; Zhang et al., 2014; Ré et al., 2014; Kumar et al., 2015b]. Ideally, we desire a *decision rule* that can help answer the question for data scientists. Apart from being effective and concurring with our formal analysis, we desire that any such rule be *simple* (making it easy for data scientists to understand and implement), *generic* (not tied too closely to a specific ML model), *flexible* (tunable based on what error is tolerable), and *fast*. These additional desiderata are motivated by practical real-world systems-oriented concerns.

Figure 5.1 illustrates our situation in a rough but intuitive manner. The whole box is the set of KFK joins that gather features. (Section 5.1 makes our assumptions precise.) Box A is the set of joins that are "safe to avoid," i.e., avoiding them is unlikely to blow up the test error (Section 5.2 makes this precise), while box B is the rest. Our goal is to characterize boxes A and B and develop a decision rule to tell if a given join belongs to box A. We first perform a simulation study (using Naive Bayes as an example) to measure how different properties of the base tables affect the test error. We apply our theoretical and simulation results to devise an intuitive definition of box A and design a decision rule for it that exploits a powerful ML notion called the *VC dimension* [Vapnik, 1995]. Applying a standard theoretical result from ML, we define a heuristic quantity called the *Risk Of Representation* (ROR) that intuitively captures the increased risk of the test error being higher than the train error by avoiding the join. Using an appropriate threshold on the ROR yields a decision rule that can tell if a join is "safe to avoid" or not; this is how boxes A and B in Figure 5.1 are defined. Sadly, it is *impossible* in general to compute the ROR a priori, i.e., without performing the very feature selection computations we are trying to avoid. To resolve this quandary, we derive an upper bound on the ROR (we call it *worst-case* ROR) that is computable a priori. It yields a more *conservative* decision rule, i.e., it might wrongly predict that a join is not safe to avoid even though it actually is. Figure 5.1 illustrates this relationship: box C is contained in box A. The intersection of box A with the complement of box C is the set of missed opportunities for the worst-case ROR rule.

The worst-case ROR rule still requires us to inspect the foreign features (without having to do the join, of course). This motivates us to design an even simpler rule that does not even require us to look at the foreign features. We define a quantity we call the *tuple ratio* (TR) that only depends on the number of training examples and the size of the foreign key's domain. The TR is a conservative simplification of the worst-case ROR. Thus, the TR rule might miss even more opportunities for avoiding joins. Figure 5.1 illustrates this relationship: box D is contained in box C.

In the rest of this chapter, we develop the precise theoretical machinery needed to explain our problem, characterize the effects of KFK joins on ML, and explain how we design our decision rules. Furthermore, since both of these rules are conservative, it is important to know how they perform on real data. Thus, we perform an empirical analysis with seven real-world datasets from diverse application domains: retail, hospitality, transportation, e-commerce, etc. We combine a few popular feature selection methods and popular ML classifiers. We find that there are indeed many cases on real datasets where joins are safe to avoid, and both of our rules work surprisingly well: out of 14 joins in total across all 7 datasets, both of our rules correctly classified 7 joins as safe to avoid and 3 joins as not safe to avoid but deemed 4 joins as not safe to avoid even though avoiding them did

not blow up the test errors (note that our decision rules are conservative). Overall, our decision rules improved the performance of the feature selection methods significantly in many cases, including by over 10x in some cases.
In summary, this project makes the following contributions:

- To the best of our knowledge, this is the first project to study the problem of formally characterizing the effects of KFK joins on ML classifiers and feature selection to help predict when joins can be avoided safely.
- We perform a simulation study using Naive Bayes as an example in order to measure how different properties of the normalized data affect ML error.
- We apply our theoretical and simulation results to design simple decision rules that can predict a priori if it is perhaps safe to avoid a given join.
- We perform an extensive empirical analysis using real-world datasets to validate that there are cases where avoiding joins does not increase ML error significantly and that our rules can accurately predict such cases.

Outline Section 5.1 presents an in-depth theoretical analysis of the effects of joins on ML and feature selection. Readers more interested in the practical implications can skip to Section 5.2, which presents our simulation study and also explains how we design our decision rules. Section 5.3 presents our empirical validation with real data.

5.1 Effects of KFK Joins on ML

We start with a description of some extra assumptions for this work compared to Project Orion. We then provide an information theoretic analysis of the effects of KFK joins on ML and then dig deeper to establish the formal connections between KFKDs and the bias-variance trade-off in ML. For ease of exposition, we assume there is only one attribute table $\mathbf{R}$. Readers more interested in the practical aspects can skip to the summary at the end of this section or to Section 5.3.

Assumptions and Setup We focus on the case in which all features (including Y) are *nominal*, i.e., each feature has a finite discrete domain.[1] Thus, we focus on *classification*. We assume that the foreign keys are not keys of $\mathbf{S}$ (e.g., `EmployerID` is clearly not a key of `Customers`). We also assume that the domains of all features in $\mathbf{X}_S$, $\mathbf{X}_R$, and all FK_i are "closed with respect to the prediction task" and the domain of FK_i is the same as the set

[1] Numeric features are assumed to have been discretized to a finite set of categories, say, using binning [Mitchell, 1997].

of RID_i values in $\mathbf{R}_i$ (and there are no missing/NULL values)[2]. We explain the closed domain assumption. In ML, recommendation systems assume that the foreign keys of the ratings table, e.g., `MovieID` and `UserID` have closed domains, say, to enable matrix factorization models [Koren et al., 2009]. A movie might have several past ratings. So, `MovieID` can help predict future ratings. Note that closed domain does *not* mean new `MovieID` values can never occur! It means that data scientists build models using only the movies seen so far but revise their feature domains and update ML models periodically (say, monthly) to absorb movies added recently. In between these revisions, the domain of `MovieID` is considered closed. `EmployerID` in our example plays exactly the same role: many customers (past and future) might have the same employer. Thus, it is reasonable to use `EmployerID` as a feature. Handling new movies (or employers) is a well-known problem called *cold-start* [Schein et al., 2002]. It is closely related to the *referential integrity constraint* for foreign keys in databases [Ramakrishnan and Gehrke, 2003]. In practice, a common way to handle it is to have a special "Others" record in **`Employers`** as a placeholder for new employers seen in between revisions. Cold-start is orthogonal to our problem; we leave it to future work.

Overall, each feature in $\mathbf{X}_S$, $\mathbf{X}_{R_i}$, and FK_i is a discrete random variable with a known finite domain. We also assume that the foreign keys are not skewed (we relax this in a discussion in the appendix). We discuss the effects of KFK joins on ML classifiers and feature selection in general, but later we use Naive Bayes as an example.[3] Naive Bayes is a popular classifier with diverse applications ranging from spam detection to medical diagnosis [Mitchell, 1997; Pearl, 1988]. It is also easy to understand and use; it does not require expensive iterative optimization or "black magic" for tuning hyper-parameters.

We emphasize that our goal is *not* to design new ML models or new feature selection methods, nor is to study which feature selection method or ML model yields the highest accuracy. Rather, our goal is to understand the theoretical and practical implications of ubiquitous database dependencies, viz., KFK dependencies (KFKDs) and functional dependencies (FDs) on ML. In this work, we use the phrase "the join is safe to avoid" to mean that $\mathbf{X}_R$ can be dropped before feature selection without significantly affecting the test error of the subset obtained after feature selection. We make this notion more precise later (Section 5.2).

[2]To handle RID_i values absent from FK_i in a given instance of $\mathbf{S}$, we adopt the standard practice of *smoothing* [Manning et al., 2008].

[3]We present some empirical results and a discussion of some other popular ML models in Section 5.3 and the appendix.

The Information Theoretic Perspective

A standard theoretical approach to ascertain which features are "useful" is to use the information theoretic notions of feature redundancy and relevancy [Guyon et al., 2006; Koller and Sahami, 1995]. Thus, we now perform such an analysis to help explain why it might be safe to avoid the join with $\mathbf{R}$, i.e., ignore $\mathbf{X}_R$.

Feature Redundancy

We start with some intuition. The foreign key FK is also a feature used to predict Y. In our example, it is reasonable to use `EmployerID` to help predict `Churn`. The equi-join condition $\mathbf{S}.\mathsf{FK} = \mathbf{R}.\mathsf{RID}$ that creates $\mathbf{T}$ causes FK to *functionally determine* all of $\mathbf{X}_R$ in $\mathbf{T}$. It is as if the FD $\mathsf{RID} \rightarrow \mathbf{X}_R$ in $\mathbf{R}$ becomes the FD $\mathsf{FK} \rightarrow \mathbf{X}_R$ in $\mathbf{T}$.[4] Thus, given FK, $\mathbf{X}_R$ is fixed, i.e., $\mathbf{X}_R$ does *not* provide any more "information" over FK. The notion of feature *redundancy* helps capture such behavior formally [Yu and Liu, 2004; Koller and Sahami, 1995]. Its rigorous definition is as follows.

Definition 5.1. Weak relevance. *A feature* $\mathsf{F} \in \boldsymbol{X}$ *is* weakly relevant *iff* $\mathsf{P}(\mathsf{Y}|\boldsymbol{X}) = \mathsf{P}(\mathsf{Y}|\boldsymbol{X} - \{\mathsf{F}\})$ *and* $\exists \boldsymbol{Z} \subseteq \boldsymbol{X} - \{\mathsf{F}\}$ s.t. $\mathsf{P}(\mathsf{Y}|\boldsymbol{Z}, \mathsf{F}) \neq \mathsf{P}(\mathsf{Y}|\boldsymbol{Z})$.

Definition 5.2. Markov blanket. *Given a feature* $\mathsf{F} \in \boldsymbol{X}$*, let* $\boldsymbol{M}_\mathsf{F} \subseteq \boldsymbol{X} - \{\mathsf{F}\}$*;* $\boldsymbol{M}_\mathsf{F}$ *is a* Markov Blanket *for* F *iff* $\mathsf{P}(\mathsf{Y}, \boldsymbol{X} - \{\mathsf{F}\} - \boldsymbol{M}_\mathsf{F}|\boldsymbol{M}_\mathsf{F}, \mathsf{F}) = \mathsf{P}(\mathsf{Y}, \boldsymbol{X} - \{\mathsf{F}\} - \boldsymbol{M}_\mathsf{F}|\boldsymbol{M}_\mathsf{F})$.

Definition 5.3. Redundant feature. *A feature* $\mathsf{F} \in \boldsymbol{X}$ *is* redundant *iff it is* weakly relevant *and it has a* Markov blanket *in* $\boldsymbol{X}$*.*

Applying the above notion of feature redundancy yields our first, albeit simple, result (let $\mathbf{X} \equiv \mathbf{X}_S \cup \{\mathsf{FK}\} \cup \mathbf{X}_R$).

Proposition 5.1.1. *In* $\boldsymbol{T}$*, all* $\mathsf{F} \in \boldsymbol{X}_R$ *are redundant.*

The proof is in the appendix. This result extends trivially to multiple attribute tables. In fact, it extends to a more general set of FDs, as shown by the following corollary.

Definition 5.4. *A set of FDs* $\mathcal{Q}$ *over* $\boldsymbol{X}$ *is* acyclic *iff the digraph on* $\boldsymbol{X}$ *created as follows is acyclic: include an edge from feature* X_i *to* X_j *if there is an FD in* $\mathcal{Q}$ *in which* X_i *is in the determinant set and* X_j *is in the dependent set.*

Corollary 5.5. *Given a table* $\boldsymbol{T}(\underline{\mathsf{ID}}, \mathsf{Y}, \boldsymbol{X})$ *with a canonical acyclic set of FDs* $\mathcal{Q}$ *on the features* $\boldsymbol{X}$*, a feature that appears in the dependent set of an FD in* $\mathcal{Q}$ *is redundant.*

[4] KFKDs differ from FDs [Abiteboul et al., 1995], but we can treat the dependencies in $\mathbf{T}$ as FDs since we had assumed that there are no `NULL` values and that all feature domains are *closed*.

In the ML literature, the redundancy of a feature is often considered an indication that the feature should be dropped. In fact, many feature selection methods explicitly search for such redundancy in order to remove the redundant features [Koller and Sahami, 1995; Yu and Liu, 2004; Guyon et al., 2006]. However, they try to detect the presence of feature redundancy approximately based on the dataset *instance*. Our scenario is stronger because Proposition 5.1.1 *guarantees* the existence of redundant features. This motivates us to consider the seemingly "radical" step of avoiding these redundant features, i.e., avoiding the join with $\mathbf{R}$.

Feature Relevancy

A redundant feature might sometimes be more "useful" in predicting Y; this is captured using the formal notion of feature *relevancy*. This leads to the classical redundancy-relevancy trade-off in ML [Guyon et al., 2006]. We provide some intuition. Suppose, in our example, that customers of rich corporations never churn and they are the only ones who do not churn; then `Revenue` is perhaps the most relevant feature, even though it is redundant. Thus, we would like to know if it is possible for some $F \in \mathbf{X}_R$ to be *more* relevant than FK. Feature relevancy is often formalized using scores such as the mutual information $I(F;Y)$ or the information gain ratio $IGR(F;Y)$; formal definitions of these quantities are provided in Chapter 2 (Section 2.3). A feature with a higher score is considered to be more relevant [Guyon et al., 2006; Yu and Liu, 2004]. However, when we apply these notions to our setting, we realize that our information theoretic analysis hits a wall.

Theorem 5.6. $\forall F \in \boldsymbol{X}_R, I(F;Y) \leqslant I(FK;Y)$

Proposition 5.1.2. *It is possible for a feature* $F \in \boldsymbol{X}_R$ *to have higher* $IGR(F;Y)$ *than* $IGR(FK;Y)$.

The proofs are in the appendix. Basically, we get near-opposite conclusions depending on the score! Using mutual information, features in $\mathbf{X}_R$ are no more relevant than FK. Coupled with the earlier fact that $\mathbf{X}_R$ is redundant, this suggests strongly that the join is not too useful and that we might as well stick with using FK as a "representative" of $\mathbf{X}_R$. However, using information gain ratio, a feature in $\mathbf{X}_R$ could be *more* relevant than FK. This suggests that we should bring in $\mathbf{X}_R$ by performing the join and let the feature selection method ascertain which features to use. We explain this strange behavior intuitively.

The domain of FK is likely to be much larger than any feature in $\mathbf{X}_R$. For example, there are fewer than fifty states in the USA, but there are millions of employers. Thus, `EmployerID` might have a much larger domain than `State` in **`Employers`**. Mutual information tends to prefer features with larger domains, but information gain ratio resolves this issue by penalizing features with larger domains [Guyon et al., 2006; Mitchell, 1997]. This brings

us back to square one with our original conundrum! It seems the information theoretic analysis is insufficient to help us precisely answer our core question of when it is safe to avoid a join. Thus, we now dive deeper into our problem by analyzing the effects of KFK joins on ML and feature selection from the perspective of the bias-variance trade-off, which lies at the heart of ML.

The Join Strikes Back: KFK Joins and the Bias-Variance Trade-off

We now expose a "danger" in avoiding the join (using FK as a representative for $\mathbf{X}_R$). Our intuition is simple: information theoretic arguments are generally applicable to asymptotic cases but in the real world, training datasets are *finite*. Thus, we need to look to statistical learning theory to understand precisely how ML error is affected. We start with an intuitive explanation of some standard concepts.

The expected *test error* (i.e., error on an unseen labeled example) can be decomposed into three components: *bias* (a.k.a *approximation error*), *variance* (a.k.a. *estimation error*), and *noise* [Hastie et al., 2003; Shalev-Shwartz and Ben-David, 2014]. The noise is an inevitable component that is independent of the ML model. The bias captures the lowest possible error by any model instance in the class of ML models considered. For example, using Naive Bayes introduces a bias compared to learning the expensive joint distribution, since it assumes conditional independence [Pearl, 1988]. The variance captures the error introduced by the fact that the training dataset is only a random finite sample from the underlying data distribution, i.e., it formalizes the "instability" of the model with respect to the training data. For example, if we train Naive Bayes on two different training samples, their test errors are likely to differ. And if we provide fewer training examples to the ML model, its test error is likely to increase due to higher variance. In colloquial terms, a scenario with high variance is called *overfitting* [Hastie et al., 2003].

The crux of the argument is as follows: *using* FK *as a representative is likely to yield a model with higher variance than a model obtained by including* $\boldsymbol{X}_R$ *for consideration.* Thus, the chance of getting a higher test error might increase if we avoid the join. Perhaps surprisingly, this holds true irrespective of the number of features in $\mathbf{X}_R$! Before explaining why, we observe that there is no contradiction with Section 5.1; the information theoretic analysis deals primarily with bias, not variance. We explain more below.

Relationship between Hypothesis Spaces

To help formalize our argument, we first explain the relationship between the classes of models built using FK and $\mathbf{X}_R$. This is important to understand because a model with a larger hypothesis space might have higher variance.

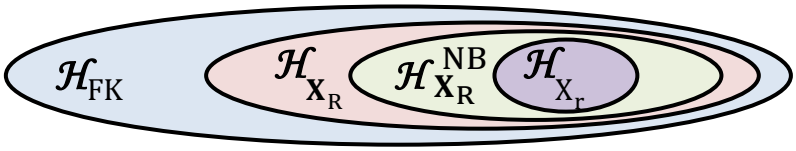


Figure 5.2: Relationship between hypothesis spaces.

As before, let $\mathbf{X} \equiv \mathbf{X}_S \cup \{FK\} \cup \mathbf{X}_R$. For simplicity of exposition, let $\mathcal{D}_Y = \{0, 1\}$. Also, since our primary goal is to understand the effects of the FD $FK \rightarrow \mathbf{X}_R$, we set $\mathbf{X}_S = \phi$ (empty) for ease of exposition. A learned ML model instance is just a prediction function $f : \mathcal{D}_{\mathbf{X}} \rightarrow \{0, 1\}$. The universe of all possible prediction functions based on $\mathbf{X}$ is denoted $\mathcal{H}_{\mathbf{X}} = \{f | f : \mathcal{D}_{\mathbf{X}} \rightarrow \{0, 1\}\}$. A given class of ML models, e.g., Naive Bayes, can only learn a subset of $\mathcal{H}_{\mathbf{X}}$ owing to its bias. This subset of functions that it can learn is called the *hypothesis space* of the ML model. Let $\mathcal{H}^{NB}_{\mathbf{X}}$ denote the hypothesis space of Naive Bayes models on $\mathbf{X}$. Given $\mathbf{Z} \subseteq \mathbf{X}$, we define the *restriction* of $\mathcal{H}_{\mathbf{X}}$ to $\mathbf{Z}$ as follows: $\mathcal{H}_{\mathbf{Z}} = \{f | f \in \mathcal{H}_{\mathbf{X}} \wedge \forall \mathbf{u}, \mathbf{v} \in \mathcal{D}_{\mathbf{X}}, \mathbf{u}_{|\mathbf{Z}} = \mathbf{v}_{|\mathbf{Z}} \implies f(\mathbf{u}) = f(\mathbf{v})\}$. Here, $\mathbf{u}_{|\mathbf{Z}}$ denotes the projection of $\mathbf{u}$ to only the features in $\mathbf{Z}$. We now establish the relationship between the various hypothesis spaces. Figure 5.2 depicts it pictorially.

