RICE UNIVERSITY

Computer Systems for Distributed Machine Learning

by

Zekai J. Gao

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE
Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Christopher Jermaine, Chair
Professor of Computer Science

Luay Nakhleh
Professor of Computer Science

Marina Vannucci
Professor of Statistics

Swarat Chaudhuri
Associate Professor of Computer Science

Houston, Texas
January, 2018
ABSTRACT

Computer Systems for Distributed Machine Learning

by

Zekai J. Gao

My thesis considers the design and development of state-of-the-art distributed data analytics systems supporting the implementation and execution of machine learning algorithms. Specifically, I consider how to support iterative, large-scale machine algorithms problems on a prototype distributed relational database system called SimSQL. SimSQL allows a programmer to leverage the power of declarative programming and data independence to specify what a computation does, and not how to implement it. This increases programmer productivity and means that the same implementation can be used for different data sets of different sizes and complexities, and different hardwares.

The thesis considers three specific problems in the context of adapting SimSQL for the implementation and execution of large-scale machine learning algorithms. First, during learning, when a user-defined function is parameterized with a data object and a statistical model used to process that object, the fully parameterized model can be huge. How do we deal with the potential massive blowup in size during distributed learning? Second, although the idea of data independence—a fundamental design principle upon which relational database systems are built—supports the notion of “one implementation, any model/data size and compute hardware”, such systems lack sufficient support for recursive computations in deep learning and other applications.
How should such a system be modified to support these applications? Third, some key features of distributed platforms aim at more general applications in data processing and are not always the best fit for large-scale machine learning and distributed linear algebra. Can we achieve higher efficiency on these platforms by avoiding some widely existing pitfalls?

My thesis addresses the issues above by first describing and studying the ubiquitous join-and-co-group pattern for user-defined function parameterization, and carefully describing the alternatives for implementing this pattern on top of both SimSQL and Apache Spark. Second, I enhance SimSQL to support declarative recursion via multidimensional tables, then modify the query optimization framework so that it can handle the massive query plans that result from complicated recursive computations. I benchmark the resulting system, comparing it with TensorFlow and Spark. Third, I examine various performance bottlenecks associated with SimSQL in running large-scale machine learning applications, and consider three enhancements in large vector-type or matrix-type data partitioning, choice of physical plans for complicated operations as well as runtime compilation respectively.
Acknowledgements

In the process of completing this thesis, I’ve received help, support and inspiration from many individuals and groups. I would like to extend my sincere thanks to each one of them for making this happen.

First and foremost, I would like to express my deepest gratitude to my PhD advisor at Rice University, Prof. Chris Jermaine. Without his help and supervision, I would not have been able to complete this thesis or publish papers at world-class conferences. He is a true role model for me to learn from.

I am also deeply grateful to Prof. Luay Nakhleh, Prof. Swarat Chaudhuri, Prof. Marina Vannucci and Prof. Devika Subramanian for being on my committees and challenging me with their brilliant ideas.

It is also my great pleasure to thank all of my colleagues at Rice University, including Mr. Shangyu Luo, Dr. Risa Myers, Mr. Dimitrie Jankov, Dr. Jia Zou, Mr. Letao Qi, Mr. Arkabandhu Chowdhury, Mr. Rohan Mukherjee, Mr. Sourav Sikdar, Dr. Carlos Monroy, Dr. Kia Teymourian, Dr. Luis L. Perez, Dr. Zhuhua Cai, Dr. Niketan Pansare, Dr. Anna Drummond and Mr. Rahul Kumar, for sharing their wisdom selflessly with me.

I also owe deep gratitude to everyone at Rice Computer Science Department and Rice CSGSA. I thank the Houston Astros for being a true inspiration in the year of 2017.

Last but not least, my most sincere appreciation goes to my dear family for being there during times of need and my great friends for standing by me no matter what happens. This thesis would not have been possible without all of their support.
## Contents

Abstract  
List of Illustrations  
List of Tables  

1 Introduction  
1.1 Overview of My PhD Research Projects  
1.2 Declarative UDF Parameterizations  
1.3 Declarative Recursive Computation  
1.4 Overcoming Performance Bottlenecks in Distributed Machine Learning  
1.5 Thesis Organization  

2 Declarative Parameterizations of User-Defined Functions  
2.1 Motivation  
2.2 The Join-And-Co-Group Pattern  
2.2.1 Generic Form  
2.2.2 Realization in SimSQL  
2.2.3 Examples  
2.3 Implementation  
2.3.1 Naive Implementation  
2.3.2 Local Implementation  
2.3.3 Global Implementation  
2.4 Optimization  
2.5 On a Dataflow Platform  
2.5.1 Key Design Considerations
2.5.2 Naive implementation ........................................... 43
2.5.3 Local implementation ............................................ 45
2.5.4 Global implementation .......................................... 45
2.6 Experiments .......................................................... 46
  2.6.1 Machine Learning Tasks ....................................... 47
  2.6.2 Join-and-Co-Group Specifics ................................. 49
  2.6.3 Parallel Database Results ..................................... 50
  2.6.4 Spark Results .................................................... 51
  2.6.5 Discussion ....................................................... 52

3 Declarative Recursive Computation on a RDBMS ............. 58
  3.1 Motivation .......................................................... 58
  3.2 Parallelism in ML ................................................ 63
  3.3 Deep Learning on an RDBMS .................................... 65
    3.3.1 A Simple Deep Learner .................................. 65
    3.3.2 Deep Learning on an RDBMS .............................. 68
    3.3.3 So, What’s the Catch? .................................... 69
  3.4 Extensions to SQL ................................................ 72
    3.4.1 The Extensions .............................................. 72
    3.4.2 Learning Using Recursive SQL ............................ 75
  3.5 Handling Very Large Query Plans .............................. 76
  3.6 Problem Definition ............................................... 79
    3.6.1 Intuition ..................................................... 79
    3.6.2 Cost-Based Cutting ....................................... 80
  3.7 Heuristic Solution ................................................ 82
    3.7.1 Choosing Costs ............................................ 85
  3.8 Experiments ........................................................ 86
    3.8.1 Overview ..................................................... 87
Illustrations

2.1 Iterative optimization via the join-and-co-group pattern ........................................ 20
2.2 Pseudo-code for iterative optimization via the join-and-co-group pattern .......................... 21
2.3 Naive join-and-co-group for LDA. The entire topics portion of the model is broadcast to each data point. For a large model, this is unrealizable ...................................................... 24
2.4 Join-and-co-group for LDA. In this case, topics is broken into blocks (subsets of words) so that the entire model need not be broadcast to each and every document during join-and-co-group ................................................. 27
2.5 A template plan depicting the join-and-co-group pattern in a general setting ..................... 29
2.6 Total join-and-co-group time for parallel database system. Format is HH:MM:SS .............. 51
2.7 Breakdown of SimSQL execution time; linear regression. The figure shows the grouping of operations into pipelines. The implementation constructs two pipelines for naive/local. Three are constructed for global (at right) ................................................................. 52
2.8 Breakdown of SimSQL execution time; GMM. The implementation constructs three pipelines for naive/local. Four are constructed for global (at right) ................................................................. 53
2.9 Breakdown of SimSQL execution time; LDA (smaller model). The implementation constructs five pipelines. Both naive/local fail ....... 53
2.10 Breakdown of SimSQL execution time; LDA (larger model). The implementation constructs five pipelines. Both naive/local fail.

2.11 Data set sizes.

2.12 Total running time for Spark (data flow platform).

3.1 Structure of a feed-forward neural network.

3.2 SQL code to implement the backward pass for iteration $i$ of a feed-forward deep network with eight hidden layers.

3.3 Greedily cutting a frame from a compute plan.

3.4 Lines of source code for each of the implementations. Comments and blank lines are ignored. TensorFlow lines of source code for the feed-forward network and for Word2Vec are from the library implementations. "*" denotes a data-parallel implementation.

3.5 Latency of feed-forward neural networks and Word2Vec on 10 machines, using different graph cut algorithms in SimSQL. The hidden layers of feed-forward neural networks have 40K neurons each. The embedding dimensionality of Word2Vec is 100. Experiments over 2 hours are terminated. Format is HH:MM:SS.

3.6 Latency of feed-forward neural networks on a cluster of 10 machines, as the size of hidden layers increases. Each mini-batch contains 100 documents. Format is HH:MM:SS.