Proposition 5.1.3. $\mathcal{H}_{\mathbf{X}} = \mathcal{H}_{FK} \supseteq \mathcal{H}_{\mathbf{X}_R}$

The proof is in the appendix. Note that the first part ($\mathcal{H}_{\mathbf{X}} = \mathcal{H}_{FK}$) is essentially the learning theory equivalent of Proposition 5.1.1. The second part might seem counter-intuitive because even if there are, say, a million features in $\mathbf{X}_R$ but only a hundred FK values, using FK instead of $\mathbf{X}_R$ is still more "powerful" than using $\mathbf{X}_R$ and dropping FK. But the intuition is simple: since $\mathbf{R}$ is fixed in our setting, we can only ever observe at most $|\mathcal{D}_{FK}|$ distinct values of $\mathbf{X}_R$, even if $\Pi_{F \in \mathbf{X}_R} |\mathcal{D}_F| \gg |\mathcal{D}_{FK}|$. Hence, using $\mathbf{X}_R$ instead of FK might increase the bias. For example, suppose that customers employed by Profit University and Charity Inc. churn and they are the only ones who churn. Then it is *impossible* in general to learn this concept correctly if `EmployerID` is excluded. Note that $\mathcal{H}^{NB}_{\mathbf{X}_R} \subseteq \mathcal{H}_{\mathbf{X}_R}$. Finally, $\forall X_r \in \mathbf{X}_R, \mathcal{H}^{NB}_{\mathbf{X}_R} \supseteq \mathcal{H}_{X_r} = \mathcal{H}^{NB}_{X_r}$ (and $\mathcal{H}^{NB}_{FK} = \mathcal{H}_{FK}$), since Naive Bayes has no bias if there is only one feature.

The relationship between the size of the hypothesis space and the variance is formalized in ML using the powerful notion of the *VC dimension* [Vapnik, 1995; Shalev-Shwartz and Ben-David, 2014]. We use this notion to complete the formalization of our argument.

VC Dimension

We give an intuitive explanation and an example here and refer the interested reader to Shalev-Shwartz and Ben-David [2014] for more details. Intuitively, the VC dimension captures the ability of a classifier to assign the true class labels to a set of labeled data points of a given cardinality – a capability known as "shattering." For example, consider a linear classifier in 2-D. It is easy to see that it can shatter any set of 3 points (distinct and non-collinear). But it cannot shatter a set of 4 points due to the XOR problem [Domingos, 2012]. Thus, the VC dimension of a 2-D linear classifier is 3. For finite hypothesis spaces (as in our case), the VC dimension is a direct indicator of the size of the hypothesis space [Shalev-Shwartz and Ben-David, 2014]. A standard result from ML bounds the difference between the test and train errors (this difference is solely due to variance) as a function of the VC dimension (ν) and the number of training examples (n):

Theorem 5.7. (From Shalev-Shwartz and Ben-David [2014], p. 51) *For every $\delta \in (0, 1)$, with probability at least $1 - \delta$ over the choice of the training dataset, and for $n > \nu$, we have:*

$$|\mathtt{Test\ error} - \mathtt{Train\ error}| \leqslant \frac{4+\sqrt{\nu \mathtt{log}(2en/\nu)}}{\delta\sqrt{2n}}$$

Thus, a higher VC dimension means a looser bound on the variance and, possibly, higher variance. The VC dimension usually increases with the number of features. For example, it is linear in the number of features for "linear" classifiers such as logistic regression and also Naive Bayes [Vapnik, 1995]. But note that we had assumed that all features are nominal in our setting. Thus, we recode the features to numeric space using the standard binary vector representation, i.e., a feature F is converted to a $0/1$ vector with $|\mathcal{D}_F| - 1$ dimensions (the last category is represented as a zero vector). With this recoding, the VC dimension of Naive Bayes (or logistic regression) on a set $\mathbf{X}$ of nominal features is $1 + \sum_{F \in \mathbf{X}}(|\mathcal{D}_F| - 1)$. If we use FK alone, the maximum VC dimension for any classifier is $|\mathcal{D}_{FK}|$, which is matched by almost all popular classifiers such as Naive Bayes. However, as per the argument for Figure 5.2, the VC dimension of any classifier on $\mathbf{X}_R$ is at most the number of distinct values of $\mathbf{X}_R$ in the given table $\mathbf{R}$, say, r. Since RID is the primary key of $\mathbf{R}$, we have $|\mathcal{D}_{FK}| \geqslant r$. Thus, the VC dimension is likely to be higher if FK is used as a representative for $\mathbf{X}_R$.

In the Context of Feature Selection

The above variance-based argument gets stronger when we consider the fact that we might *not* retain all of $\mathbf{X}_R$ *after* feature selection. Consider an extreme scenario: suppose the "true" distribution can be succinctly described using a lone feature $X_r \in \mathbf{X}_R$. In our churn

example, this represents a case where all customers with employers based in Wisconsin churn and they are the only ones who churn (X_r is `State`). Suppose an "oracle" told us to only use X_r. Clearly, $\mathcal{H}_{X_r}$ is likely to be much smaller than $\mathcal{H}_{FK}$, as illustrated in Figure 5.2. Thus, the variance for a model based on FK is likely to be higher than a model based on X_r, as per Theorem 5.7. For the opposite extreme, i.e., the true concept needs all of $\mathbf{X}_R$, the gap with $\mathcal{H}_{FK}$ might decrease, but it might still be large.

Alas, in the real world, we do not have oracles to tell us which features are part of the true distribution; it could be none, some, or all features in $\mathbf{X}_R$. What we do have instead of an oracle is a feature selection method, although it does the job approximately using a finite labeled sample. For example, in the above extreme scenario, if we input $\{FK, X_r\}$ to a feature selection method, it is likely to output $\{X_r\}$ precisely because a model based on $\{X_r\}$ is likely to have lower variance than one based on $\{FK\}$ or $\{FK, X_r\}$. By avoiding the join, we shut the door on such possibilities and force the model to work only with FK. Thus, overall, even though FK can act as a representative for $\mathbf{X}_R$, it is probably safer to give the entire set $\mathbf{X}$ to the feature selection method and let it figure out the subset to use. Finally, note that we had assumed $\mathbf{X}_S$ is empty for the above discussion because it is orthogonal to our core issue. If $\mathbf{X}_S$ is not empty, all the hypothesis spaces shown in Figure 5.2 will blow up, but their relative relationships, and hence the above arguments about the variance, will remain unaffected.

Summary Our analysis reveals a dichotomy in the accuracy effects of avoiding a KFK join for ML and feature selection: avoiding the join and using FK as a representative of $\mathbf{X}_R$ might not increase the bias, but the variance (compared after feature selection) might increase significantly. All of the above arguments and results extend trivially to the case of multiple attribute tables.

5.2 Predicting a priori if it is Safe to Avoid a KFK Join

Given our understanding of the dichotomy in the effects of joins, we now focus on answering our core question: how to predict a priori if a join with $\mathbf{R}$ is safe to avoid. We start with a simulation study using "controlled" datasets to validate our theoretical analysis and measure precisely how the error varies as we vary different properties of the normalized data. We then explain our decision rules and how we use our simulation measurements to tune the rules. All the plots in Section 5.2 are based on our synthetic datasets.

Simulation Study

We perform a Monte Carlo-style study. For the sake of tractability and depth of understanding, we use Naive Bayes as the classifier in this section. But note that our methodology is generic enough to be applicable to any classifier because we measure the accuracy only using standard notions of error, bias, and variance for a generic ML classifier.

Data Synthesis We sample the labeled examples in an independently and identically distributed manner from a controlled true distribution $P(Y, \mathbf{X})$. Different scenarios are possible based on what features in $\mathbf{X}$ are used: it could be any or all features in $\mathbf{X}_S$, FK, and/or $\mathbf{X}_R$. Our primary goal is to understand the effects of the FD $FK \rightarrow \mathbf{X}_R$ and explain the danger in avoiding the join. Thus, we focus on a key scenario that intuitively represents the worst-case scenario for avoiding the join: the true distribution is succinctly captured using a lone feature $X_r \in \mathbf{X}_R$. This corresponds to the example in Section 5.1 in which all customers with employers based in Wisconsin churn and they are the only ones who churn. In line with Proposition 5.1.3, we expect FK to play an indirect role in predicting Y in both scenarios. All other features are random noise. We also studied two other representative scenarios: one in which all of $\mathbf{X}_S$ and $\mathbf{X}_R$ are part of the true distribution, and one in which only $\mathbf{X}_S$ and FK are. Since these two did not yield any major additional insights, we present them in the appendix.

Simulation Setup There is one attribute table $\mathbf{R}$ ($k = 1$). All of $\mathbf{X}_S$, $\mathbf{X}_R$, and Y are boolean (i.e., domain size 2). The following parameters are varied one at a time: number of features in $\mathbf{X}_S$ (d_S), number of features in $\mathbf{X}_R$ (d_R), $|\mathcal{D}_{FK}|$ ($= n_R$), and total number of training examples (n_S). We also sample $\frac{n_S}{4}$ examples for the test set. We generate 100 different training datasets and measure the test error and the variance based on the different models obtained from these 100 runs. This whole process was repeated 100 times with different seeds for the pseudo-random number generator [Rish et al., 2001]. Thus, we have $10,000$ runs in total for one combination of the parameters studied. While the test error and variance were defined intuitively in Section 5.1, we now define them formally (based on Domingos [2000]).

Definitions We start with the formal definition of error (based on Domingos and Pazzani [1997]).

Definition 5.8. Zero-one loss. *Let* t *be the true class label of a given example with* $\boldsymbol{X} = \boldsymbol{x}$*, while* $c_M(\boldsymbol{x})$ *be the class predicted by a classifier* M *on* $\boldsymbol{x}$*. The* zero-one loss *(of* M *on* $\boldsymbol{x}$*), denoted* $L(t, c_M(\boldsymbol{x}))$*, is defined as* $L(t, c_M(\boldsymbol{x})) = \mathbf{1}_{t=c_M(\boldsymbol{x})}$*, where* $\mathbf{1}$ *is the indicator function.*

Definition 5.9. *The* local error *(in short, the* error*) of* M *on* $\boldsymbol{x}$*, denoted* $E_M(\boldsymbol{x})$*, is defined as the expected value of the zero-one loss, where the expectation is over the values of* Y*, given* $\boldsymbol{X} = \boldsymbol{x}$*, i.e.,* $E_M(\boldsymbol{x}) = \sum_{y \in \mathcal{D}_Y} P(Y = y|\boldsymbol{X} = \boldsymbol{x}) L_M(y, c_M(\boldsymbol{x}))$.

Since M depends on the training data, we compute the expectation of the error over different training datasets, say, by averaging over a given finite collection of training datasets $\mathcal{S}$ (typically, all of the same size). Note that $|\mathcal{S}| = 100$ for our Monte Carlo runs. The *expected error* of a classifier (across true distributions and $\mathcal{S}$) on $\mathbf{x}$ is decomposed as follows:

$$E[L(t, c_M(\mathbf{x}))] = B(\mathbf{x}) + (1 - 2B(\mathbf{x}))V(\mathbf{x}) + cN(\mathbf{x}) \tag{5.1}$$

Here, $B(\mathbf{x})$ is the bias, $V(\mathbf{x})$ is the variance, and $N(\mathbf{x})$ is the noise. The quantity $(1 - 2B(\mathbf{x}))V(\mathbf{x})$ is called the *net variance*, which is needed to capture the opposing effects of the variance on biased and unbiased predictions [Domingos, 2000]. The *main prediction* on $\mathbf{x}$, given $\mathcal{S}$, is defined as the mode among the multi-set of predictions that result from learning M over each training dataset in $\mathcal{S}$. The main prediction is denoted y_m, while a single prediction based on some dataset in $\mathcal{S}$ is denoted y. The bias is defined as the zero-one loss of the main prediction, i.e., $B(\mathbf{x}) = L(t, y_m)$. The variance is defined as the average loss with respect to the main prediction, i.e., $V(\mathbf{x}) = E_D[L(y_m, y)]$. The *average bias* (resp. *average net variance* and *average test error*) is the average of the bias (resp. net variance and test error) over the entire set of test examples. Our goal is to understand how the average test error and the average net variance are affected by avoiding a join. For the sake of readability, we only discuss the key results here and present the others in the appendix.

Results We compare three classes of models: *UseAll*, which uses all of $\mathbf{X}_S$, FK, and $\mathbf{X}_R$, *NoJoin*, which omits $\mathbf{X}_R$, and *NoFK*, which omits FK. Figure 5.3 plots the average test error and average net variance against n_S as well as $|\mathcal{D}_{FK}|$.

At a high level, Figure 5.3 validates our theoretical results on the dichotomy in the effects of avoiding the join. Both *UseAll* and *NoFK* use X_r, which enables them to achieve the lowest errors. When n_S is large, *NoJoin* matches their errors even though it does not use X_r. This confirms our arguments in Section 5.1 about FK acting as a representative of X_r. However, when n_S drops, the error of *NoJoin* increases, and as Figure 5.3(A2) shows, this is due to the increase in the net variance. Figure 5.3(B) drills into why that happens: for a fixed n_S, a higher $|\mathcal{D}_{FK}|$ yields a higher error for *NoJoin*, again because of higher net variance. This confirms our arguments in Section 5.1 about the danger of using FK as a representative due to the increase in the variance. For the sake of readability, we discuss other parameters and the other two simulation scenarios in the appendix.

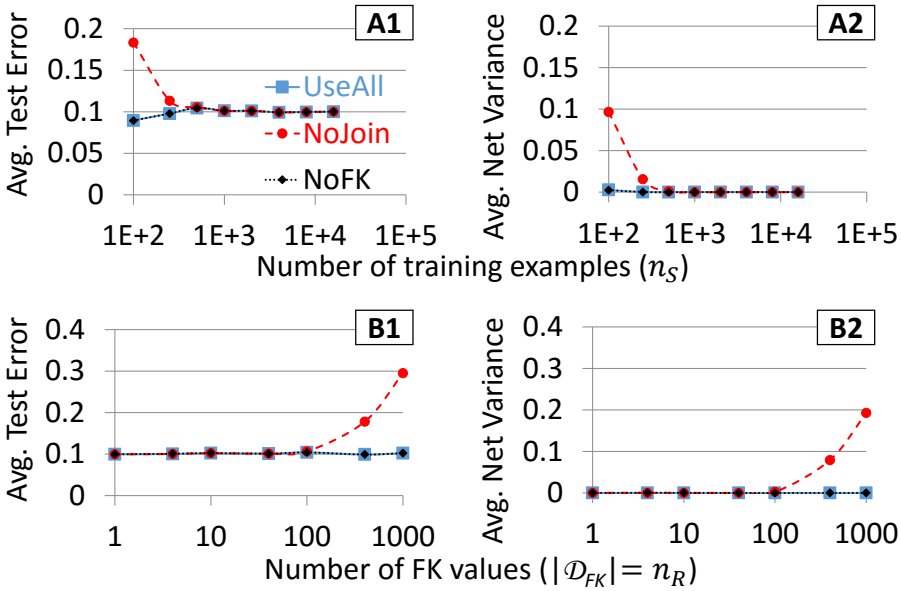


Figure 5.3: Simulation results for the scenario in which only a single $X_r \in \mathbf{X}_R$ is part of the true distribution, which has $P(Y = 0|X_r = 0) = P(Y = 1|X_r = 1) = p$. For these results, we set $p = 0.1$ (varying this probability did not change the overall trends). (A) Vary n_S, while fixing $(d_S, d_R, |\mathcal{D}_{FK}|) = (2, 4, 40)$. (B) Vary $|\mathcal{D}_{FK}|$ $(= n_R)$, while fixing $(n_S, d_S, d_R) = (1000, 4, 4)$.

Towards a Decision Rule

We now precisely define what we mean by "a join is safe to avoid" and devise intuitive decision rules to predict such cases. While our definition and decision rules are heuristic, they are based on our theoretical and simulation-based insights and they satisfy all the desiderata listed in the introduction of this chapter. Our key guiding principle is *conservatism*: it is fine to not avoid a join that could have been avoided in hindsight (a missed opportunity), but we do not want to avoid a join that should not be avoided (i.e., the error blows up if it is avoided). This is reasonable since the feature selection method is there to figure out if features in $\mathbf{X}_R$ are not helpful, albeit with poorer performance. Designing a rule to avoid joins safely is challenging mainly because it needs to balance this subtle performance-accuracy trade-off correctly.

The Risk of Representation

We start be defining a heuristic quantity based on the *increase* in the error bound given by Theorem 5.7. Intuitively, it quantifies the "extra risk" caused by avoiding the join. We compare the bounds for a hypothetical "best" model that uses some subset of $\mathbf{X}_R$ instead

Symbol	Meaning
ϵ	Tolerance for safety with the ROR
δ	Failure probability for the VC-dim bound and the ROR
v_{Yes}	VC-dim of a (hypothetical) "best" classifier that avoids the join
v_{No}	VC-dim of a "best" classifier that does not avoid the join
q_S	$\sum_{F \in \mathbf{X}_S}(\lvert D_F \rvert - 1)$
q_R	Number of unique values of $\mathbf{X}_R$ in $\mathbf{R}$
q_{No}	$v_{No} - q_S$
q_R^*	$\min_{F \in \mathbf{X}_R} \lvert D_F \rvert$

Table 5.1: Notation used in this chapter.

of FK (join performed) against one that uses FK instead (join avoided). A subset of $\mathbf{X}_S$ might be used by both. We call this quantity the *Risk of Representation* (ROR):

$$\text{ROR} = \frac{\sqrt{v_{Yes} \log(\frac{2en}{v_{Yes}})} - \sqrt{v_{No} \log(\frac{2en}{v_{No}})}}{\delta\sqrt{2n}} + \Delta bias$$

In the above, v_{Yes} is the VC dimension of a classifier that uses FK as a representative and avoids the join, while v_{No} is for one that does not avoid the join. Table 5.1 lists all the extra notation used in the rest of this section. We first define them and then explain the intuition behind their definition. For the sake of simplicity, we restrict ourselves to models such as Naive Bayes and logistic regression that have VC dimension linear in the number of features, but discuss some other classifiers later.[5] We are given $\mathbf{X} = \mathbf{X}_S \cup \{FK\} \cup \mathbf{X}_R$. Suppose an "oracle" told us that $\mathbf{U}_S \subseteq \mathbf{X}_S$ and $\mathbf{U}_R \subseteq \mathbf{X}_R$ are the only features in the true distribution. We only consider the case in which $\mathbf{U}_R$ is non-empty.[6] Thus, ideally, $v_{Yes} = \sum_{F \in \mathbf{U}_S}(|\mathcal{D}_F| - 1) + |\mathcal{D}_{FK}|$. Denote $\sum_{F \in \mathbf{U}_S}(|\mathcal{D}_F| - 1)$ by q_S; this is the sum of the number of unique values of all features in $\mathbf{U}_S$. Let q_R denote the number of unique values of $\mathbf{U}_R$ (taken jointly; not as individual features) in $\mathbf{R}$. In general, v_{No} does not have a closed form expression, since it depends on $\mathbf{R}$, but we have $q_S < v_{No} \leqslant q_S + q_R$. Thus, $v_{No} \leqslant v_{Yes}$. Once again, since we do not have oracles in the real world, we will not know $\mathbf{U}_S$ and $\mathbf{U}_R$ a priori. Thus, it is *impossible* to compute the ROR exactly in general.[7] Furthermore, Theorem 5.7 only deals with the variance, not the bias. Thus, we denote the difference in bias using $\Delta bias$ in the ROR. Given this definition of the ROR, we now precisely state what we mean by the join with $\mathbf{R}$ is "safe to avoid."