3.7 Runtime of feed-forward neural networks with a fixed hidden layer size of 160000, as the number of machines increases. Format is HH:MM:SS.

3.8 Latency of Word2Vec on 10 machines, as the dimension of word embeddings increases. Format is HH:MM:SS.

3.9 Average sub-step runtime of collapsed LDA on 10 machines with varied numbers of topics. Format is HH:MM:SS.
3.10 Per-hour NCE loss of Word2vec with 100-dimensional word embeddings. 96

4.1 The pre-compilation. Only two threads are needed. One handles job preparation and the other servers as the master node during the execution of the Hadoop job. . . . . . . . . . . . . . . . . . . . . . . . 101

4.2 Per-minibatch latency (in seconds) of feed-forward neural networks with 40000 and 160000 neurons in the hidden layers respectively, on a cluster of ten EC2 m2.4xlarge machines. . . . . . . . . . . . . . . . . . . . 105

4.3 Per-minibatch latency (in seconds) of Word2Vec with embedding dimensionalities of 100, 1000 and 10000 respectively, on a cluster of ten EC2 m2.4xlarge machines. . . . . . . . . . . . . . . . . . . . 105

4.4 Per-minibatch latency (in seconds) of feed-forward neural networks with 40000 and 160000 neurons in the hidden layers respectively, on a cluster of ten EC2 m2.4xlarge machines using different partitioning schemes. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108

4.5 Runtime (in seconds) of block matrix multiplication $XY$ with sizes $10k \times 10k$ and $10k \times 10k$ on a cluster of ten EC2 m2.4xlarge machines. 109

4.6 Runtime (in seconds) of block matrix multiplication $XY$ with sizes $10k \times 100k$ and $100k \times 10k$ on a cluster of ten EC2 m2.4xlarge machines.109

4.7 Per-minibatch latency (in seconds) of feed-forward neural networks with 40000 and 160000 neurons in the hidden layers respectively, on a cluster of ten EC2 m2.4xlarge machines. . . . . . . . . . . . . . . . . . . . 112

4.8 Per-iteration time (in seconds) of Gaussian mixture models with two sets of data sizes for the number of data points ($N$), the dimensionality of data ($D$) and the number of clusters ($C$) on ten EC2 m2.4xlarge machines. . . . . . . . . . . . . . . . . . . . 113
Tables
Chapter 1

Introduction

Machine learning has become difficult in the big data era due to the increase in scale of real-world machine learning applications, both in data size and in model size [1, 2, 3]. For example, in a dense feed-forward neural network [4] with two adjacent layers of 100,000 neurons each, the weight matrix connecting the layers contains $10^{10}$ model parameters to train. As another example, consider the per-topic word distribution represented in a dense matrix while learning a latent Dirichlet allocation (LDA) [5] model with non-collapsed Gibbs sampling [6, 7]. Given 10,000 topics and one million words in the vocabulary, the per-topic word distribution matrix is as large as 80GB in size, larger than the available RAM in many commodity machines.

Designing distributed systems targeted at large-scale machine learning has attracted a lot of attention. The earliest system widely used for distributed machine learning was Hadoop [8]. When Hadoop and its underlying MapReduce computing framework became successful, practitioners began building scalable machine learning solutions on top of it. Despite the fact that it was not designed for machine learning applications in the very beginning, it was still used as such due to its distributed storage and data processing capabilities. Soon developers built alternative distributed and parallel systems from scratch for large-scale machine learning.

One of the most successful has been Spark [9]. Hadoop is built to process simple map-reduce jobs and not to repeatedly process data as is needed in machine learning. Hadoop reads in the data from the disk, maps the data to an intermediate form,
reduces the intermediate data and eventually writes the results back to the disk. All of this disk I/O is expensive for multi-pass data processing. Spark addresses this by allowing cyclic dataflow. It caches important data or intermediate results in memory (using resilient distributed datasets, or RDDs [10]). This directly facilitates large-scale optimization in machine learning, which is usually iterative.

However, despite the powerful set of APIs (transformations and actions) provided by Spark, a programmer always finds himself/herself stuck in deciding which RDDs to persist, whether an RDD should be stored in memory or persisted to disk, in what granularity should input data be partitioned and parallelized, and so on. None of these is an easy decision to make for the programmer.