[5]The upper bound derivation is similar for classifiers with more complex VC dimensions, e.g., the joint distribution. We leave a deeper formal analysis to future work.

[6]If $\mathbf{U}_R$ is empty, $\mathbf{R}$ is trivially useless.

[7]A feature selection method can ascertain $\mathbf{U}_S$ and $\mathbf{U}_R$ roughly but our goal is to avoid this computation.

Definition 5.10. *Given a failure probability δ and a bound $\epsilon > 0$, we say the join with $\boldsymbol{R}$ is* (δ, ϵ)-safe to avoid *iff the ROR with the given δ is no larger than ϵ.*

The ROR Rule

While the ROR intuitively captures the risk of avoiding the join and provides us a threshold-based decision rule, we immediately hit another wall: it is *impossible* in general to compute $\Delta bias$ a priori without knowing $\mathbf{U}_S$ and $\mathbf{U}_R$. Thus, prima facie, using the ROR directly for a decision rule seems to be a hopeless idea to pursue! We resolve this quandary using a simple observation: we do not really need the exact ROR but only a "good enough" indicator of the risk of using FK as a representative. Thus, drawing upon our guiding principle of conservatism, we upper bound the ROR and create a more conservative decision rule. We explain the derivation step by step. Assume $n > v_{Yes}$. First, Proposition 5.1.3 showed that dropping $\mathbf{X}_R$ a priori does not increase the bias but dropping FK might, which means $\Delta bias \leqslant 0$. Hence, we ignore $\Delta bias$ entirely:

$$ROR \leqslant \frac{\sqrt{v_{Yes} log(\frac{2en}{v_{Yes}})} - \sqrt{v_{No} log(\frac{2en}{v_{No}})}}{\delta\sqrt{2n}}$$

Second, we substitute the values of some variables in the above inequality. Denote $q_{No} = v_{No} - q_S$ (the slack in the inequality $q_S < v_{No}$). The inequality now becomes:

$$ROR \leqslant \frac{1}{\delta\sqrt{2n}}[\sqrt{(q_S + |\mathcal{D}_{FK}|) log(2en/(q_S + |\mathcal{D}_{FK}|))} - \sqrt{(q_S + q_{No}) log(2en/(q_S + q_{No}))}]$$

Third, we observe that since $|\mathcal{D}_{FK}| \geqslant q_R \geqslant q_{No}$, the RHS above is a non-increasing function of q_S that is maximum when $q_S = 0$. This lets us eliminate $\mathbf{U}_S$ from the picture:

$$ROR \leqslant \frac{1}{\delta\sqrt{2n}}(\sqrt{|\mathcal{D}_{FK}| log(2en/|\mathcal{D}_{FK}|)} - \sqrt{q_{No} log(2en/q_{No})})$$

Fourth, let q_R^* denote the minimum possible number of unique values of $\mathbf{U}_R$ in $\mathbf{R}$, i.e., $q_R^* = min_{F \in X_R} |\mathcal{D}_F|$. Now, we observe that since $q_{No} \leqslant |\mathcal{D}_{FK}| \leqslant n$, the RHS above is a non-increasing function of q_{No} that is maximum when $q_{No} = q_R^*$. Thus, finally, we get the following inequality:

$$\text{ROR} \leqslant \frac{1}{\delta\sqrt{2n}}(\sqrt{|\mathcal{D}_{FK}|log(2en/|\mathcal{D}_{FK}|)} - \sqrt{q_R^* log(2en/q_R^*)})$$

Intuitively, the above represents the worst-case scenario in which $\mathbf{U}_S$ is empty and $\mathbf{U}_R = \{\text{argmin}_{F \in \mathbf{X}_R}|\mathcal{D}_F|\}$. Thus, we call this bound the *worst-case* ROR, and its "gap" with the exact ROR could be large (Figure 5.1). Henceforth, we use the term "ROR" to refer to this worst-case upper bound. The ROR rule uses a threshold: given a parameter ρ, avoid the join if ROR $\leqslant \rho$.[8] This rule is conservative because given a join that is (δ, ρ)-safe to avoid, there might be some $\rho' < \rho$ such that the join is actually also (δ, ρ')-safe to avoid.

The TR Rule

The ROR rule still requires us to look at $\mathbf{X}_R$ to ascertain the features' domains and obtain q_R^*. This motivates us to consider an even simpler rule that depends only on n_S (number of training examples in $\mathbf{S}$) and $|\mathcal{D}_{FK}|$ ($= n_R$, by definition). We call $\frac{n_S}{n_R}$ the *tuple ratio* (TR). The key advantages of the TR over the ROR are that this is easier to understand and implement and that this does not even require us to look at $\mathbf{X}_R$, i.e., this enables us to ignore the join without even looking at $\mathbf{R}$. We now explain the relationship between the TR and the ROR.

When $|\mathcal{D}_{FK}| \gg q_R^*$, the ROR can be approximated as follows:

$$\text{ROR} \approx \frac{\sqrt{|\mathcal{D}_{FK}|log(2en/|\mathcal{D}_{FK}|)}}{\delta\sqrt{2n}}$$

Since $n \equiv n_S$, we also have ROR $\approx (1/\sqrt{n_S/n_R})(\frac{\sqrt{log(2en_S/n_R)}}{\delta\sqrt{2}})$, which is approximately linear in $(1/\sqrt{TR})$ for a reasonably large TR. Thus, the TR is a conservative simplification of the ROR. The TR rule applies a threshold on the TR to predict if it is safe to avoid join: given a parameter τ, the join is avoided if TR $\geqslant \tau$. Note that since the TR rule is more conservative, it might lead to more missed opportunities than the ROR rule (Figure 5.1). We now explain why this "gap" arises. The key reason is that the TR cannot distinguish between scenarios where $|\mathcal{D}_{FK}| \gg q_R^*$ and where $|\mathcal{D}_{FK}|$ is comparable to q_R^*, as illustrated by Figure 5.4. When $|\mathcal{D}_{FK}| \gg q_R^*$, the ROR is high, which means the join may not be safe to avoid. But when $|\mathcal{D}_{FK}| \approx q_R^*$, the ROR is low, which means the join may be safe to avoid. The TR is oblivious to this finer distinction enabled by the ROR. But in practice, we expect that this extra capability of the ROR might not be too significant since it only matters if *all* features in $\mathbf{X}_R$ have domain sizes comparable to FK. Such an extreme situation is perhaps

[8] We set the failure probability δ to be always 0.1, but it can also be folded into ρ since ROR $\propto \frac{1}{\delta}$.

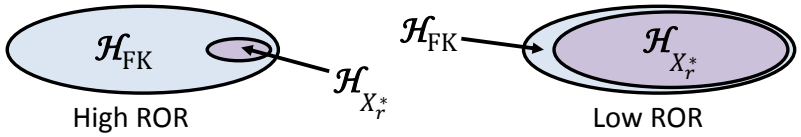


Figure 5.4: When $q_R^* = |\mathcal{D}_{X_r^*}| \ll |\mathcal{D}_{FK}|$, the ROR is high. When $q_R^* \approx |\mathcal{D}_{FK}|$, the ROR is low. The TR rule cannot distinguish between these two scenarios.

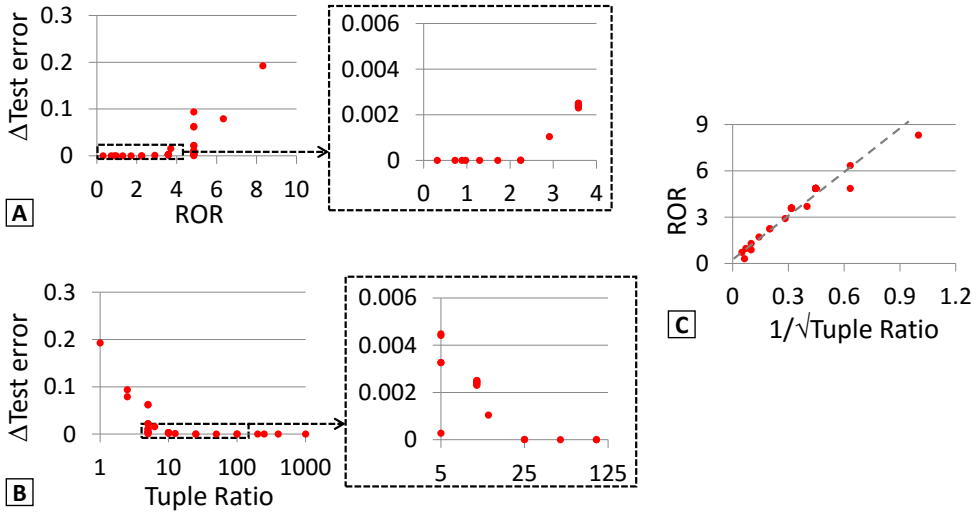


Figure 5.5: Scatter plots based on all the results of the simulation experiments referred to by Figure 5.3. (A) Increase in test error caused by avoiding the join (denoted "ΔTest error") against ROR. (B) ΔTest error against tuple ratio. (C) ROR against inverse square root of tuple ratio.

unlikely in the real world. In fact, as we explain later in Section 5.3, the ROR rule and the TR rule yielded identical results for join avoidance on all our real datasets.

Tuning the Thresholds

We now explain how to tune the thresholds of our decision rules (ρ for ROR and τ for TR). For our purposes, we define a "significant increase" in test error as an absolute increase of 0.001. This might be too strict (or lenient) based on the application. Our goal here is only to demonstrate the feasibility of tuning our rules. Applications willing to tolerate a higher (or lower) error can easily retune the rules based on our methodology.

Figure 5.5(A) shows a scatter plot of the (asymmetric) test error difference between *NoJoin* and *UseAll* based on our diverse set of simulation results (varying n_S, $|\mathcal{D}_{FK}|$, d_R, etc.) for the first scenario: a lone $X_r \in \mathbf{X}_R$ is part of the true distribution. We see that as the ROR increases, the test error difference increases, which confirms that the ROR is an

indicator of the test error difference. In fact, for sufficiently small values of the ROR, the test error is practically unchanged. The zoomed-in portion of Figure 5.5(A) suggests that a threshold of $\rho = 2.5$ is reasonable. Figure 5.5(B) shows the same errors against the TR. We see that as the TR increases, the test error difference decreases. For sufficiently large values of the TR, the test error is practically unchanged. Thus, even the more conservative TR can be a surprisingly good indicator of the test error difference. The zoomed-in portion of Figure 5.5(B) suggests a threshold of $\tau = 20$ is reasonable. Finally, Figure 5.5(C) confirms the relationship between the ROR and the TR: the ROR is approximately linear in $1/\sqrt{\text{TR}}$ (Pearson correlation coefficient ≈ 0.97).

These thresholds need to be tuned only *once per ML model* (more precisely, *once per VC dimension expression*). Thus, they are qualitatively different from (hyper-)parameters in ML that need to be tuned *once per dataset instance* using cross-validation. The above threshold values can be directly used in practice for models such as Naive Bayes and logistic regression. In fact, they worked unmodified for *all* the real datasets in our experiments for both Naive Bayes and logistic regression! If the error tolerance is changed, one can use our simulation results to get new thresholds. For an ML model with a completely different VC dimension expression, our simulations have to be repeated with that ML model and new thresholds obtained in a manner similar to the above.

Foreign Key Skew We now briefly explain what happens if we relax the assumption that the foreign key distribution is not skewed. Note that neither ROR nor TR capture skew in $P(FK)$. We studied this issue with more simulations. For the sake of readability, we only provide the key results here and provide more details in the appendix. Overall, we found that skew in $P(FK)$ in and of itself is not what matters for ML accuracy, but rather whether $P(Y)$ is also skewed, and whether these two skews "collude". We call such skews "malign" and found that a simple way to account for them is to check if $H(Y)$ is too low. We leave more complex approaches to handle malign skews to future work.

Multiple Attribute Tables It is trivial to extend the TR rule to multiple $\mathbf{R}_i$: avoid the join with $\mathbf{R}_i$ if $\frac{n_S}{n_{R_i}} \geqslant \tau$. As for the ROR rule, since $\mathbf{U}_S$ was eliminated when deriving the worst-case ROR, we can ignore the other foreign keys in $\mathbf{S}$. Thus, we can avoid the join with $\mathbf{R}_i$ if the ROR computed using FK_i and $\min_{F \in \mathbf{X}_{R_i}} |\mathcal{D}_F|$ is $\leqslant \rho$. Making join avoidance decisions for multiple $\mathbf{R}_i$ jointly, rather than independently as we do, might yield less conservative (albeit more complex) decision rules. We leave this to future work.

Multi-Class Case The VC dimension makes sense only when Y has two classes, which might limit the applicability of the ROR rule. In the ML literature, there are various

Dataset	$\#Y$	(n_S, d_S)	k	k'	$(n_{R_i}, d_{R_i}), i = 1\ to\ k$
Walmart	7	(421570, 1)	2	2	(2340, 9), (45, 2)
Expedia	2	(942142, 6)	2	1	(11939, 8), (37021, 14)
Flights	2	(66548, 20)	3	3	(540, 5), (3182, 6), (3182, 6)
Yelp	5	(215879, 0)	2	2	(11537, 32), (43873, 6)
MovieLens1M	5	(1000209, 0)	2	2	(3706, 21), (6040, 4)
LastFM	5	(343747, 0)	2	2	(4999, 7), (50000, 4)
BookCrossing	5	(253120, 0)	2	2	(49972, 4), (27876, 2)

Table 5.2: Dataset statistics. $\#Y$ is the number of target classes. k is the number of attribute tables. k' is the number of foreign keys with closed domains.

generalizations of the VC dimension for multi-class classifiers, e.g., the Natarajan dimension or the graph dimension [Shalev-Shwartz and Ben-David, 2014]. Intuitively, they generalize the notion of the power of the classifier by also considering the number of classes. However, it is known that these more general dimensions are bounded (for "linear" classifiers such as Naive Bayes or logistic regression) by a log-linear factor in the product of the total number of feature values (sum of the domain sizes for nominal features) and the number of classes [Daniely et al., 2012]. Hence, intuitively, the ROR rule might be a stricter condition than needed for the multi-class case, which is in line with our guiding principle of conservatism for avoiding joins. We leave a deeper analysis of the multi-class case to future work.

5.3 Experiments on Real Data

Our goal here is three-fold: (1) verify that there are cases where avoiding joins does not increase error significantly, (2) verify that our rules can accurately predict those cases, and (3) analyze the robustness and sensitivity of our rules.

Datasets Standard sources such as the UCI ML repository did not have datasets with known KFKDs/FDs. Thus, we obtained real datasets from other sources: Walmart, Expedia, and Yelp are from the data science contest portal Kaggle (`www.kaggle.com`); MovieLens1M and BookCrossing are from GroupLens (`grouplens.org`); Flights is from `openflights.org`; LastFM is from `mtg.upf.edu/node/1671` and `last.fm`. Table 5.2 provides the dataset statistics. We describe each dataset and the prediction task. We used a standard unsupervised binning technique (equal-length histograms) for numeric features. All of these datasets are available on our project webpage: `http://pages.cs.wisc.edu/~arun/hamlet` and all of our codes for data preparation are available on GitHub:

`https://github.com/arunkk09/hamlet`. To the best of our knowledge, this is the first work to gather and clean so many normalized real datasets for ML. We hope our efforts help further research on this topic.

Walmart. Predict department-wise sales levels by joining data about past sales with data about stores and weather/economic indicators. $\mathbf{S}$ is Sales (SalesLevel, IndicatorID, StoreID, Dept), while Y is SalesLevel. $\mathbf{R}_1$ is Indicators (IndicatorID, TempAvg, TempStdev, CPIAvg, CPIStdev, FuelPriceAvg, FuelPriceStdev, UnempRateAvg, UnempRateStdev, IsHoliday). $\mathbf{R}_2$ is Stores (StoreID, Type, Size). Both foreign keys (StoreID and IndicatorID) have closed domains with respect to the prediction task.

Expedia. Predict if a hotel will be ranked highly by joining data about past search listings with data about hotels and search events. $\mathbf{S}$ is Listings (Position, HotelID, SearchID, Score1, Score2, LogHistoricalPrice, PriceUSD, PromoFlag, OrigDestDistance). Y is Position. $\mathbf{R}_1$ is Hotels (HotelID, Country, Stars, ReviewScore, BookingUSDAvg, BookingUSDStdev, BookingCount, BrandBool, ClickCount). $\mathbf{R}_2$ is Searches (SearchID, Year, Month, WeekOfYear, TimeOfDay, VisitorCountry, SearchDest, LengthOfStay, ChildrenCount, AdultsCount, RoomCount, SiteID, BookingWindow, SatNightBool, RandomBool). HotelID has a closed domain with respect to the prediction task, while SearchID does not.

Flights. Predict if a route is codeshared by joining data about routes with data about airlines and airports (both source and destination). $\mathbf{S}$ is Routes (CodeShare, AirlineID, SrcAirportID, DestAirportID, Equipment1, ..., Equipment20). Y is CodeShare. $\mathbf{R}_1$ is Airlines (AirlineID, AirCountry, Active, NameWords, NameHasAir, NameHasAirlines). $\mathbf{R}_2$ is SrcAirports (SrcAirportID, SrcCity, SrcCountry, SrcDST, SrcTimeZone, SrcLongitude, SrcLatitude). $\mathbf{R}_3$ is DestAirports (DestAirportID, DestCity, DestCountry, DestTimeZone, DestDST, DestLongitude, DestLatitude). All foreign keys have closed domains with respect to the prediction task.

Yelp. Predict business ratings by joining data about past ratings with data about users and businesses. $\mathbf{S}$ is Ratings (Stars, UserID, BusinessID). Y is Stars. $\mathbf{R}_1$ is Businesses (BusinessID, BusinessStars, BusinessReviewCount, Latitude, Longitude, City, State, WeekdayCheckins1, ..., WeekdayCheckins5, WeekendCheckins1, ..., WeekendCheckins5, Category1, ... Category15, IsOpen). $\mathbf{R}_2$ is Users (UserID, Gender, UserStars, UserReviewCount, VotesUseful, VotesFunny, VotesCool). Both foreign keys have closed domains with respect to the prediction task.

MovieLens1M. Predict movie ratings by joining data about past ratings with data about users and movies. $\mathbf{S}$ is Ratings (Stars, UserID, MovieID). Y is Stars. $\mathbf{R}_1$ is Movies (MovieID, NameWords, NameHasParentheses, Year, Genre1, ..., Genre18). $\mathbf{R}_2$ is Users (UserID,

Gender, Age, Zipcode, Occupation). Both foreign keys have closed domains with respect to the prediction task.

LastFM. Predict music play levels by joining data about past play levels with data about users and artists. **S** is Plays (PlayLevel, UserID, ArtistID), *Y* is PlayLevel, $\mathbf{R}_1$ is Artists (ArtistID, Listens, Scrobbles, Genre1, ..., Genre5), and $\mathbf{R}_2$ is Users (UserID, Gender, Age, Country, JoinYear). Both foreign keys have closed domains with respect to the prediction task.

BookCrossing. Predict book ratings by joining data about past ratings with data about readers and books. **S** is Ratings (Stars, UserID, BookID). *Y* is Stars. $\mathbf{R}_1$ is Users (UserID, Age, Country). $\mathbf{R}_2$ is Books (BookID, Year, Publisher, NumTitleWords, NumAuthorWords). Both foreign keys have closed domains with respect to the prediction task.

Experimental Setup All experiments were run on CloudLab, which offers free and exclusive access to physical compute nodes for research [Ricci et al., 2014]. We use their default ARM64 OpenStack Juno profile with Ubuntu 14.10. It provides an HP Proliant server with 8 ARMv8 cores, 64 GB RAM, and 100 GB disk. Our code is written in R (version 3.1.1), and all datasets fit in memory as R data frames.

End-to-end Error and Runtime

We compare two approaches: *JoinAll*, which joins all base tables, and *JoinOpt*, which joins only those base tables predicted by the TR rule to be not safe to avoid (the ROR rule gave identical results). For each approach, we pair Naive Bayes with one of four popular feature selection methods: two wrappers (forward selection, and backward selection) and two filters (mutual information-based and information gain ratio-based). We compare only the runtimes of feature selection and exclude the time taken to join the tables. This can work against *JoinOpt* but as such, the joins took $< 1\%$ of the total runtime in almost all our results. As mentioned before, we use the standard holdout validation method with the entity table (**S**) split randomly into 50%:25%:25% for training, validation, and final holdout testing. For the filter methods, the number of features filtered after ranking was actually tuned using holdout validation as a "wrapper."

In order to make the comparison more meaningful, the error metric used to report the holdout test error of the learned model depends on the type of the target and the number of classes. Specifically, the zero-one error is used for Expedia and Flights, which have binary targets, while the root mean squared error (RMSE) is used for the other datasets, which have multi-class ordinal targets. Note that our goal is to check if *JoinOpt* avoided any

	Walmart		Expedia		Flights		Yelp	
Error Metric	**RMSE**		**Zero-one**		**Zero-one**		**RMSE**	
Approach	JoinAll	**JoinOpt**	JoinAll	**JoinOpt**	JoinAll	**JoinOpt**	JoinAll	**JoinOpt**
# Tables in i/p	3	1	3	2	4	3	3	3
# Features in i/p	14	3	29	21	40	35	40	40
Error with full set	0.9850	0.8927	0.2820	0.2417	0.2445	0.1699	1.3279	1.3279
Forward Selection								
Validation error	0.8927	0.8927	0.2338	0.2338	0.1390	0.1390	1.1361	1.1361
# Features in o/p	3	3	7	7	13	13	9	9
Holdout test error	0.8910	0.8910	0.2336	0.2336	0.1359	0.1359	1.1317	1.1317
Backward Selection								
Holdout test error	0.8961	0.8910	0.2337	0.2337	0.1390	0.1354	1.1330	1.1330
Mutual Information-based Filter								
Holdout test error	0.8910	0.8910	0.2339	0.2339	0.1610	0.1610	1.1676	1.1676
Information Gain Ratio-based Filter								
Holdout test error	0.9014	0.8910	0.2401	0.2352	0.1476	0.1476	1.1423	1.1423

	MovieLens1M		LastFM		BookCrossing	
Error Metric	**RMSE**		**RMSE**		**RMSE**	
Approach	JoinAll	**JoinOpt**	JoinAll	**JoinOpt**	JoinAll	**JoinOpt**
# Tables in i/p	3	1	3	2	3	3
# Features in i/p	27	2	13	6	8	8
Error with full set	1.1671	1.0671	1.2344	1.2145	1.4678	1.4678
Forward Selection						
Validation error	1.0670	1.0671	1.0247	1.0247	1.4313	1.4313
# Features in o/p	3	2	1	1	2	2
Holdout test error	1.0687	1.0685	1.0248	1.0248	1.4295	1.4295
Backward Selection						
Holdout test error	1.0693	1.0685	1.0475	1.0248	1.4422	1.4422
Mutual Information-based Filter						
Holdout test error	1.0685	1.0685	1.0248	1.0248	1.4327	1.4327
Information Gain Ratio-based Filter						
Holdout test error	1.0692	1.0685	1.0248	1.0248	1.4327	1.4327

Figure 5.6: End-to-end results on real data: Error after feature selection.

joins, and if so, whether its error is much higher than *JoinAll*. Figures 5.6 and 5.7 present the accuracy and runtime results respectively.

Errors and Output Features The results in Figure 5.6 validate our key claim: *JoinOpt* did avoid some joins, and in all the cases where it did, the holdout test error did not increase significantly. For example, *JoinOpt* avoided both joins in both Walmart and MovieLens1M ("#Tables in input") without any increase in error. On Expedia, Flights, and LastFM, only one join each was avoided by *JoinOpt*, and the error did not increase much here either. Finally, on Yelp and BookCrossing, none of the joins were predicted to be safe to avoid. Furthermore, the trends are the same irrespective of the feature selection method used. In general, sequential greedy search had lower errors than the filter methods, which is

consistent with the literature [Guyon et al., 2006]. Surprisingly, in 12 of the 20 results (4 methods $\times$ 5 datasets; Yelp and BookCrossing excluded), *JoinOpt* and *JoinAll* had *identical* errors! This is because the output feature sets were identical even though the input for *JoinOpt* was smaller. For example, both *JoinAll* and *JoinOpt* had selected the same 3 features on Walmart for both forward selection and MI-based filter: {IndicatorID, StoreID, Dept}. Thus, none of the foreign features seem to matter for accuracy here. Similarly, on Expedia, the outputs were identical for forward selection: {HotelID, Score2, RandomBool, BookingWindow, Year, ChildrenCount, SatNightBool}, and backward selection but with 12 features. Thus, the HotelID sufficed and the hotel's features were not needed. On LastFM, for all methods except backward selection, both *JoinAll* and *JoinOp*t returned only {UserID}. It seems even ArtistID does not help (not just the artist's features), but our rules are not meant to detect this. For the sake of readability, we provide all other output features in the appendix.

In 3 of the 20 results, *JoinOpt* had almost the same error as *JoinAll* but with a different output feature set. For example, on MovieLens1M, for forward selection *JoinOpt* gave {UserID, MovieID}, while *JoinAll* also included a movie genre feature. More surprisingly, the error was actually significantly *lower* for *JoinOpt* in 5 of the 20 results (e.g., backward selection on Walmart and LastFM). This lower error is a serendipity caused by the variability introduced by the heuristic nature of the feature selection method. Since these feature selection methods are not globally optimal, *JoinAll*, which uses all the redundant features, seems to face a higher risk of being stuck at a poor local optimal. For example, for backward selection on Walmart, *JoinAll* dropped IndicatorID but retained many store and weather features even though it was less helpful for accuracy.

Runtime The runtime speedup depends on the ratio of the number of features used by *JoinAll* against *JoinOpt*: the more features avoided, the higher the speedup. Hence, the speedup depends on the number of features in the base tables, as Figure 5.7 shows. Since *JoinOpt* avoided both joins on MovieLens1M and Walmart, the speedups were high: 186x and 82x resp. for backward selection; 26x and 15x resp. for the filter methods. But on Expedia, the ratio of the number of features used is lower ($\approx$ 1.4), which yielded more modest speedups of between 1.5x to 2.8x. Similarly, the speedups are lower on Flights and LastFM. It is noteworthy that these datasets cover the entire spectrum in terms of how many joins can be avoided: Walmart and MovieLens1M on one end; Yelp and BookCrossing on the other.

Summary In all 28 results, *JoinOpt* had either identical or almost the same error as *JoinAll* but was often significantly faster, thus validating our core claim.

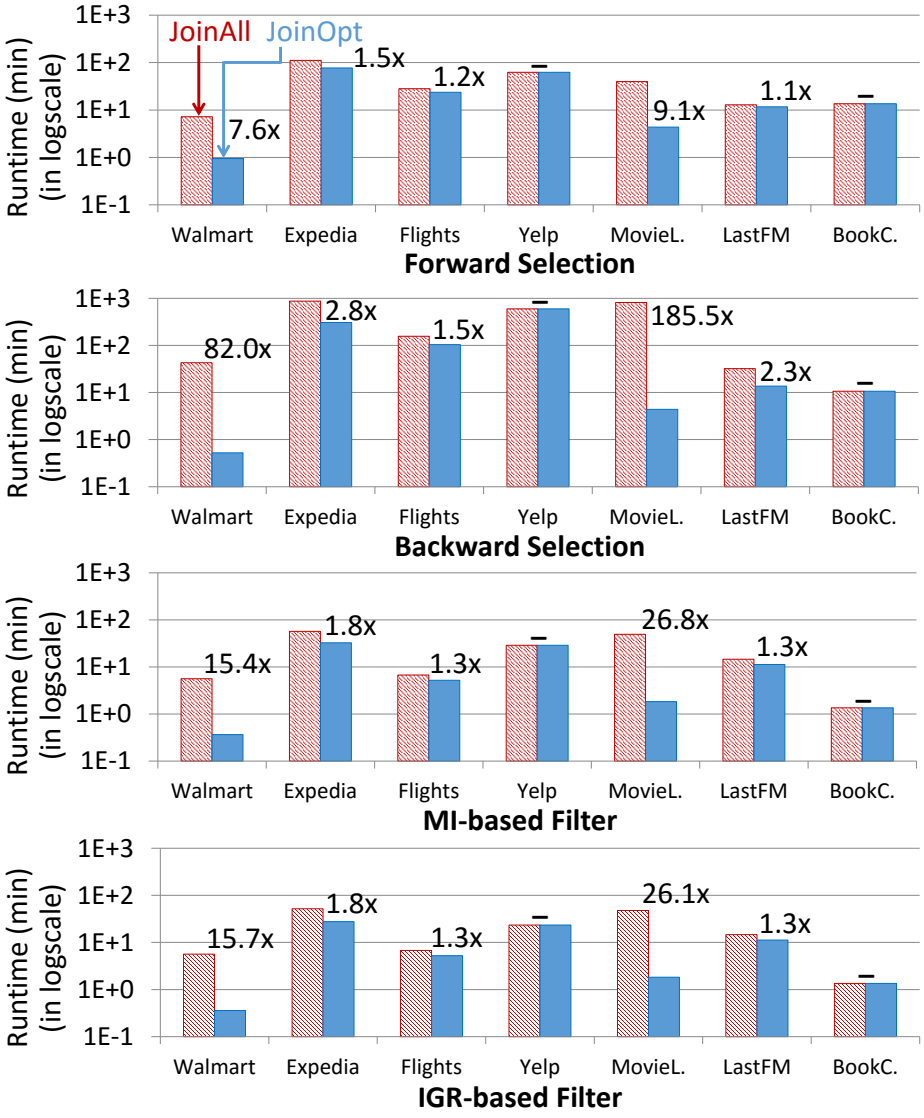


Figure 5.7: End-to-end results on real data: Runtime of feature selection.

Drill-down on Real Data

Robustness of Join Avoidance Decisions

JoinOpt avoids only the joins predicted by the TR rule as being safe to avoid. We would like to understand the robustness of its decisions, i.e., what if we had avoided a different subset of the joins? We focus only on sequential greedy search for brevity sake. Figure 5.8 presents the results. Expedia is absent because it has only one foreign key with a closed domain and so, Figure 5.6 suffices for it.

Walmart

	JoinAll	No R1	No R2	*NoJoins*
FS	0.8910	0.8910	0.8910	0.8910
BS	0.8961	0.8910	0.8910	0.8910

MovieLens1M

	JoinAll	No R1	No R2	*NoJoins*
FS	1.0687	1.0685	1.0687	1.0685
BS	1.0693	1.0685	1.0699	1.0685

BookCrossing

	JoinAll	No R1	No R2	NoJoins
FS	1.4295	1.4295	1.4327	1.4327
BS	1.4422	1.4563	1.4422	1.4327

Yelp

	JoinAll	No R1	No R2	NoJoins
FS	1.1317	1.2350	1.2052	1.3322
BS	1.1330	1.2145	1.2446	1.3322

LastFM

	JoinAll	*No R1*	No R2	NoJoins
FS	1.0248	1.0248	1.0248	1.0248
BS	1.0475	1.0248	1.0456	1.0248

Flights

	JoinAll	*No R1*	No R2	No R3	No R1,R2	No R1,R3	No R2,R3	NoJoins
FS	0.1359	0.1359	0.1359	0.1359	0.1359	0.1359	0.1359	0.1359
BS	0.1390	0.1354	0.1363	0.1363	0.1354	0.1387	0.1363	0.1354

Figure 5.8: Robustness: Holdout test errors after Forward Selection (FS) and Backward Selection (BS). The "plan" chosen by JoinOpt is highlighted, e.g., NoJoins on Walmart.

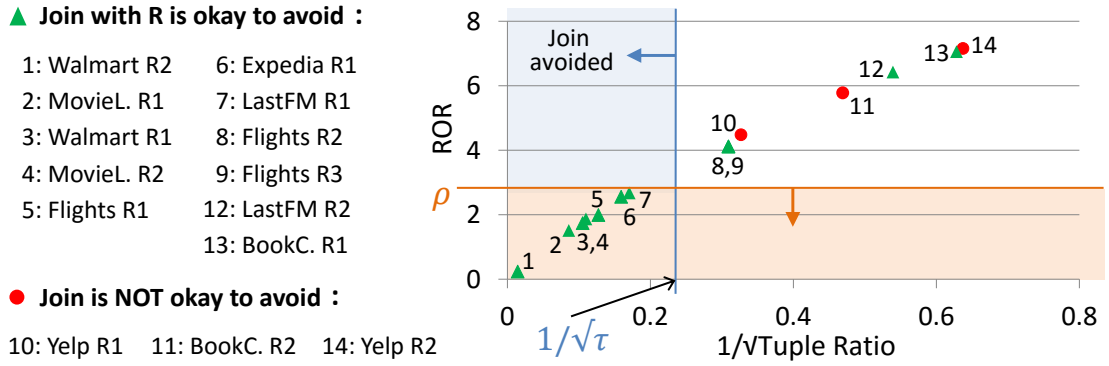


Figure 5.9: Sensitivity: We set $\rho = 2.5$ and $\tau = 20$. An attribute table is deemed "okay to avoid" if the increase in error was within 0.001 with either Forward Selection (FS) and Backward Selection (BS).

On Walmart and MovieLens1M, it was safe to avoid both joins. Our rule predicted this correctly. At the opposite end, avoiding either join in Yelp and BookCrossing blows up the error. Our rule predicted this correctly as well. This shows the need for decision rules such as ours: the state-of-the-art *JoinAll* misses the speedup opportunities on Walmart and MovieLens1M, while its naive opposite *NoJoins* causes the error to blow up on Yelp and BookCrossing. On Flights, our rule predicted that the Airlines table is safe to avoid but not the other two attribute tables. Yet, it turned out that even the other two could have been avoided. This is an instance of the "missed opportunities" we had anticipated; recall that our rules are conservative (Figure 5.1). LastFM and BookCrossing also have an attribute table each that was deemed not safe to avoid, but it turned out that they could have been avoided. However, the results for *JoinAll* suggest that the features from those tables were

FS	Walmart	Expedia	Flights	Yelp	MovieL.	LastFM	BookC.
JoinOpt	0.8910	0.2336	0.1359	1.1317	1.0685	1.0248	1.4295
JoinAllNoFK	0.9477	0.2800	0.1688	1.1317	1.1940	1.5375	1.5347
BS	**Walmart**	**Expedia**	**Flights**	**Yelp**	**MovieL.**	**LastFM**	**BookC.**
JoinOpt	0.8910	0.2337	0.1354	1.1330	1.0685	1.0248	1.4422
JoinAllNoFK	0.9455	0.2760	0.1817	1.1381	1.1806	1.5863	1.5347

Table 5.3: Holdout test errors of JoinOpt and JoinAllNoFK, which drops all foreign keys a priori. FS is Forward Selection. BS is Backward Selection.

not useful for accuracy anyway because feature selection removed them; exploiting such opportunities are beyond the scope of this work.

Sensitivity to Thresholds

The TR rule uses $\tau = 20$ based on our simulation results. Similarly, we had picked $\rho = 2.5$ for the ROR rule. We now validate the sensitivity of the rules by comparing the threshold settings with the actual TR and ROR values from the data. Figure 5.9 shows the results.

We see that the ROR is almost linear in the inverse square root of the TR even on the real data. Many attribute tables of Walmart, MovieLens1M, Flights, and Expedia lie below the thresholds chosen for either rule. They were all correctly predicted to be safe to avoid. Both attribute tables of Yelp and one of BookCrossing were correctly predicted to be not safe to avoid. The others fall in between and include some "missed opportunities." But note that there is no case in which avoiding an attribute table that was deemed safe to avoid by the TR rule caused the error to blow up.

Finally, we also tried a higher error tolerance of 0.01 instead of 0.001. This yielded new thresholds $\tau = 10$ and $\rho = 4.2$ based on Figure 5.5. This setting correctly predicted that two more joins could be avoided (both on Flights).

What if Foreign Keys Are Dropped?

Data scientists sometimes judge foreign keys as being too "uninterpretable" and simply drop them. To assess the impact of this choice, we compare JoinOpt (whose accuracy is similar to JoinAll) with JoinAllNoFK, which is similar to JoinAll, but drops all foreign keys a priori. Table 5.3 shows the results. In 6 of the 7 datasets, dropping foreign keys proved to be catastrophic for accuracy for both forward and backward selection. As we explained in our theoretical analysis in Section 5.1 (Figure 5.2), this is primarily because dropping foreign keys (JoinAllNoFK) might cause the bias to blow up drastically, while JoinOpt does not increase the bias.

	Walmart-10%			
	JoinAll	**No R1**	***No R2***	**No Joins**
Forward Selection	0.9006	0.9090	0.9006	0.9242
Backward Selection	0.9006	0.9090	0.8991	0.9242
	Yelp-10%			
	JoinAll	**No R1**	***No R2***	**No Joins**
Forward Selection	1.0775	1.0774	1.0775	1.1421
Backward Selection	1.1037	1.1028	1.0780	1.1421

Table 5.4: Effects of row sampling on join avoidance.

Effects of Row Sampling

We conduct one more experiment to demonstrate an interesting behavior of foreign key features when row sampling is performed. We use two "opposite extreme" datasets: Walmart and Yelp. We down-sample Walmart to 10% of its size uniformly randomly, but retain the original domains of the foreign keys. We call this dataset *Walmart-10%*. Intuitively, such a sampling will increase the variance for models that include foreign keys, and thus, make it less safe to avoid the join. We down-sample Yelp in a way that retains only the top 10% of the users and businesses (based on frequency). Thus, the domains of the foreign keys are altered. This is not far-fetched because Yelp Inc. itself probably performed a similar sampling of its users and businesses before releasing their dataset to Kaggle. The resultant sample turned out to be 35% of the full labeled set; we call this dataset *Yelp-10%*. We run our accuracy experiments on these two datasets. Table 5.4 presents the results.

We see that the "optimal plan" has now changed compared to Figure 5.8. Whereas both attribute tables were safe to avoid on the original Walmart dataset, only the Stores table is safe to avoid on Walmart-10%. This is because the TR for the Indicators table fell from 90 to just 9 and our rule correctly predicted that avoiding it might cause the error to increase. Thus, this result exposes an interesting danger in using row sampling as a way to get a quick estimate when large-domain features such as foreign keys are present: *the results of feature selection might be significantly different with and without row sampling*.

On the other extreme, whereas neither attribute table was safe to avoid on Yelp, it turns out that either attribute table at a time (but not both together) is safe to avoid on Yelp-10%. This is because the TRs of the attribute tables increased and the variance decreased. Interestingly, avoiding both tables together increases the error significantly, which suggests that there might be more complex relationships between the features from both attributes tables. It is interesting future work to design more complex decision rules that can model and capture such effects that straddle multiple tables.