Another recent scalable platform targeting machine learning applications is Google's TensorFlow [11]. With good performance on machine learning computations as well as its ability to automatic differentiate machine learning models, TensorFlow was quickly adopted widely in both industry and academia. However, TensorFlow is not easy to use for distributed machine learning. The so-called “parameter server” compute engine architecture [12] used by TensorFlow is fundamentally a distributed key-value store, where a set of machines are specialized as the servers that store the model fragments. TensorFlow requires a programmer to be a human optimizer and make critical implementation decisions such as how data and model are partitioned and stored, which machines serve which data or model, how to write optimized code given the size of the problem and the computation between data and model, etc. All these mean that getting many distributed computations to work on TensorFlow is not easy, especially when the data or model is very large.

In fact, when coding up a machine learning problem, it would be ideal if a programmer were only asked to manage the logic governing interaction between the data and
model parameters of the algorithm, instead of managing implementation details such as data representation, data affinity, memory usage and execution plan generation. That is exactly why declarative systems and the relational model should be considered. The attractiveness of the relational approach and declarative programming lies in data independence [13], where a programmer only needs to specify what he/she wants from the computation, without worrying about how to achieve it [14]. This makes the codes for large scale machine learning applications portable and independent of lower-level details, such as storage and hardware. A lot of the optimizations, choice of algorithms, and implementation of parallelism is left to the optimizer and the execution engine, making declarative systems a natural fit for large-scale machine learning.

1.1 Overview of My PhD Research Projects

My PhD research has been concerned with the design and implementation of scalable machine learning systems, with an emphasis on applying ideas from declarative programming and database systems. I study three specific problems in the area.

First, I study the parameterization of user-defined functions with data and corresponding model parameters in a distributed setting, when both data and model sizes are large. If not done wisely, the physical execution plan might waste a lot of time in repetitive parameterization or fail to complete due to memory overflow. I propose three approaches for efficient parameterization, and demonstrate their effectiveness on a declarative system as well as a dataflow system.

Second, I study how to adapt an SQL database to facilitate a declarative approach to large-scale recursive model specification (common in deep learning) that can allow programmers to write distributed codes without worrying about model placement and
manual optimization of the computation. In contrast, for very complex or (especially) very large learning problems, it can be difficult to implement model parallel learning on top of a parameter server, where workers must synchronously update parts of the model being learned.

Third, I study some of the pitfalls associated with a distributed system to run large-scale machine learning computations, as well as possible solutions. Specifically, I consider unbalanced partitioning in distributed linear algebra, poor choice of physical plans, and the overhead of runtime compilation.

1.2 Declarative UDF Parameterizations

**Problem Description.** Large-scale optimization has become an important application for data management systems, particularly in the context of statistical machine learning. In this chapter of the thesis, I consider how one might implement the join-and-co-group pattern in the context of a fully declarative data processing system.

The join-and-co-group pattern is ubiquitous in iterative, large-scale optimization. In the join-and-co-group pattern, a user-defined function $g$ is parameterized with a data object $x$ as well as the subset of the statistical model $\Theta_x$ that applies to that object, so that $g(x|\Theta_x)$ can be used to compute a partial update of the model. This is repeated for every $x$ in the full data set $X$. All partial updates are then aggregated and used to perform a complete update of the model.

The join-and-co-group pattern has several implementation challenges, including the potential for a massive blow-up in the size of a fully parameterized model when the results of the joins have to be materialized. Thus, unless the correct physical execution plan be chosen for implementing the join-and-co-group pattern, it is easily possible to have an execution that takes a very long time or even fails to complete.
For example, in some machine learning models, the UDF $g$ contains a matrix inverse calculation once $g$ is parameterized with $\Theta_x$. And in many cases, the same matrix inverse is associated with many data objects (denoted as $x_1, x_2, \ldots$). If the matrix in $\Theta_x$ is large in dimensionality, both the parameterization of this matrix and its inverse computation will be very expensive, because the same matrix is repeatedly loaded into $g$ and then inverted for each of $x_1, x_2$, and so on.