Dataset	Error Metric	L1 Regularization		L2 Regularization	
		JoinAll	**JoinOpt**	JoinAll	**JoinOpt**
Walmart	RMSE	0.8335	0.8335	1.1140	0.9370
Expedia	Zero-one	0.2130	0.2134	0.2559	0.2632
Flights	Zero-one	0.1167	0.1183	0.1494	0.1485
Yelp	RMSE	1.1426	1.1426	1.1744	1.1744
MovieLens1M	RMSE	1.0444	1.0459	1.1438	1.1940
LastFM	RMSE	1.0308	1.0300	1.5629	1.5605
BookCrossing	RMSE	1.3993	1.3993	1.6561	1.6561

Table 5.5: Holdout test errors for logistic regression with regularization for the same setup as Figure 5.6.

Other ML Classifiers

It is natural to wonder if the trends observed on Naive Bayes would translate to other ML models. Note that the theoretical results and arguments in Section 5.1 apply to ML classifiers in general. Nevertheless, we now discuss another popular classifier: logistic regression, followed by one more classifier: Tree-Augmented Naive Bayes (TAN).

Unlike Naive Bayes, the most popular feature selection method for logistic regression is the embedded method of *regularization* that constrains the L1 or L2 norm of the coefficient vector [Hastie et al., 2003; Guyon et al., 2006]. We consider both and use the well-tuned implementation of logistic regression from the popular *glmnet* library in R [Friedman et al., 2010]. Table 5.5 presents the results. We see that the errors of *JoinOpt* are comparable to those of *JoinAll* with L1. Thus, the trends are the same as for Naive Bayes, which agrees with our theoretical results. Interestingly, L2 errors are significantly higher than L1 errors. This is an artefact of the data existing in a sparse feature space for which L1 is known to be usually better than L2 [Hastie et al., 2003].

TAN strikes a balance between the efficiency of Naive Bayes and the expressiveness of general Bayesian networks [Friedman et al., 1997]. TAN searches for conditional dependencies among pairs of features in $\mathbf{X}$ given Y using mutual information to construct a tree of dependencies on $\mathbf{X}$. Surprisingly, TAN might actually be *less* accurate than Naive Bayes on datasets with the KFKDs we study because TAN might not even use $\mathbf{X}_R$. Intuitively, this is because the FD $FK \rightarrow \mathbf{X}_R$ causes all features in $\mathbf{X}_R$ to be dependent on FK in the tree computed by TAN. This leads to $\mathbf{X}_R$ participating only via unhelpful Kronecker delta distributions, viz., $P(\mathbf{X}_R|FK)$. Depending on how structure learning is done, general Bayesian networks could face this issue too. We leave techniques to solve this issue to future work.

Discussion: Implications for Data Scientists

Our work presents at least three major practical implications. These are based on our conversations with data scientists at multiple settings – a telecom company, a Web company, and an analytics vendor – about our results.

1. Data scientists often join all tables almost by instinct. Our work shows that this might lead to much poorer performance without much accuracy gain. Avoiding joins that are safe to avoid can speed up the exploratory process of comparing feature sets and ML models. Our rules, especially the easy-to-understand TR rule, could help with this task.

2. We found many cases in which avoiding some joins led to a counter-intuitive *increase* in accuracy. Thus, at the least, it might be helpful for data scientists to try both *JoinOpt* and *JoinAll*.

3. Data scientists often *drop* all foreign key features (even if they have closed domains, e.g., `StoreID` in Walmart), since they subjectively deem such features as too "uninterpretable." Our work shows that this ad hoc step could seriously hurt accuracy. This finding helps data scientists be better informed of the precise consequences of such a step.

Finally, most of the burden of feature engineering (designing and choosing features) for ML today falls on data scientists [Domingos, 2012; Kandel et al., 2012]. But the database community is increasingly recognizing the need to provide more systems support for feature engineering, e.g., Columbus provides declarative feature selection operations along with a performance optimizer [Zhang et al., 2014; Konda et al., 2013]. We think it is possible for such systems to integrate our decision rules for avoiding joins either as new optimizations or as "suggestions" for data scientists.

5.4 Conclusion: Avoiding Joins Logically

In this era of "big data," it is becoming almost "big dogma" that more features and data are somehow *always* better for machine learning. This project makes a contrarian case that in some situations "less is more." Specifically, using theoretical, simulation, and empirical analyses, we show that in many cases, which can be predicted, features obtained using key-foreign key joins might not improve accuracy much, but degrade runtime performance. Our work opens up new connections between data management, machine learning, and feature selection, and it raises new fundamental research questions at their intersection,

solving which could make it faster and easier for data scientists to use machine learning on multi-table datasets.

Most of the content of this chapter is from our paper titled "To Join or Not to Join? Thinking Twice about Joins before Feature Selection" that appeared in the ACM SIGMOD 2016 conference. All of our codes from this project are available on GitHub: `https://github.com/arunkk09/hamlet`, while the real datasets are available on our project webpage: `http://pages.cs.wisc.edu/~arun/hamlet`.

To join or not to join?
That is the question.
Tuple ratio, we did coin,
Use at your discretion!

---

6 Related Work

6.1 Related Work for Orion

Factorized Computation The abstraction of factorized databases was proposed recently to improve the efficiency of RDBMSs [Bakibayev et al., 2013]. The basic idea is to succinctly 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 trade-off 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 [Rendle, 2013] has also shown that combining features from multiple tables can improve ML accuracy 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 Rendle [2013]. 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, which necessitates a cost model such as ours.

Query Optimization As noted in Bakibayev et al. [2013], factorized computations generalize prior work on optimizing SQL aggregates over joins [Yan and Larson, 1995; Chaudhuri and Shim, 1994]. 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 trade-offs. While we leave SGD for future work, this paper provides a framework for designing and evaluating 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 [Sellis, 1988]. 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 [Hellerstein et al., 2012; Apache, b]. 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 [Zhang et al., 2010; Ghoting et al., 2011; Oracle], systems for Bayesian inference [Cai et al., 2013], systems for graph-based ML [Low et al., 2010], and systems that combine dataflow-based languages for ML with data management platforms [Kumar et al., 2013; Kraska et al., 2013; Zhang et al., 2014]. 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 [Koc and Ré, 2011; Nikolic et al., 2014]. However, neither of those papers address learning over joins. It is interesting future work to study the interplay between these two problems.

6.2 Related Work for Orion Extensions and Generalization

Compressed Query Processing RDBMSs have long integrated data compression schemes to reduce storage and improve query efficiency [Iyer and Wilhite, 1994]. Most of these systems store compressed data on disk and decompress it on the fly when processing queries [Westmann et al., 2000]. Some systems process queries directly over the compressed representation [Chen et al., 2001]. More recently, column-store databases integrate columnar compression with SQL query processing [Abadi et al., 2006]. While our work on compressed clustering is inspired by these systems, to the best of our knowledge, none of these systems integrate clustering algorithms with data compression. We introduce a new

row-wise compression scheme that exploits specific properties of the data redundancy in denormalized tables and modify three popular clustering algorithms to operate directly on our compressed representation. Our empirical analysis also explains subtle trade-offs between compressed clustering and factorized clustering.

Clustering Algorithms There are numerous clustering algorithms [Jain et al., 1999; Fahad et al., 2014]. For the sake of tractability, we focus on three of the most popular clustering algorithms to explain the principles and techniques involved in extending factorized learning to clustering and in creating versions of clustering algorithms that operate directly on our compressed data representation. We leave it to future work to handle other clustering algorithms.

Linear Algebra-based Systems Several systems based on (or incorporating) linear algebra have been built over the last few decades to support ML analytics, e.g., R, Matlab, and SAS. Among these, the open source R is perhaps the most popular, with a rich set of libraries on the Comprehensive R Archive Network [CRAN] and a diverse user base. There has been a lot of research and industrial interest in building systems to improve data processing in R [Sridharan and Patel, 2014] and integrating R (or R-like languages) with large-scale data processing systems such as RDBMSs, Hive/Hadoop, and Spark, e.g., RIOT [Zhang et al., 2010], IBM's SystemML [Ghoting et al., 2011], Oracle R Enterprise [Oracle], and SparkR [Apache, c]. There are also linear algebra-based database systems with their own query language, e.g., SciDB [Cudré-Mauroux et al., 2009]. None of these systems address the problem of optimizing linear algebra operations over normalized data. In a sense, while these systems offer *physical data independence* for linear algebra, our work is the first to bring the notion of *logical data independence* to linear algebra. Since our framework is closed with respect to linear algebra, we think that it can be easily integrated into any of these systems. It is interesting future work to study how to closely integrate our framework with such systems.

6.3 Related Work for Hamlet

Database Dependencies Given a set of FDs, a relation can be decomposed into BCNF (in most cases) [Abiteboul et al., 1995; Beeri and Bernstein, 1979]. Our work deals with the opposite scenario of joins resulting in a relation with FDs. There are database dependencies that are more general than FDs [Abiteboul et al., 1995]. We think it is interesting future work to explore the implications of these more general database dependencies for ML and feature selection.

Graphical ML Models The connection between embedded multi-valued dependencies (EMVDs) and probabilistic graphical models in ML was first shown by Pearl and Verma [1987] and studied further by Wong et al. [1995]. FDs are stronger constraints than EMVDs and cause feature redundancy. We perform a theoretical and empirical analysis of the effects of such redundancy in terms of its implications for avoiding joins.

Feature Selection There is a large body of work in the ML and data mining literature whose focus is designing new feature selection methods to improve ML accuracy [Guyon et al., 2006; Hastie et al., 2003]. Our focus is *not* on designing new feature selection methods but on understanding the effects of joins on feature selection. Our work is orthogonal to the feature selection method used. The redundancy-relevancy trade-off is also well-studied in the ML literature [Guyon et al., 2006; Yu and Liu, 2004; Koller and Sahami, 1995]. There is also some prior work on inferring approximate FDs from the data and using them as part of feature selection [Uncu and Turksen, 2007]. However, all these methods generally focus on approximately estimating redundancy and relevancy using the dataset *instance* [Guyon et al., 2006]. In contrast, our results are based on the *schema*, which enables us to safely avoid features *without even looking at instances*, that is, using just the metadata. To the best of our knowledge, no feature selection method has this radical capability. A technique to "bias" the input and reduce the number of features was proposed by FOCUS [Almuallim and Dietterich, 1992]. At a high level, our rules are akin to a "bias" in the input. But FOCUS still requires expensive computations on the instance, unlike our rules.

Analytics Systems There is growing interest in both industry and academia to more closely integrate ML with data processing [Zhang et al., 2010; Oracle; Ghoting et al., 2011; Cai et al., 2013; Kumar et al., 2013; Kraska et al., 2013; Hellerstein et al., 2012]. In this context, there is a growing recognition that feature engineering is one of the most critical bottlenecks for ML [Anderson et al., 2013; Ré et al., 2014; Zhang et al., 2014; Kumar et al., 2015b,c]. Our work helps make it easier for analysts to apply feature selection over normalized data. We hope our work spurs more research in this important direction.

7 Conclusion and Future Work

In this dissertation, we take a step back and question a widespread practice in advanced analytics – when applying machine learning (ML) over multi-table datasets, data scientists almost always join all the base tables and materialize a single table to use as input to their ML toolkits. This practice of *learning after joins* introduces redundancy in the data, which causes storage and runtime inefficiencies, as well as data maintenance overheads for data scientists. To mitigate such issues, this dissertation asks if such joins are really "needed" and in response, introduces the paradigm of *learning over joins*. We present two orthogonal techniques: *avoiding joins physically*, in which we push ML computations through joins to the base tables, and *avoiding joins logically*, in which we show that in many cases, entire base tables can be, somewhat surprisingly, ignored outright.

We demonstrate the feasibility and efficiency of our technique of avoiding joins physically for a wide variety of ML models, including generalized linear models solved with various optimization methods, probabilistic classifiers, and clustering algorithms, as well as the formal framework of linear algebra. Applying learning theory, we explain why the technique of avoiding joins logically works and devise easy-to-understand decision rules to help data scientists use this technique in practice. Our work forces one to think twice about common data management practices in the ML setting and in the process, opens up new research questions at the intersection of data management and ML.

Future Work Related to Project Orion

Automating Factorized Learning An open question from Project Orion is whether it is possible to automate the application of factorized learning to existing implementations of ML algorithms. Our generalization of factorized learning with linear algebra gives a formal framework for algebraic rewrites and an implementation in R. But it does not help with existing ML code bases in other languages such as C, Java, or Python. These arise in user-defined functions in an RDBMS [Feng et al., 2012], on Hadoop [Apache, b], and on Spark [Kraska et al., 2013]. One could imagine combining static analysis techniques with our framework of rewrite rules to automatically obtain new factorized ML codes on these systems. This would obviate the need for developers to manually reimplement these code bases.

Including Aggregates In some cases, data scientists perform joins before ML that are not key-foreign-key joins but rather key-key joins following pre-aggregations of some features on one of the input tables. These joins do not introduce redundancy in the output but they perform extra data accesses. It would be interesting to consider jointly optimizing ML with these queries that have aggregates along with joins, say, by pushing more of the computations closer to the base tables.

Integrating Linear Algebra and Relational Algebra Our work on pushing linear algebra operators through joins show the utility of jointly considering operations from both algebras. It is interesting future work to explore the interactions of linear algebra operations with other relational operations, devise a single language to express ML and data processing computations, and build a system that optimizes these computations in a joint fashion. There is some related work in this direction that one could build on, especially array stores [Cudré-Mauroux et al., 2009] and R-based analytics systems [Oracle].

Compressed Machine Learning Our work with compressed clustering shows the feasibility and efficiency of integrating ML techniques with compressed data representations. While the benefits of compression have been studied extensively for SQL query processing [Iyer and Wilhite, 1994; Chen et al., 2001], its interplay with ML algorithms is less understood. This is a rich avenue for more future work.

Future Work Related to Project Hamlet

Partial Avoidance A simple extension to Project Hamlet is to avoid a base table partially rather than fully, i.e., ignore only a subset of the features from attribute tables. This introduces a new trade-off space that could help improve performance for datasets on which our decision rules say that none of the joins are safe to avoid.

More Complex ML Models An open question from Project Hamlet is whether the idea of avoiding joins logically would work for more complex ML models such as neural networks, decision trees, and Gaussian kernel SVMs. Recall that our analysis and decision rules assumed that the ML model has a finite VC dimension that is linear in the number of features. These more complex ML models either do not have a closed-form VC dimension or have an infinite VC dimension [Shalev-Shwartz and Ben-David, 2014]. Nevertheless, we suspect that it might still be possible to devise some heuristics for these models that exploit the functional dependencies to short-circuit their learning and feature selection.

More General Database Dependencies Another open question is to formally explain the relationship between database dependencies that are more general than functional dependencies, e.g., multi-valued dependencies, and feature selection and ML algorithms. Understanding this relationship might shed new light on the behavior of feature selection algorithms, most of which are heuristic, and the underlying dependencies between the features in the data.

More General Future Work

Stepping up to a higher level of abstraction, joins before ML are just one part of the process of *feature engineering* for ML [Anderson et al., 2013]. Along with *algorithm selection* and *hyper-parameter tuning*, this constitutes what is known as the iterative process of *model selection* [Mitchell, 1997]. Currently, there is little systems support for the stages of this process beyond faster implementations of ML algorithms. It is an open challenge to provide more principled systems support for this process. Our recently proposed unifying abstraction of *model selection triples* is a step in this direction [Kumar et al., 2015b]. We describe a new kind of advanced analytics system called a *model selection management system* that uses our abstraction and provides more pervasive systems support for the different stages of model selection: defining or reconstructing declarative interfaces for specifying model selection operations, optimizing the specified computations by avoiding redundancy and sharing intermediate data, and defining and managing *ML provenance* to help data scientists consume the results and pose what-if questions. Much work remains to be done in making these ideas more precise and solving the technical challenges that arise. But we believe that this direction of work could potentially make it dramatically easier and faster for data scientists to use ML in data-driven applications.

Abadi, Daniel, Samuel Madden, and Miguel Ferreira. 2006. Integrating Compression and Execution in Column-oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, 671–682. SIGMOD '06, New York, NY, USA: ACM.

Abiteboul, Serge, Richard Hull, and Victor Vianu, eds. 1995. *Foundations of Databases: The Logical Level*. 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Agarwal, Alekh, Oliveier Chapelle, Miroslav Dudík, and John Langford. 2014. A Reliable Effective Terascale Linear Learning System. *Journal of Machine Learning Research (JMLR)* 15:1111–1133.

Aggarwal, Charu C., and Chandan K. Reddy. 2013. *Data clustering: Algorithms and applications*. 1st ed. Chapman & Hall/CRC.

Almuallim, Hussein, and Thomas G. Dietterich. 1992. Efficient Algorithms for Identifying Relevant Features. Tech. Rep., Corvallis, OR, USA.

Anderson, Michael R., Dolan Antenucci, Victor Bittorf, Matthew Burgess, Michael J. Cafarella, Arun Kumar, Feng Niu, Yongjoo Park, Christopher Ré, and Ce Zhang. 2013. Brainwash: A Data System for Feature Engineering. In *Sixth Biennial Conference on Innovative Data Systems Research (CIDR)*.

Apache. a. Apache Hive. `hive.apache.org`.

———. b. Apache Mahout. `mahout.apache.org`.

———. c. SparkR. `spark.apache.org/R`.

Bakibayev, Nurzhan, Tomáš Kočiský, Dan Olteanu, and Jakub Závodný. 2013. Aggregation and Ordering in Factorised Databases. *Proc. VLDB Endow.* 6(14):1990–2001.

Beeri, Catriel, and Philip A. Bernstein. 1979. Computational Problems Related to the Design of Normal Form Relational Schemas. *ACM Trans. Database Syst.* 4(1):30–59.

Cai, Zhuhua, Zografoula Vagena, Luis Perez, Subramanian Arumugam, Peter J. Haas, and Christopher Jermaine. 2013. Simulation of Database-valued Markov Chains Using SimSQL. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 637–648. SIGMOD '13, New York, NY, USA: ACM.

Chaudhuri, Surajit, and Kyuseok Shim. 1994. Including Group-By in Query Optimization. In *Proceedings of the 20th International Conference on Very Large Data Bases*, 354–366. VLDB '94, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Chen, Zhiyuan, Johannes Gehrke, and Flip Korn. 2001. Query Optimization in Compressed Database Systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, 271–282. SIGMOD '01, New York, NY, USA: ACM.

CRAN. Comprehensive R Archive Network. `cran.org`.

Cudré-Mauroux, Philippe, Hideaki Kimura, Kian-Tat Lim, Jennie Rogers, Roman Simakov, Emad Soroush, Pavel Velikhov, Daniel Wang, Magdalena Balazinska, Jacek Becla, David J. DeWitt, Bobbi Heath, David Maier, Samuel Madden, Jignesh M. Patel, Michael Stonebraker, and Stanley B. Zdonik. 2009. A Demonstration of SciDB: A Science-Oriented DBMS. *Proc. VLDB Endow.* 2(2):1534–1537.

Daniely, Amit, Sivan Sabato, and Shai Shalev-Shwartz. 2012. Multiclass Learning Approaches: A Theoretical Comparison with Implications. In *In Neural Information Processing Systems (NIPS)*.

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

Das, Sudipto, Yannis Sismanis, Kevin S. Beyer, Rainer Gemulla, Peter J. Haas, and John McPherson. 2010. Ricardo: Integrating R and Hadoop. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 987–998. SIGMOD '10, New York, NY, USA: ACM.