Therefore, I carefully consider the alternatives for implementing the join-and-co-group pattern on top of a declarative system, as well as how the best alternative can be selected automatically. My focus is on the SimSQL database system [15], which is an SQL-based system with special facilities for large-scale, iterative optimization. Since it is an SQL-based system with a query optimizer, those choices can be made automatically. A Spark implementation was also completed.

**Proposed Approach.** There are three ways to implement the join-and-co-group pattern, based on the frequency and location (among the compute cluster) of UDF parameterization for each co-group:

**Naive.** This method consists of a series of joins followed by a co-group that is pipelined into the UDF. For a particular $x \in X$ and its corresponding co-group in $\Theta_x$, the UDF $g$ re-parameterizes itself with the co-group every time.

**Local.** If a large number of data points $x_1, x_2, \ldots$ all share the same parameter set $\Theta$, then one can first parameterize the UDF with $\Theta$, and then use the same, parameterized UDF to compute $g(x_1|\Theta), g(x_2|\Theta)$, and so on. In this implementation, the locally-parameterized UDF is saved, and re-used across invocations whenever possible—hence the name “local”.

**Global.** The problem with local is that there will often be cases where it is *not* possible to store all of a parameter set $\Theta$ in RAM. This issue can be addressed by
partitioning a single copy of the model globally, and then moving the data to the relevant portion of the model. This way, a unique parameterization of $g$ occurs in just one location globally in the compute cluster.

**Overview of Results.** We benchmarked our join-and-co-group implementations in both SimSQL and Spark using three machine learning tasks, including linear regression, Gaussian mixture model (GMM) [16] imputation and LDA [5]. There are two goals for our benchmarking. One is to determine the advantages and disadvantages of the naive, local and global methods for each machine learning task and in each system. Another is to see how a declarative system like SimSQL compares in efficiency and ease of programmability compared to a non-declarative dataflow system such as Spark. Based on the benchmarking results, we have found that:

1. For all three machine learning tasks in SimSQL and Spark, local was always the best of the three whenever it could complete. The advantage in GMM learning is especially significant because processing the GMM model requires expensive matrix calculations that can be re-used across data points.

2. Global was the only one that could complete LDA learning in SimSQL for both a small model size and a large model size. The LDA model itself is many gigabytes in size, leaving it the only option to run global.

3. Compared to SimSQL, it was harder to make all three methods work on top of Spark, due to the 2GB frame size limit Spark had as well as the difficulty of fitting a large model into the driver memory and broadcasting to the executors. Thus, the only LDA benchmarking that was successful in Spark was local under a small model size.

4. Thanks to the declarative SQL front end and the cost-based optimizer in Sim-
SQL, machine learning applications can be simpler and even more efficient given how difficult it is to make key implementation decisions in Spark.

Overall, our experiments showed that it is easier to produce a high-performance implementation of the join-and-co-group pattern within a parallel database system where it is possible to control the low-level details of the implementation, than it is to implement the join-and-co-group pattern on top of a dataflow platform.

### 1.3 Declarative Recursive Computation

**Problem Description.** Many modern big data computations have complicated dependencies. For example, in the forward pass of a feed-forward neural network, the activations of neurons in one layer depend on those in the previous layer. For another example, the errors in one layer depend on those in its following layer in the backward pass [17]. Such dependencies result in a recursive computation.

Much of the cutting-edge work in the area assumes that a parameter server is the right compute platform for such recursive computations. For example, consider Google’s TensorFlow [18], which allows programmers to build large-scale machine learning computations using a relatively high-level “dataflow graph” specification that describes the dependencies in a deep neural network or other machine learning abstractions. In a distributed TensorFlow cluster, some machines are designated as servers which typically store fragments of the model, addressable via a unique key. The remainder of the machines are designated as workers which communicate with the servers and complete the computations. However, when model gets large in size (for example, due to large number of neurons in a dense neural network) and model parallelism [1] is required, it is often beneficial to the programmers to declaratively express their ultimate goals to achieve [19], without getting into the complicated
details on model partitioning or placement over hardware and being responsible for coming up with the most optimized implementation.