Domingos, Pedro. 2000. A Unified Bias-Variance Decomposition and its Applications. In *In Proc. 17th International Conf. on Machine Learning (ICML)*, 231–238. Morgan Kaufmann.

———. 2012. A Few Useful Things to Know About Machine Learning. *Commun. ACM* 55(10):78–87.

Domingos, Pedro, and Michael Pazzani. 1997. On the Optimality of the Simple Bayesian Classifier under Zero-One Loss. *Machine Learning* 29(2-3):103–130.

Fahad, Adil, et al. 2014. A survey of clustering algorithms for big data: Taxonomy and empirical analysis. *IEEE Trans. Emerging Topics Comput.* 2(3):267–279.

Feng, Xixuan, Arun Kumar, Benjamin Recht, and Christopher Ré. 2012. Towards a Unified Architecture for in-RDBMS Analytics. In *Proceedings of the 2012 ACM SIGMOD*

International Conference on Management of Data, 325–336. SIGMOD '12, New York, NY, USA: ACM.

Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. 2010. Regularization Paths for Generalized Linear Models via Coordinate Descent. *Journal of Statistical Software* 33(1): 1–22.

Friedman, Nir, Dan Geiger, and Moises Goldszmidt. 1997. Bayesian Network Classifiers. *Machine Learning* 29(2-3):131–163.

Garey, Michael R., and David S. Johnson. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co.

Gartner. Gartner Report on Analytics. `gartner.com/it/page.jsp?id=1971516`.

Getoor, Lise, and Ben Taskar. 2007. *Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning)*. The MIT Press.

Ghoting, Amol, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. 2011. SystemML: Declarative Machine Learning on MapReduce. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, 231–242. ICDE '11, Washington, DC, USA: IEEE Computer Society.

Gray, Jim, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Min. Knowl. Discov.* 1(1):29–53.

Guyon, Isabelle, Steve Gunn, Masoud Nikravesh, and Lotfi A. Zadeh. 2006. *Feature Extraction: Foundations and Applications (Studies in Fuzziness and Soft Computing)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. 2003. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer.

Hellerstein, Joseph M., Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library: Or MAD Skills, the SQL. *Proc. VLDB Endow.* 5(12):1700–1711.

Iyer, Balakrishna R., and David Wilhite. 1994. Data Compression Support in Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, 695–704. VLDB '94, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Jain, Anil K., et al. 1999. Data clustering: A review. *ACM Comput. Surv.* 31(3):264–323.

Kandel, Sean, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2012. Enterprise Data Analysis and Visualization: An Interview Study. In *IEEE Visual Analytics Science & Technology (VAST)*.

Koc, M. Levent, and Christopher Ré. 2011. Incrementally Maintaining Classification Using an RDBMS. *Proc. VLDB Endow.* 4(5):302–313.

Kohavi, Ron, and George H. John. 1997. Wrappers for Feature Subset Selection. *Artif. Intell.* 97(1-2):273–324.

Koller, Daphne, and Mehran Sahami. 1995. Toward Optimal Feature Selection. In *In 13th International Conference on Machine Learning (ICML)*, 284–292.

Konda, Pradap, Arun Kumar, Christopher Ré, and Vaishnavi Sashikanth. 2013. Feature Selection in Enterprise Analytics: A Demonstration Using an R-based Data Analytics System. *Proc. VLDB Endow.* 6(12):1306–1309.

Koren, Yehuda, Robert Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *IEEE Computer* 42(8):30–37.

Kraska, Tim, Ameet Talwalkar, John C. Duchi, Rean Griffith, Michael J. Franklin, and Michael I. Jordan. 2013. MLbase: A Distributed Machine-learning System. In *Sixth Biennial Conference on Innovative Data Systems Research (CIDR)*.

Kumar, Arun, Mona Jalal, Boqun Yan, Jeffrey Naughton, and Jignesh M. Patel. 2015a. Demonstration of Santoku: Optimizing Machine Learning over Normalized Data. *Proc. VLDB Endow.* 8(12):1864–1867.

Kumar, Arun, Robert McCann, Jeffrey Naughton, and Jignesh M. Patel. 2015b. Model Selection Management Systems: The Next Frontier of Advanced Analytics. *SIGMOD Rec.* 44(4):17–22.

Kumar, Arun, Jeffrey Naughton, and Jignesh M. Patel. 2015c. Learning Generalized Linear Models Over Normalized Data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 1969–1984. SIGMOD '15, New York, NY, USA: ACM.

Kumar, Arun, Jeffrey Naughton, Jignesh M. Patel, and Xiaojin Zhu. 2016. To Join or Not to Join? Thinking Twice About Joins Before Feature Selection. In *Proceedings of the 2016 International Conference on Management of Data*, 19–34. SIGMOD '16, New York, NY, USA: ACM.

Kumar, Arun, Feng Niu, and Christopher Ré. 2013. Hazy: Making It Easier to Build and Maintain Big-data Analytics. *ACM Queue* 11(1):30:30–30:46.

Lin, Jimmy, and Alek Kolcz. 2012. Large-scale Machine Learning at Twitter. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 793–804. SIGMOD '12, New York, NY, USA: ACM.

Low, Yucheng, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2010. GraphLab: A New Parallel Framework for Machine Learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*. Catalina Island, California.

Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press.

Mitchell, Thomas M. 1997. *Machine Learning*. 1st ed. New York, NY, USA: McGraw-Hill, Inc.

Nikolic, Milos, Mohammed ElSeidy, and Christoph Koch. 2014. LINVIEW: Incremental View Maintenance for Complex Analytical Queries. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 253–264. SIGMOD '14, New York, NY, USA: ACM.

Nocedal, Jorge, and Stephen J. Wright. 2006. *Numerical Optimization*. 2nd ed. New York, NY: Springer.

Oracle. Oracle R Enterprise. `oracle.com/technetwork/database/database-technologies/r/r-enterprise/overview/index.html`.

Pavlo, Andrew, Carlo Curino, and Stanley Zdonik. 2012. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems. In *Proceedings of the 2012 acm sigmod international conference on management of data*, 61–72. SIGMOD '12, New York, NY, USA: ACM.

Pearl, Judea. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Pearl, Judea, and Thomas Verma. 1987. The Logic of Representing Dependencies by Directed Graphs. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, vol. 1 of *AAAI'87*, 374–379. AAAI Press.

R. Project R. `r-project.org`.

Ramakrishnan, Raghu, and Johannes Gehrke. 2003. *Database Management Systems*. New York, NY, USA: McGraw-Hill, Inc.

Ré, Christopher, Amir Abbas Sadeghian, Zifei Shan, Jaeho Shin, Feiran Wang, Sen Wu, and Ce Zhang. 2014. Feature Engineering for Knowledge Base Construction. *IEEE Data Eng. Bull.* 37(3):26–40.

Rendle, Steffen. 2013. Scaling Factorization Machines to Relational Data. *Proc. VLDB Endow.* 6(5):337–348.

Ricci, Robert, Eric Eide, and the CloudLab Team. 2014. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *;login:* 39(6).

Rish, Irina, et al. 2001. An Analysis of Data Characteristics that Affect Naive Bayes Performance. In *In 19th International Conference on Machine Learning (ICML)*.

SAS. a. Feature Selection and Dimension Reduction Techniques in SAS. `nesug.org/Proceedings/nesug11/sa/sa08.pdf`.

———. b. SAS Report on Analytics. `sas.com/reg/wp/corp/23876`.

Schein, Andrew I., Alexandrin Popescul, Lyle H. Ungar, and David M. Pennock. 2002. Methods and Metrics for Cold-start Recommendations. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 253–260. SIGIR '02, New York, NY, USA: ACM.

Selinger, P. Griffiths, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, 23–34. SIGMOD '79, New York, NY, USA: ACM.

Sellis, Timos K. 1988. Multiple-query Optimization. *ACM Trans. Database Syst.* 13(1): 23–52.

Shalev-Shwartz, Shai, and Shai Ben-David. 2014. *Understanding Machine Learning: From Theory to Algorithms*. New York, NY, USA: Cambridge University Press.

Shapiro, Leonard D. 1986. Join Processing in Database Systems with Large Main Memories. *ACM Trans. Database Syst.* 11(3):239–264.

Silberschatz, Abraham, Henry Korth, and S. Sudarshan. 2006. *Database Systems Concepts*. 5th ed. New York, NY, USA: McGraw-Hill, Inc.

Sridharan, Shriram, and Jignesh M. Patel. 2014. Profiling R on a Contemporary Processor. *Proc. VLDB Endow.* 8(2):173–184.

Uncu, Ozge, and I.B. Turksen. 2007. A Novel Feature Selection Approach: Combining Feature Wrappers and Filters. *Information Sciences* 177(2).

Vapnik, Vladimir N. 1995. *The Nature of Statistical Learning Theory*. New York, NY, USA: Springer-Verlag New York, Inc.

Westmann, Till, et al. 2000. The implementation and performance of compressed databases. *SIGMOD Record* 29(3):55–67.

Wong, S. K. M., C. J. Butz, and Y. Xiang. 1995. A Method for Implementing a Probabilistic Model As a Relational Database. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, 556–564. UAI'95, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Yan, Weipeng P., and Per-Åke Larson. 1995. Eager Aggregation and Lazy Aggregation. In *Proceedings of the 21th International Conference on Very Large Data Bases*, 345–357. VLDB '95, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Yu, Lei, and Huan Liu. 2004. Efficient Feature Selection via Analysis of Relevance and Redundancy. *Journal of Machine Learning Research (JMLR)* 5:1205–1224.

Zhang, Ce, Arun Kumar, and Christopher Ré. 2014. Materialization Optimizations for Feature Selection Workloads. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 265–276. SIGMOD '14, New York, NY, USA: ACM.

Zhang, Yi, Weiping Zhang, and Jun Yang. 2010. I/O-efficient Statistical Computing with RIOT. In *IEEE 26th International Conference on Data Engineering (ICDE)*, 1157–1160.

A Appendix: Orion

A.1 Proofs

Proposition A.1.1. *The output* $(\nabla F, F)$ *of FL is identical to the output* $(\nabla F, F)$ *of both Materialize and Stream.*

Proof. 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 $\nabla F = \sum_{t \in \mathbf{T}} G(t.y, \mathbf{w}^\top t.\mathbf{x}) t.\mathbf{x}$ and $F = \sum_{t \in \mathbf{T}} F_e(t.y, \mathbf{w}^\top t.\mathbf{x})$. 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 3.3), we have:

$$\mathbf{HR} = \{(rid, pip)_i\}_{i=1}^{n_R}, \text{ s.t. } \forall h \in \mathbf{HR}, \exists r \in \mathbf{R} \text{ s.t. } h.rid = r.rid \wedge h.pip = \mathbf{w}_R^\top r.\mathbf{x}_R$$

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 $F = \sum_{u \in \mathbf{U}} F_e(u.y, (\mathbf{w}_S^\top u.\mathbf{x}_S + u.pip))$. But $u.pip = HR(u).pip = \mathbf{w}_R^\top R(HR(u)).\mathbf{x}_R$ and $\mathbf{w}_S^\top u.\mathbf{x}_S = \mathbf{w}_S^\top S(u).\mathbf{x}_S$. 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 $F_e(u.y, (\mathbf{w}_S^\top u.\mathbf{x}_S + u.pip)) = F_e(t.y, \mathbf{w}^\top t.\mathbf{x})$. That along with the relationship $\mathbf{w}^\top t.\mathbf{x} = \mathbf{w}_S^\top S(t).\mathbf{x}_S + \mathbf{w}_R^\top R(t).\mathbf{x}_R$ 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 $\nabla F_S = \sum_{u \in \mathbf{U}} G(u.y, (\mathbf{w}_S^\top u.\mathbf{x}_S + u.pip))\mathbf{x}_S$. 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 $G(u.y, (\mathbf{w}_S^\top u.\mathbf{x}_S + u.pip)) = G(t.y, \mathbf{w}^\top t.\mathbf{x})$, 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:

$$\forall h \in \mathbf{HS}, h.\mathrm{fip} = \sum_{u \in \mathbf{U}: u.\mathrm{rid} = h.\mathrm{rid}} G(u.y, (\mathbf{w}_S^T u.\mathbf{x}_S + u.\mathrm{pip}))$$

We then have another relation: $\mathbf{V} \leftarrow \pi(\mathbf{HS} \bowtie_{\mathbf{HS}.\mathrm{rid}=\mathbf{R}.\mathrm{rid}} \mathbf{R})$ as $\mathbf{V} = \{(\mathrm{rid}, \mathrm{fip}, \mathbf{x}_R)_i\}_{i=1}^{n_R}$. Now, FL simply aggregates $\mathbf{V}$ to compute $\nabla F_R = \sum_{v \in \mathbf{V}} (v.\mathrm{fip}) v.\mathbf{x}_R$. Due to their join expressions, we also have that $\forall v \in \mathbf{V}$, there is exactly one $r \in \mathbf{R}$ s.t. $v.\mathrm{rid} = r.\mathrm{rid}$. Since rid imposes a partition on $\mathbf{T}$, we define the partition corresponding to v as $\mathbf{T}_v = \{t \in T | t.\mathrm{rid} = v.\mathrm{rid}\}$. Thus, $v.\mathrm{fip} = \sum_{t \in \mathbf{T}_v} G(t.y, \mathbf{w}^T t.\mathbf{x})$, which coupled with the distributivity of product over a sum, implies $\nabla F_R = \nabla F_R^*$. □

Proposition A.1.2. *The output* $(\nabla F, F)$ *of FLP, FLSQL, and FLSQL+ are all identical to the output* $(\nabla F, F)$ *of FL.*

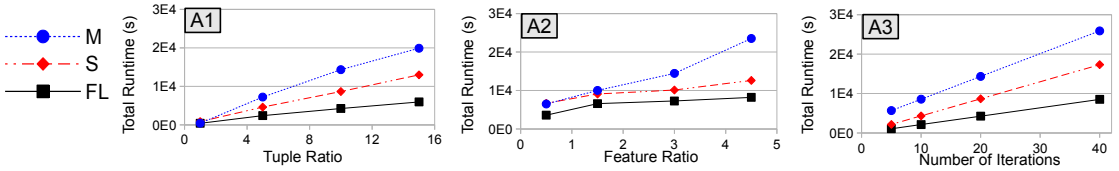
Proof. The proof for FLSQL is identical to FL, since FLSQL simply materializes some intermediate relations, while the proofs for FLSQL+ and FLP are along the same lines. For FLP, we also use the fact that addition is associative over $\mathbb{R}$ and $\mathbb{R}^d$, and both F and ∇F are just sums of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

Problem: FL-MULTJOIN 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:

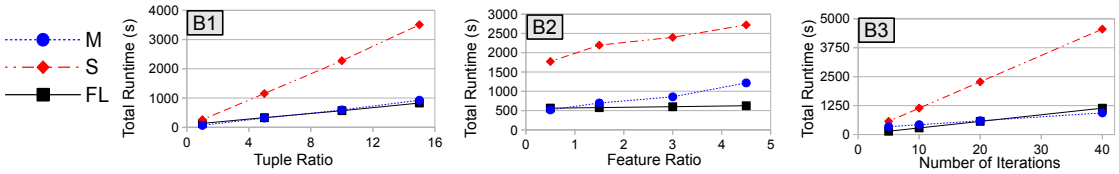
$$\max \sum_{i=1}^{k} x_i |\mathbf{R}_i|, \text{ s.t. } \sum_{i=1}^{k} x_i (|\mathbf{HR}_i| - 1) \leqslant m - 1 - k$$

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

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. $\sum_{i=1}^{n} z_i w_i \leqslant W$. 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 $|\{i | W \geqslant w_i > 0\}|$, which after reduction becomes $|\{i | m - k \geqslant |\mathbf{HR}_i| > 1\}| = l$. Thus, FL-MULTJOIN is NP-Hard in l. □

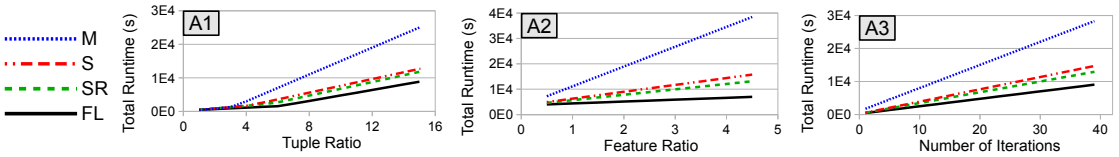


(A) RMM: Total runtimes.

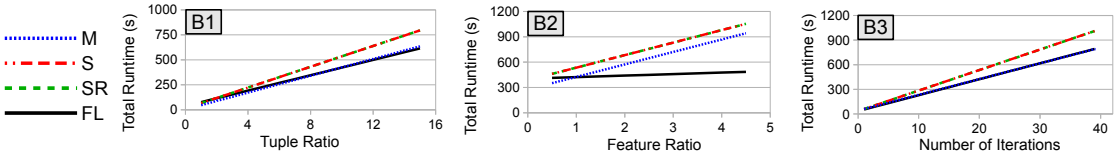


(B) RLM: Total runtimes.

Figure A.1: 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 3.3.



(A) RMM: Total runtimes.



(B) RLM: Total runtimes.

Figure A.2: 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 3.3.

A.2 Additional Runtime Plots

The implementation-based runtime results for RMM and RLM are presented in Figure A.1. The analytical cost model-based plots for the same are in Figure A.2. Note that the parameters for both these figures are fixed as per Table 3.3.

Case $|\mathbf{S}| < |\mathbf{R}|$ or Tuple Ratio $\leqslant 1$

In this case, an RDBMS optimizer would probably simply choose to build the hash table on **S** instead of **R**. It is straightforward to extend the models of M, S, and, SR to this case.

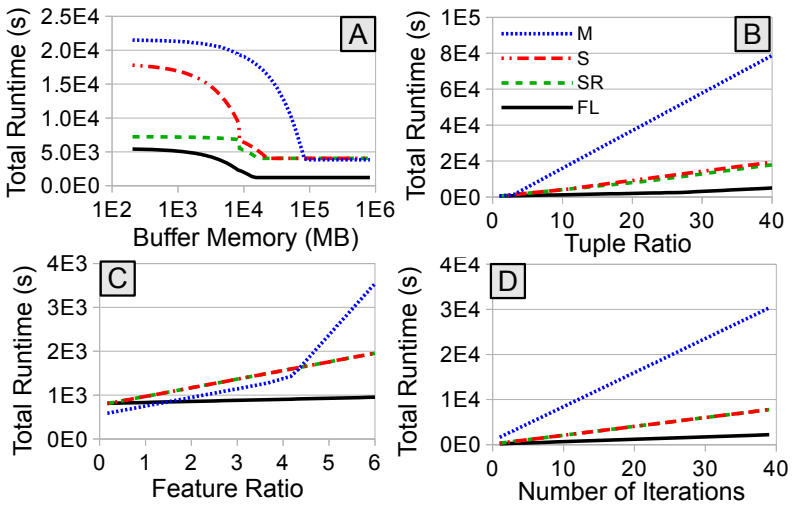


Figure A.3: Analytical plots for the case when $|\mathbf{S}| < |\mathbf{R}|$ but $n_S > n_R$. We plot the runtime against each of m, n_S, d_R, Iters, and n_R, while fixing the others. Wherever they are fixed, we set $(m, n_S, n_R, d_S, d_R, \text{Iters}) = (24\text{GB}, 1\text{E}8, 1\text{E}7, 6, 100, 20)$.