Therefore, TensorFlow with its parameter server paradigm is not necessarily the right choice when it comes to building large-scale, model parallel machine learning applications. Instead, we consider a relational database based solution having an SQL-like front end and a cost-based optimizer that automatically makes key decisions (such as data representation, data materialization and order of operators) using the data/model size and taking into account the hardware and compute resources. RDBMSs are designed around the idea of data independence, with the computation optimized to match the data and hardware. This is the “one implementation, any model/data size and compute hardware” ideal we strive to achieve.

**Proposed Approach.** Despite the fact that RDBMSs fit distributed machine learning well, they normally provide no or little support for recursion. Hence, there are two main challenges addressed in this chapter, and I have implemented these changes in our prototype relational database system—SimSQL. The first is how to extend SQL so that it can support recursive, machine learning-like computations without using imperative constructs. Our approach to tackling this problem is to enable multiple versions of a database table via one or more array-style indices. Using these indices, various versions of the tables can be defined recursively. SimSQL is also extended to be able to unfold the recursion and represent all the required computation as one whole graph of operators (or equivalently, relational algebra). In this way, the optimizer gains control of all the computation, freeing the programmer from any difficult decision.

The second challenge is how to run very large plans generated when SQL is extended with recursion. We propose a frame-based execution strategy for running the
very large plans (perhaps tens of thousands of relational operations) that result. For example, the implementation of a ten-layer feed-forward neural network for 100 training iterations results in 56,900 operators. Obviously, no existing RDBMS can handle this. Our solution is to cut the query plan into smaller *frames*. Still, data between frames has to be materialized, which means frame cuts need to be chosen carefully to maintain good performance. The frame cutting algorithm we propose associates costs with operator-operator pairs, and in so doing attempts to reduce the number of *pipeline breakers* induced. We carefully design a cost function so that the cost of an operator-operator link is determined by operator types and the number of consumers of the source operator. The higher this cost, the more unlikely the frame cutting algorithm cuts the link between the operators. In order to speed up frame cutting, we propose two heuristic solutions based on a greedy algorithm that builds frames one at a time.

**Overview of Results.** To demonstrate our added support for recursion, I implemented three recursive machine learning algorithms (feed-forward neural networks, Word2Vec [20] and LDA with collapsed Gibbs sampling [21]) in SimSQL, TensorFlow and Spark. Using those implementations, I evaluate the proposed SQL extensions from three aspects (utility of plan cutting, scalability with respect to model size and scalability with respect to compute cluster size). Some remarkable findings are:

(1) Non-greedy plan cutting with tuned costs based on operator types worked the best for both feed-forward neural network and Word2Vec in terms of per mini-batch latency.

(2) For all three algorithms, the latency of TensorFlow or Spark was lower than that of SimSQL for smaller model sizes. But SimSQL had better scalability with respect
to model size and gradually outperformed the other two systems with the increase of model size. Also, the data parallel implementations in TensorFlow could not handle very large model sizes due to memory overflow, while the model parallel SimSQL implementations are immune to such failures.

(3) Model parallel implementations in SimSQL achieved good scale out with an increasing number of machines in the cluster and fast convergence (reflected by the observed per-hour noise-contrastive estimation (NCE) loss [22] of Word2Vec). In contrast, TensorFlow suffered from the staleness of model updates due to its asynchronicity in Word2Vec training. A lot of work was being done using old parameters, leading to slower convergence.

(4) Model parallel implementations are easy in SQL. When model size or the number of machines in the cluster changed, no change needed to be made to the SQL codes.

Therefore, we have given strong evidence through our experiments that a parameter server is not the right platform—that a RDBMS with an extended SQL API is a better choice.

1.4 Overcoming Performance Bottlenecks in Distributed Machine Learning

Problem Description. Distributed systems targeted at machine learning suffer from a few performance bottlenecks. Here, I list three that can be found in many distributed platforms.

(1) In distributed systems, data partitioning is usually done with hash partitioning to achieve a relatively balanced distribution through randomness. For some data object, the hash value of its key moded by the number of workers/cores is commonly used as
the index of the unit housing that object. This solution is problematic for distributed linear algebra computations (common in machine learning), in which a large matrix is often chunked into a set of sub-matrices indexed by row and column block identifiers.

To reduce the amount of communication, we favor large sub-matrices when chunking a large matrix. Thus, the number of data objects in distributed linear algebra is usually not large. However, hash partitioning has the implicit assumption that the number of data objects is large. If the row and column identifiers of sub-matrices are distributed with hash partitioning, some workers might get more than the average number of identifiers while some other workers get zero, resulting in inefficiency.

(2) In a distributed system, either the optimizer of the system or the programmer is responsible for figuring out the best physical plan for each computation, and avoiding suboptimal choices that may result in shuffling and sorting of large amount of data. This can be difficult in the case of complicated computations. For example, the activation table, the weight table and the error table need to do a three-way join [23] in the backward pass of neural network training. Given a model parallel implementation, it is non-trivial to choose the correct table to hash and the correct table to pipeline so that the entire three-way join is efficient. A simple heuristic that hashes the smaller table in memory (if that table can fit in memory) and pipelines the other can cause a performance slowdown: if the pipelined table in the join of the first two tables fails to follow the same order as the third table, expensive sorting and data materialization will be inevitable.

(3) Runtime compilation involves compilation of new code during the execution of an application rather than at compile time prior to execution [24]. This is common in a distributed data processing platform, as it allows the system to dynamically react to the changes in an application and determine the execution plan on the fly after each
run or deployment in the cluster. However, runtime compilation incurs quite a bit of overhead as well. For example, in a Hadoop based system such as SimSQL, the master machine compiles a jar file which is later deployed to the cluster and executed by the worker machines. During the compilation on the master machine, the workers end up idle. The compilation of one jar file can take about 12 seconds. For a feedforward neural network implementation in SimSQL that translates to 15 MapReduce jobs per iteration, the inter-job idle time for workers can be as bad as $15 \times 12 = 180$ seconds, about 25% of the per-iteration time.

**Proposed Approach.** To address these issues, I propose the following:

1. In the case of distributed matrix computation, a round-robin partitioning is used instead of random hash partitioning. Specifically, data partitioning is done by direct 
\texttt{moding} instead of by hashing and then \texttt{moding}. Since dense matrices are not skewed by nature, this leads to a balanced distribution of matrix blocks among workers/cores. Hence, removing the hash spares a distributed platform from stragglers.

2. In the case of a three-way join (for example, $(A \bowtie B) \bowtie C$), the new heuristic will favor order-preserving opportunities in selecting the table to hash and the table to pipeline. Instead of simply hashing the smaller table between $A$ and $B$, and pipelining the other to further join with $C$, the new heuristic examines if one of table $A$ and table $B$ is sorted and joined in the same order with $C$. If so, a better physical plan is to pipeline the table that can be directly merge-joined with $C$ and hash the other in memory if applicable, therefore avoiding any data reshuffling or sorting.

3. In a distributed machine learning task made up of multiple runs or deployments, runtime compilation of one run takes into account the physical characteristics of data or statistics (such as the size of data serialized) of its previous run. Given different
statistics, the system generates different physical plans accordingly, which in turn leads to different compiled codes for execution. Without knowing the actual statistics of the previous run (by waiting for it to complete), it is difficult for the system to pre-compile for the next run. Fortunately, in many iterative machine learning applications, the statistics of the previous run can be inferred without seeing the data, because physical characteristics of data remain stable over iterations. Hence, by multithreading, a distributed system can execute one run and pre-compile for its next run concurrently using the statistics inferred from earlier iterations.

**Overview of Results.** I evaluated two machine learning tasks before and after each of the proposed changes were made in SimSQL. Some notable results are:

1. With the new partitioning strategy for model parallel computations, the runtime of both feed-forward neural networks and block matrix multiplication decreased by as much as 57%. The improvement was especially obvious when the number of data partitions was smaller than the number of workers/cores.

2. Evaluation of the efficacy of the new three-way join heuristic was performed using feed-forward neural networks and Gaussian mixture models. An increase of 10%–12% in overall performance was observed. The three-way joins alone were 32%–58% faster.