FL, however, is more interesting. Prima facie, it appears that we can switch the access to the tables: construct **H** using **S**, and join that with **R**, etc. This way, we need only one scan of **R** and two of **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 **R**, while the second is on SID with **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 **H** using **R** first as before. This is because the other approaches still perform redundant computations, and FL might still be faster. Figure A.3 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 version in some cases.

Second, when $n_S \leqslant n_R$ (irrespective of the sizes of **R** and **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 **R**, viz., with tuples that do not have a joining tuple in **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 A.4 presents the runtimes for such a setting for varying values of buffer memory as well as the other parameters. The figures confirm our observations

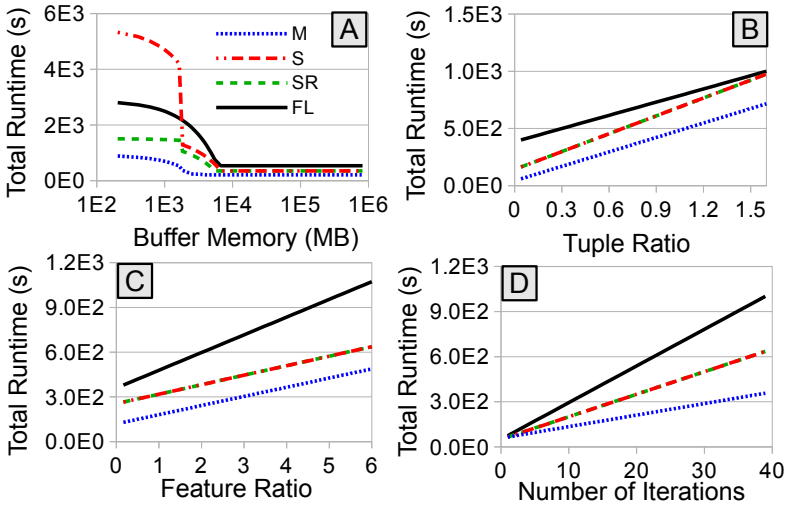


Figure A.4: Analytical plots for the case when $n_S \leqslant n_R$ (mostly). We plot the runtime against each of m, n_S, d_R, Iters, and n_R, while fixing the others. Wherever they are fixed, we set $(m, n_S, n_R, d_S, d_R, \text{Iters}) = (24\text{GB}, 2\text{E}7, 5\text{E}7, 6, 9, 20)$.

above. Of course, in the above, we have implicitly assumed that at most 1 tuple in **S** joins with a tuple in **R**. If we have multiple tuples in **S** joining with **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.3 More Cost Models and Approaches

Costs of Stream

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

```
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
]
```

Costs of Stream-Reuse

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

```
  (|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 functions
  + nS.d.(mult+add)       //Scale and add
  + nS.add                //Add for total loss
]
```

More Complex Approaches

These approaches are more complex to implement, since they might require changes to the implementation of the join operation.

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

```
  I/O Cost of Stream-Reuse
+ (Iters - 1).min{|R|+|S|,[(m-2) - f|R_0|]}     //Cache
- ⌊ Iters/2 ⌋.[|R_i| + min{|R|+|S|,(m-2)-f|R_i|}]   //H(R_B)
- ⌊ (Iters-1)/2 ⌋.[|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
- ⌊ Iters/2 ⌋.[                    //Cache HASH(R_B)
  nR.(1−q)/B.[hash + (1+dR).copy]
]
- ⌊ Iters−1/2 ⌋.[                  //Cache HASH(R_0)
  nR.q.[hash + (1+dR).copy]
]
```

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, $|\mathbf{R}'| = \lceil \frac{8n_R.(1+d_R+2)}{p} \rceil$.
2. Let $B = \lceil \frac{f|\mathbf{R}'|-(m-1)}{(m-1)-1} \rceil$. Let $|\mathbf{R}'_0| = \lfloor \frac{(m-2)-B}{f} \rfloor$, $|\mathbf{R}'_i| = \lceil \frac{|\mathbf{R}'|-|\mathbf{R}'_0|}{B} \rceil (1 \leqslant i \leqslant B)$, and $q = \frac{|\mathbf{R}'_0|}{|\mathbf{R}'|}$.
3. Similar to hybrid hash join on $\mathbf{R}'$ and $\mathbf{S}$.
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 derivable from the other approaches.

A.4 Comparing Gradient Methods

While our focus in this work 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 A.5 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 [Nocedal and Wright, 2006; Agarwal et al., 2014] but

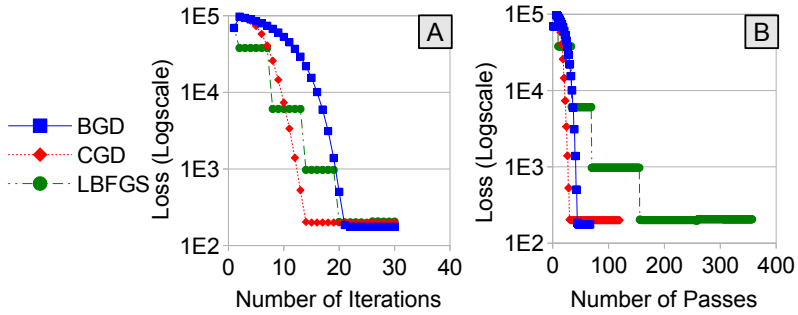


Figure A.5: Comparing gradient methods: Batch Gradient Descent (BGD), Conjugate Gradient (CGD), and Limited Memory BFGS (LBFGS) with 5 gradients saved. The parameters are $n_S = 1E5$, $n_R = 1E4$, $d_S = 40$, and $d_R = 60$. (A) Loss after each iteration. (B) Loss after each pass over the data; extra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

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 [Nocedal and Wright, 2006].

B Appendix: HAMLET

B.1 Proofs

Proposition B.1.1. *In $\boldsymbol{T}$, all $\mathsf{F} \in \boldsymbol{X}_\mathsf{R}$ are redundant.*

Proof. Consider an arbitrary feature $\mathsf{F} \in \mathbf{X}_\mathsf{R}$. We start by showing that F is weakly relevant. To show this, we need to prove that $\mathsf{P}(\mathsf{Y}|\mathbf{X}) = \mathsf{P}(\mathsf{Y}|\mathbf{X} - \{\mathsf{F}\})$ and $\exists \mathbf{Z} \subseteq \mathbf{X} - \{\mathsf{F}\}$ s.t. $\mathsf{P}(\mathsf{Y}|\mathbf{Z}, \mathsf{F}) \neq \mathsf{P}(\mathsf{Y}|\mathbf{Z})$. For the first part, we observe that due to the FD $\mathsf{FK} \rightarrow \mathbf{X}_\mathsf{R}$, fixing FK automatically fixes $\mathbf{X}_\mathsf{R}$ (and hence F). Thus, fixing $\mathbf{X}$ automatically fixes $\mathbf{X} - \{\mathsf{F}\}$ and vice versa, which implies that $\mathsf{P}(\mathsf{Y}|\mathbf{X}) = \mathsf{P}(\mathsf{Y}|\mathbf{X} - \{\mathsf{F}\})$. As for the second part, we only need to produce one instance. Choose $\mathbf{Z} = \phi$, which means we need to show that it is possible to have $\mathsf{P}(\mathsf{Y}|\mathsf{F}) \neq \mathsf{P}(\mathsf{Y})$. It is clearly trivial to produce an instance satisfying this inequality. Thus, F is weakly relevant. Next, we show that F has a Markov Blanket $\mathbf{M}_\mathsf{F} \subseteq \mathbf{X} - \{\mathsf{F}\}$. In fact, we have $\mathbf{M}_\mathsf{F} = \{\mathsf{FK}\}$. This is because the FD $\mathsf{FK} \rightarrow \mathbf{X}_\mathsf{R}$ causes F to be fixed when FK is fixed, which implies $\mathbf{M}_\mathsf{F} \cup \{\mathsf{F}\}$ is fixed when $\mathbf{M}_\mathsf{F}$ is fixed and vice versa. Hence, $\mathsf{P}(\mathsf{Y}, \mathbf{X} - \{\mathsf{F}\} - \mathbf{M}_\mathsf{F} | \mathbf{M}_\mathsf{F}, \mathsf{F}) = \mathsf{P}(\mathsf{Y}, \mathbf{X} - \{\mathsf{F}\} - \mathbf{M}_\mathsf{F} | \mathbf{M}_\mathsf{F})$. Thus, overall, F is a redundant feature. Since F was arbitrary, all features in $\mathbf{X}_\mathsf{R}$ are redundant. □

Corollary B.1. *Given a table $\boldsymbol{T}(\underline{\mathsf{ID}}, \mathsf{Y}, \boldsymbol{X})$ with a canonical acyclic set of FDs $\mathcal{Q}$ on the features $\boldsymbol{X}$, a feature that appears in the dependent set of an FD in $\mathcal{Q}$ is redundant.*

Proof. The proof is a direct extension of the proof for Proposition B.1.1, and we only present the line of reasoning here. We convert $\mathbf{T}$ into a relational schema in Boyce-Codd Normal Form (BCNF) using standard techniques that take $\mathcal{Q}$ as an input [Silberschatz et al., 2006]. Since ID is the primary key of $\mathbf{T}$, both ID and Y will be present in the same table after the normalization (call it the "main table"). Now, features that occur on the right-hand side of an FD will occur in a separate table whose key will be the features on the left-hand side of that FD. And there will be a KFKD between a feature (or features) in the main table and the keys of the other tables in a manner similar to how FK refers to RID. Thus all features that occur on the right-hand side of an FD provide no more information than the features in the main table in the same way that $\mathbf{X}_\mathsf{R}$ provides no more information than FK in Proposition B.1.1. Hence, any feature that occurs in the dependent set of an FD in $\mathcal{Q}$ is redundant. □

Theorem B.2. $\forall \mathsf{F} \in \boldsymbol{X}_\mathsf{R}, \mathsf{I}(\mathsf{F}; \mathsf{Y}) \leqslant \mathsf{I}(\mathsf{FK}; \mathsf{Y})$

Proof. Let the FD $FK \to \mathbf{X}_R$ be represented by a collection of functions of the form $f_F : \mathcal{D}_{FK} \to \mathcal{D}_F$ for each $F \in \mathbf{X}_R$. Our goal is to show that $I(FK;Y) \geqslant I(F;Y), \forall F \in \mathbf{X}_R$.

Consider any $F \in \mathbf{X}_R$. We have the following:

$$I(F;Y) = \sum_{x,y} P(F = x, Y = y) \log \frac{P(F = x, Y = y)}{P(F = x)P(Y = y)}$$
$$I(FK;Y) = \sum_{z,y} P(FK = z, Y = y) \log \frac{P(FK = z, Y = y)}{P(FK = z)P(Y = y)}$$

Due to the FD $FK \to \mathbf{X}_R$, the following equalities hold:

$$P(F = x) = \sum_{z : f_F(z) = x} P(FK = z)$$
$$P(F = x, Y = y) = \sum_{z : f_F(z) = x} P(FK = z, Y = y)$$

Since all the quantities involved are non-negative, we can apply the *log-sum inequality*, which is stated as follows.

Definition B.3. *Given non-negative numbers* $a_1, \ldots, a_n$ *and* $b_1, \ldots, b_n$, *with* $a = \sum a_i$ *and* $b = \sum b_i$, *the following inequality holds, and it is known as the* log-sum inequality*:*

$$\sum_{i=1}^{n} a_i \log(\frac{a_i}{b_i}) \geqslant a \log(\frac{a}{b})$$

In our setting, fixing (x, y), we have $a = P(F = x, Y = y)$ and a_is are $P(FK = z, Y = y)$, for each $z : f_F(z) = x$. Similarly, $b = P(F = x)P(Y = y)$ and b_is are $P(FK = z)P(Y = y)$, for each $z : f_F(z) = x$. Thus, we have the following inequality:

$$\sum_{z : f_F(z) = x} P(FK = z, Y = y) \log \frac{P(FK = z, Y = y)}{P(FK = z)P(Y = y)} \geqslant$$
$$P(F = x, Y = y) \log \frac{P(F = x, Y = y)}{P(F = x)P(Y = y)}$$

Since this is true for all values of (x, y), summing all the inequalities gives us $I(FK;Y) \geqslant I(F;Y)$. □

Proposition B.1.2. *It is possible for a feature* $F \in \boldsymbol{X}_R$ *to have higher* $IGR(F;Y)$ *than* $IGR(FK;Y)$.

Proof. It is trivial to construct such an instance. Thus, we omit the proof here. □

Proposition B.1.3. $\mathcal{H}_{\mathbf{X}} = \mathcal{H}_{\mathsf{FK}} \supseteq \mathcal{H}_{\mathbf{X}_R}$

Proof. Recall that we had assumed $\mathbf{X}_S = \phi$. Thus, $\mathbf{X} \equiv \{\mathsf{FK}\} \cup \mathbf{X}_R$. We first prove the first part. By definition, given $\mathbf{Z} \subseteq \mathbf{X}$, we have: $\mathcal{H}_{\mathbf{Z}} = \{f | f \in \mathcal{H}_{\mathbf{X}} \wedge \forall \mathbf{u}, \mathbf{v} \in \mathcal{D}_{\mathbf{X}}, \mathbf{u}_{|\mathbf{Z}} = \mathbf{v}_{|\mathbf{Z}} \implies f(\mathbf{u}) = f(\mathbf{v})\}$. It is easy to see that the FD $\mathsf{FK} \rightarrow \mathbf{X}_R$ automatically ensures that this condition is true for $\mathbf{Z} = \{\mathsf{FK}\}$. This is because $\forall \mathbf{u}, \mathbf{v} \in \mathcal{D}_{\mathbf{X}}$ s.t. $\mathbf{u}_{|\mathsf{FK}} = \mathbf{v}_{|\mathsf{FK}}$, the FD implies $\mathbf{u}_{|\mathbf{X}_R} = \mathbf{v}_{|\mathbf{X}_R}$, which in turn implies $\mathbf{u} = \mathbf{v}$, and hence, $f(\mathbf{u}) = f(\mathbf{v})$. Thus, $\mathcal{H}_{\mathbf{X}} = \mathcal{H}_{\mathsf{FK}}$.

As for the second part, we show that for any arbitrary $f \in \mathcal{H}_{\mathbf{X}_R}$, $\exists g_f \in \mathcal{H}_{\mathsf{FK}}$ s.t. $f = g_f$. Note that an $f \in \mathcal{H}_{\mathbf{X}_R}$ satisfies the condition $\forall \mathbf{u}, \mathbf{v} \in \mathcal{D}_{\mathbf{X}}, \mathbf{u}_{|\mathbf{X}_R} = \mathbf{v}_{|\mathbf{X}_R} \implies f(\mathbf{u}) = f(\mathbf{v})$. Similarly, any $g \in \mathcal{H}_{\mathsf{FK}}$ satisfies the condition $\forall \mathbf{u}, \mathbf{v} \in \mathcal{D}_{\mathbf{X}}, \mathbf{u}_{|\mathsf{FK}} = \mathbf{v}_{|\mathsf{FK}} \implies g(\mathbf{u}) = g(\mathbf{v})$. We now pick some $g_f \in \mathcal{H}_{\mathsf{FK}}$ that also satisfies the following condition: $\forall \mathbf{u} \in \mathcal{D}_{\mathbf{X}}$, $g_f(\mathbf{u}) = f(\mathbf{u})$. Such a g_f necessarily exists because of three reasons: $\mathcal{H}_{\mathbf{X}_R}$ is defined based only on those values of $\mathbf{X}_R$ that are actually present in $\mathbf{R}$, the FD $\mathsf{FK} \rightarrow \mathbf{X}_R$ ensures that there is at least one value of FK that maps to a given value of $\mathbf{X}_R$, and the same FD also ensures (by definition) that $\forall \mathbf{u}, \mathbf{v} \in \mathcal{D}_{\mathbf{X}}, \mathbf{u}_{|\mathsf{FK}} = \mathbf{v}_{|\mathsf{FK}} \implies \mathbf{u}_{|\mathbf{X}_R} = \mathbf{v}_{|\mathbf{X}_R}$. Thus, overall, $\mathcal{H}_{\mathbf{X}_R} \subseteq \mathcal{H}_{\mathsf{FK}}$. Note that the equality arises when there is exactly one value of FK that maps to one value of $\mathbf{X}_R$ in $\mathbf{R}$, i.e., all tuples in $\mathbf{R}$ have distinct values of $\mathbf{X}_R$. □

B.2 More Simulation Results

Figure B.1 presents the remaining key plots for the simulation scenario in which the true distribution is succinctly captured using a lone feature $X_r \in \mathbf{X}_R$. We also studied two other scenarios: one in which all of $\mathbf{X}_R$ and $\mathbf{X}_S$ are part of the true distribution, and another in which only $\mathbf{X}_S$ and FK are. Figure B.2 presents the plots for the former. Since the latter scenario in which $\mathbf{X}_R$ is useless did not reveal any interesting new insights, we skip it for brevity. We also plot the difference in test error between *NoJoin* and *JoinAll* for the scenario in which all of $\mathbf{X}_R$ and $\mathbf{X}_S$ are part of the true distribution in Figure B.3. We see that the trends are largely similar to those in Figure 5.5 and that the same thresholds for ρ and τ work here as well.

Foreign Key Skew

So far, we had assumed that FK values are not skewed. Neither the ROR nor the TR account for skew in $P(\mathsf{FK})$. Foreign key skew is a classical problem in the database literature due to its effects on parallel joins [Silberschatz et al., 2006] but its effects on ML have not been studied before. We now shed some light on the effects of skew in $P(\mathsf{FK})$ on ML.

We start by observing a key twist to the database-style understanding of skew: skew in $P(\mathsf{FK})$ *per se* is less important than its implications for learning the target. Thus, we classify

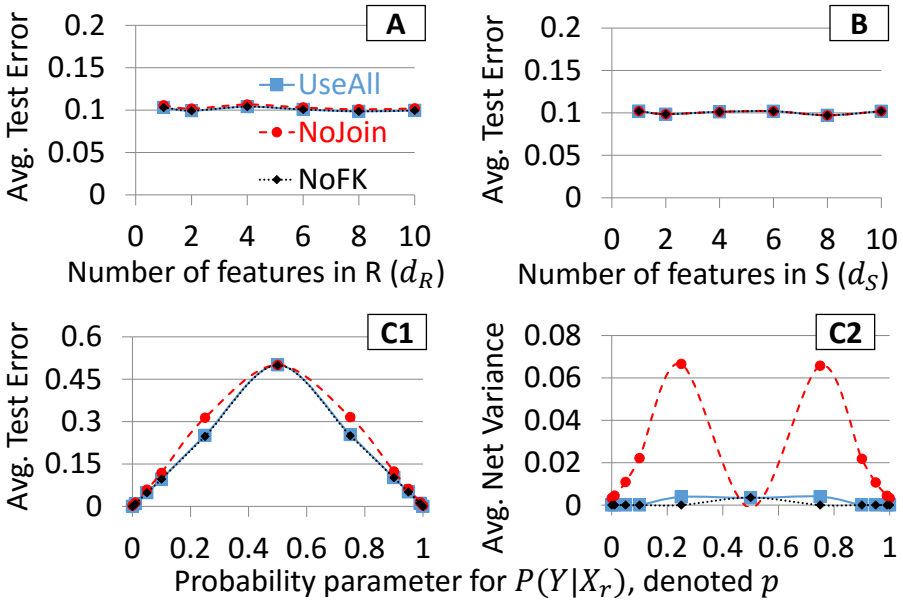


Figure B.1: Remaining simulation results for the same scenario as Figure 5.3. (A) Vary d_R, while fixing $(n_S, d_S, |\mathcal{D}_{FK}|, p) = (1000, 4, 100, 0.1)$. (B) Vary d_S, while fixing $(n_S, d_R, |\mathcal{D}_{FK}|, p) = (1000, 4, 40, 0.1)$. (C) Vary p, while fixing $(n_S, d_S, d_R, |\mathcal{D}_{FK}|) = (1000, 4, 4, 200)$.

skew in $P(FK)$ into two types: *benign* and *malign*. Loosely defined, benign skew in $P(FK)$ is that which does not make it much harder to learn the target, while malign skew is the opposite. We give some intuition using the scenario of a lone feature $X_r \in \mathbf{X}_R$ being part of the true distribution. Suppose $P(X_r)$ has no skew (the distribution is based on $\mathbf{T}$, not $\mathbf{R}$). Since multiple FK values might map to the same X_r value, there might still be high skew in $P(FK)$. But what really matters for accuracy is whether $P(Y)$ is skewed as well, and whether the skew in $P(Y)$ "colludes" with the skew in $P(FK)$.

There are three possible cases when there is skew in $P(FK)$: (1) $P(Y)$ is not skewed, (2) $P(Y)$ is skewed (some class label is dominant), and $P(FK)$ is skewed such that low-probability FK values co-occur mostly with high-probability Y values, and (3) $P(Y)$ is skewed, and $P(FK)$ is skewed such that low-probability FK values co-occur mostly with low-probability Y values. Cases (1) and (2) represent benign skew: even though some FK values have low probability, together they might still be able to learn the target concept reasonably well because there might be "enough" training examples for each class label. But case (3) is an instance of malign skew: essentially, FK "diffuses" the already low probability of some Y value(s) into a potentially large number of low-probability FK values. This issue might be less likely if X_r was used instead, i.e., the join was not avoided, since

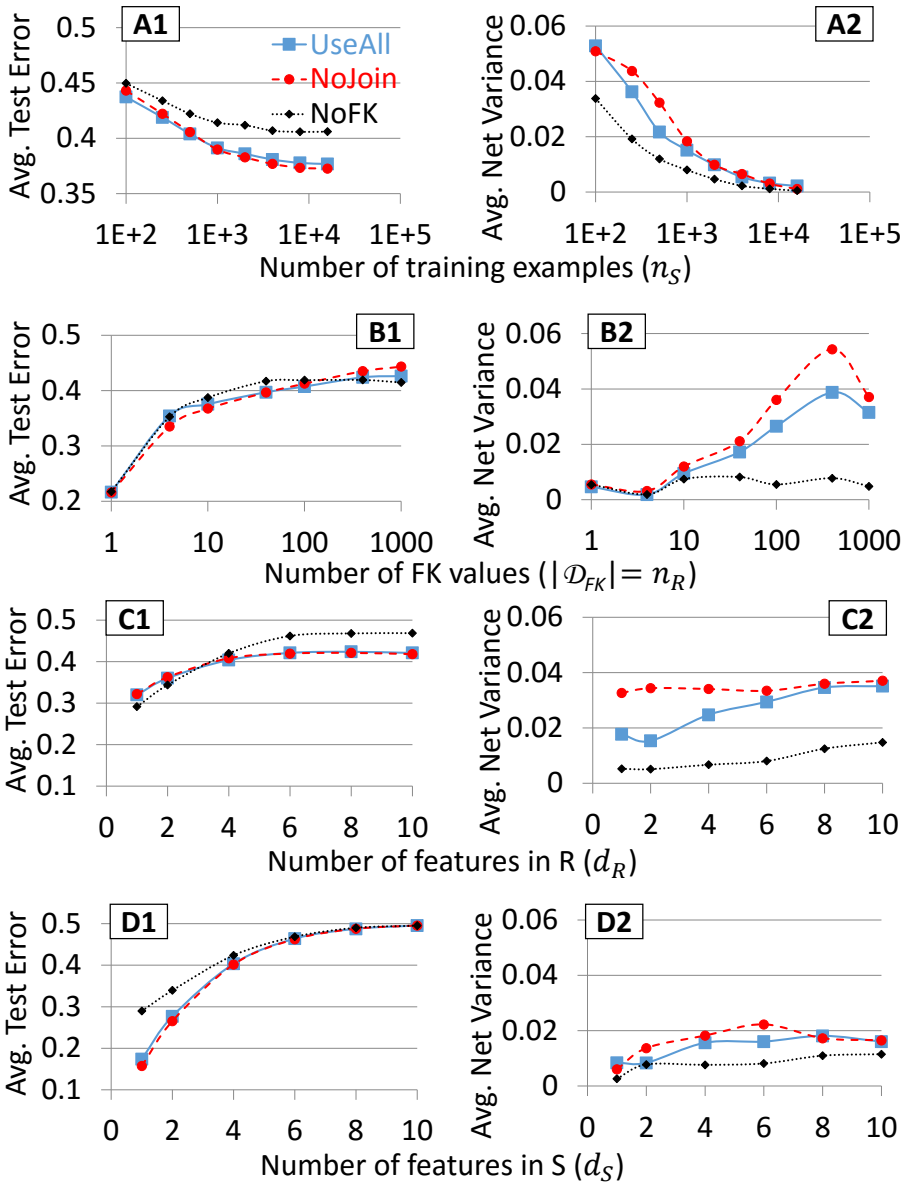


Figure B.2: Simulation results for the scenario in which all of $\mathbf{X}_S$ and $\mathbf{X}_R$ are part of the true distribution. (A) Vary n_S, while fixing $(d_S, d_R, |\mathcal{D}_{FK}|) = (4, 4, 40)$. (B) Vary $|\mathcal{D}_{FK}|$, while fixing $(n_S, d_S, d_R) = (1000, 4, 4)$. (C) Vary d_R, while fixing $(n_S, d_S, |\mathcal{D}_{FK}|) = (1000, 4, 100)$. (D) Vary d_S, while fixing $(n_S, d_R, |\mathcal{D}_{FK}|) = (1000, 4, 40)$.

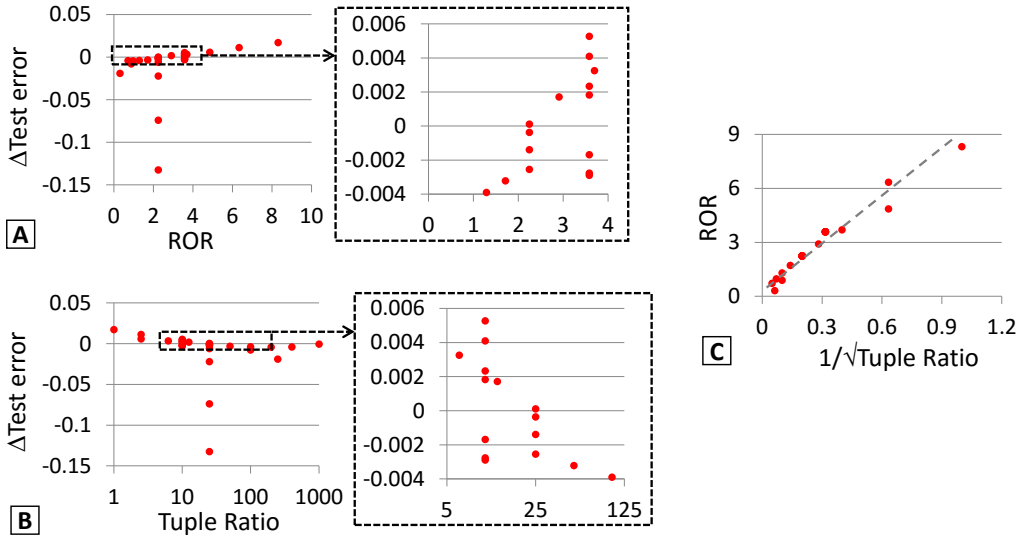


Figure B.3: Scatter plots based on all the results of the simulation experiments referred to by Figure B.2. (A) Increase in test error caused by avoiding the join (denoted "ΔTest error") against ROR. (B) ΔTest error against tuple ratio. (C) ROR against inverse square root of tuple ratio.

usually $|\mathcal{D}_{X_r}| \ll |\mathcal{D}_{FK}|$. Thus, malign skews in $P(FK)$ might make it less safe to avoid the join.

To verify the above, we performed two more simulation experiments. First, we embed a benign skew in FK using the standard Zipf distribution, which is often used in the database literature [Pavlo et al., 2012]. Second, we embed a malign skew in FK using what we call a "needle-and-thread" distribution: one FK value has a probability mass p ("needle" probability) and it is associated with one X_r value (and hence, one Y value). The remaining $1-p$ probability mass is uniformly distributed over the remaining $(n_R - 1)$ FK values, all of which are associated with the other X_r value (and hence, the other Y value). Intuitively, this captures the extreme case (3) in which the skew in FK colludes with the skew in Y. Figure B.5 presents the results for *UseAll* and *NoJoin*. As expected, the benign skew does not cause the test error of *NoJoin* to increase much, but the malign skew does. Figure B.5(A) also suggests that benign skew might sometimes work in favor of *NoJoin* (this is primarily because the bias increased for *UseAll*). But as Figure B.5(B1) shows, the test error of *NoJoin* increases when the skew in Y colludes with the skew in FK. However, as Figure B.5(B2) shows, this gap closes as the number of training examples increases.

Overall, we need to account for malign skews in FK when using the ROR or TR rules for avoiding joins safely. While it is possible to detect malign skews using $H(FK|Y)$, we take a simpler, albeit more conservative, approach. We just check $H(Y)$, and if it is too low

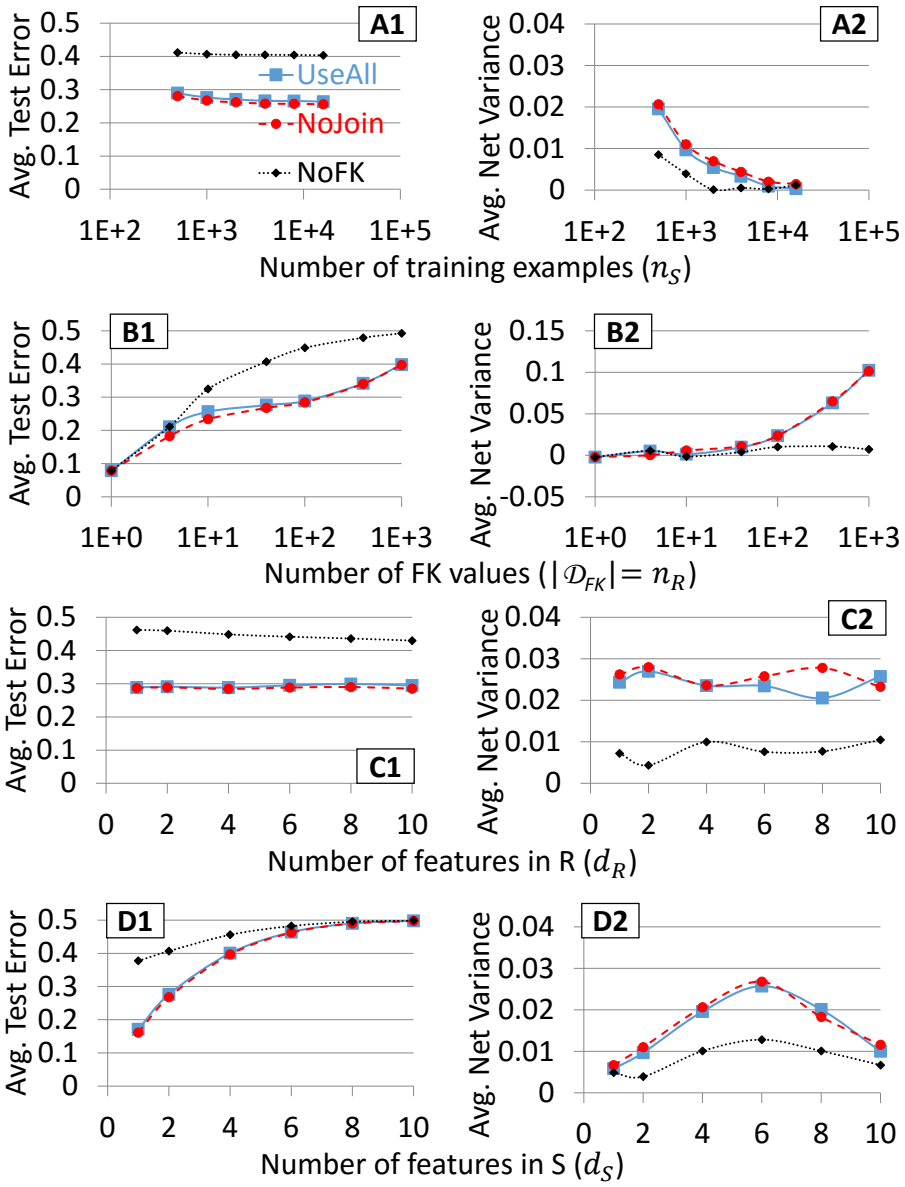


Figure B.4: Simulation results for the scenario in which only $\mathbf{X}_S$ and FK are part of the true distribution. (A) Vary n_S, while fixing $(d_S, d_R, |\mathcal{D}_{FK}|) = (2, 4, 40)$. (B) Vary $|\mathcal{D}_{FK}|$, while fixing $(n_S, d_S, d_R) = (1000, 2, 4)$. (C) Vary d_R, while fixing $(n_S, d_S, |\mathcal{D}_{FK}|) = (1000, 2, 100)$. (D) Vary d_S, while fixing $(n_S, d_R, |\mathcal{D}_{FK}|) = (1000, 4, 40)$.

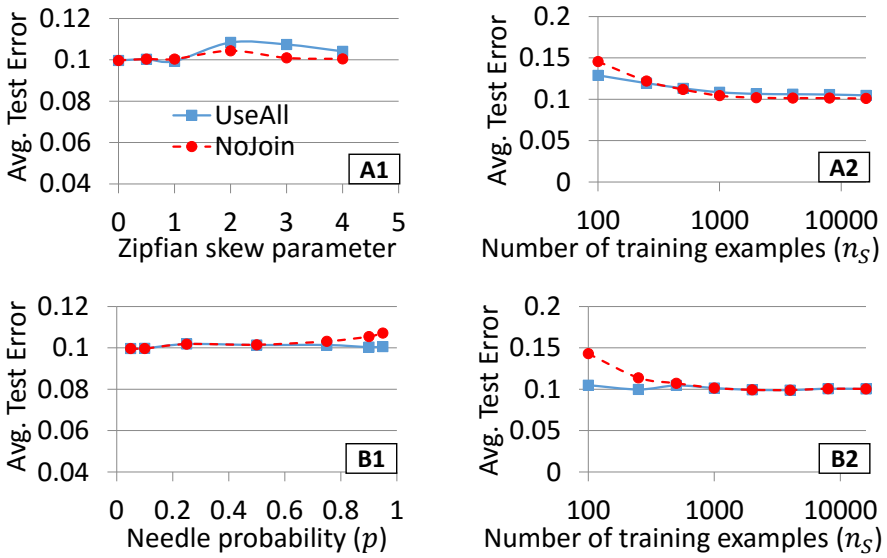


Figure B.5: Effects of foreign key skew for the scenario referred to by Figure 5.3. (A) Benign skew: P(FK) has a Zipfian distribution. We fix $(n_S, n_R, d_S, d_R) = (1000, 40, 4, 4)$, while for (A2), the Zipf skew parameter is set to 2. (B) Malign skew: P(FK) has a needle-and-thread distribution. We fix $(n_S, n_R, d_S, d_R) = (1000, 40, 4, 4)$, while for (A2), the needle probability parameter is set to 0.5.

(say, below 0.5, which corresponds roughly to a 90%:10% split), we do not avoid the join. This is in line with our guiding principle of conservatism. It also captured all the cases of malign skews in our above simulations. We leave more complex approaches for handling skew to future work.

B.3 Output Feature Sets

For each dataset and feature selection method combination, we provide the output feature sets of both *JoinAll* and *JoinOpt*. We omit Yelp and BookCrossing, since none of the joins were avoided by *JoinOpt* on those two datasets.

Walmart:

Forward Selection:

JoinAll = JoinOpt = {Dept, StoreID, IndicatorID}

Backward Selection:

JoinAll = {Dept, StoreID, Type, Size, FuelPriceStdev, TempStdev, FuelPriceAvg, CPIStdev}

JoinOpt = {Dept, StoreID, IndicatorID}

MI-Based Filter:

JoinAll = JoinOpt = {Dept, StoreID, IndicatorID}

IGR-Based Filter:

JoinAll = {Dept, StoreID, Type, Size}
JoinOpt = {Dept, StoreID, IndicatorID}

Expedia:

Forward Selection:

JoinAll = JoinOpt = {HotelID, BookingWindow, SatNightBool, Year, RandomBool, ChildrenCount, Score2}

Backward Selection:

JoinAll = JoinOpt = {HotelID, BookingWindow, Time, SatNightBool, RandomBool, ChildrenCount, Score2, AdultsCount, LengthOfStay, VisitorCountry, RoomCount, SiteID}

MI-Based Filter:

JoinAll = JoinOpt = {HotelID, Score2}

IGR-Based Filter:

JoinAll = {HotelID, Score2, PromoFlag, BookingCount}
JoinOpt = {HotelID, Score2, PromoFlag}

Flights:

Forward Selection:

JoinAll = JoinOpt = {AirlineID, Eq5, Eq4, Eq10, Eq20, Eq1, Eq17, Eq16, Eq6, Eq7, Eq13, Eq9, Eq11}

Backward Selection:

JoinAll = {AirlineID, Eq5, Eq4, Eq10, Eq20, Eq1, Eq17, Eq16, Eq6, Eq7, Eq13, Eq9, Eq11, Eq2, Eq3, Name1, Active, Eq12, Eq15, Eq19}
JoinOpt = {AirlineID, Eq5, Eq4, Eq10, Eq20, Eq1, Eq17, Eq16, Eq6, Eq7, Eq9, Eq11, Eq2, Eq17, Eq14, Eq15, Eq18, Eq19}

MI-Based Filter:

JoinAll = JoinOpt = {AirlineID, DestAirportID}

IGR-Based Filter:

JoinAll = {AirlineID, Active, Eq12, Eq12, Eq15, Eq7, Eq8, Eq9, Eq6, Eq2, Eq1, Eq3, Eq11}
JoinOpt = {AirlineID, Eq12, Eq12, Eq15, Eq7, Eq8, Eq9, Eq6, Eq2, Eq1, Eq3, Eq11, Eq19}

MovieLens1M:

Forward Selection:

JoinAll = {UserID, MovieID, Genre18}
JoinOpt = {UserID, MovieID}

Backward Selection:

JoinAll = {UserID, MovieID, Genre18, Gender, Genre3, Genre4, Genre16}
JoinOpt = {UserID, MovieID}

MI-Based Filter:

JoinAll = JoinOpt = {UserID, MovieID}

IGR-Based Filter:

JoinAll = {UserID, MovieID, Genre10}
JoinOpt = {UserID, MovieID}

LastFM:

Forward Selection:

JoinAll = JoinOpt = {UserID}

Backward Selection:

JoinAll = {UserID, Gender, Genre2, Genre3, Genre4, Genre5}
JoinOpt = {UserID}

MI-Based Filter:

JoinAll = JoinOpt = {UserID}

IGR-Based Filter:

JoinAll = JoinOpt = {UserID}