3. In the case of pre-compilation, the per-minibatch latency of feed-forward neural networks and Word2Vec directly benefited from the drop of idle time waiting for the compilation to finish. Since the compilation time is mostly fixed, the performance gain decreases with the increasing model complexity.
1.5 Thesis Organization

The remainder of the thesis is organized as follows. In Chapter 2, we identify the join-and-co-group pattern for distributed UDF parameterization and consider three implementations of this pattern within an RDBMS and on top of a dataflow system. In Chapter 3, we consider how one might extend an RDBMS to support declarative recursive computation and how this compares with a parameter server based system and a dataflow system. In Chapter 4, we examine the performance bottlenecks and various improvements. Related work is covered in Chapter 5, and Chapter 6 concludes the thesis.
Chapter 2

Declarative Parameterizations of User-Defined Functions

2.1 Motivation

Large-scale optimization has become an important application for data management systems. Given a data set $X$ and a function $f$ that measures the quality of a set of model parameters $\Theta$, the goal in large-scale optimization is to choose $\Theta$ such that $f(\Theta|X)$ has a large (or small) value.

The importance of optimization as an application of data management is due in no small part to the fact that optimization underlies the learning task in statistical machine learning, which is now a standard data analytics task. Depending on the particular learning task, $f$ may be a loss function in the case of a neural model or a likelihood function in the case of a statistical model [16], a distance or divergence [25], or a posterior probability distribution function in Bayesian learning [26]. Also depending upon the learning task, the method used to optimize $\Theta$ can vary as well: gradient-based methods [27] and second-order methods [28] are common, as are EM [29] and variational inference [30], as well as Markov chain Monte Carlo methods [31].

Despite the wide variety of functions optimized as well as the wide variety of optimization strategies employed, virtually all large-scale optimization implementations follow the same general pattern:

1. For each data point $x \in X$, the subset of the model parameters relevant to $x$ is
computed. We’ll call this subset $\Theta_x$.

2. Then, that subset is fed into a problem-specific user-defined function $g(x\mid \Theta_x)$, that computes a partial update of the parameter set $\Theta_x$.

3. Finally, all of the individual calls to $g(x\mid \Theta_x)$ result in a set of statistics that are aggregated and together used to update $\Theta$ globally.

This process is then repeated iteratively until convergence.

**Varying Complexity of Practical Applications.** Depending upon the particular optimization problem, realizing this pattern on a large problem can be trivial, extremely difficult, or anywhere in between.

For example, if the parameter set $\Theta$ is reasonably small, then $\Theta$ can be distributed around a compute cluster as a broadcast variable or via a lambda capture (assuming a dataflow platform such as Spark or Flink). Then, $g(x\mid x)$ can be computed for each $x$ via a map operation over $g(x\mid \Theta)$, where the implementation of $g(x\mid \Theta)$ references the local or captured copy of $\Theta$.

For an example where the task of parameterizing and applying $g$ is this simple, consider gradient descent for linear regression. In this case, $\Theta$ would contain the current set of regression parameters. Assuming that the number of regression coefficients is in the low millions (so that $\Theta$ is not too large) $g$ would likely be implemented as a lambda that captures $\Theta$. It would use the captured $\Theta$ to compute the gradient for the current data point $x$—in this case, a vector whose dimensionality is the same as the dimensionality of the data. The aggregation step would then sum all of the gradient vectors, and then the sum would be used to update $\Theta$.

However, things can be more complicated. In the general case, the parameter set $\Theta$ can be large—gigabytes or even terabytes in size—and complicated. $\Theta$ may maintain
many subclasses of parameters, so that $\Theta = (\Theta_1, \Theta_2, \ldots, \Theta_k)$. Each subset contains a different type of model parameter. For example, in a probabilistic graphical model, $\Theta_1$ may contain the graph structure, $\Theta_2$ parameters associated with the nodes, $\Theta_3$ global distributional parameters, and so on. In this case, the process of associating an appropriate subset of $\Theta$ with each $x \in X$ must typically be implemented as a series of joins: $\Theta_{X,1} \leftarrow X \bowtie \Theta_1$, $\Theta_{X,2} \leftarrow X \bowtie \Theta_2$, ..., $\Theta_{X,k} \leftarrow X \bowtie \Theta_K$. Next, all of the various $\Theta_{X,i}$'s resulting from these joins are co-grouped—typically according to an identifier from $X$. After the co-grouping, each of the co-groups contains a complete set of parameters for a data point. Each co-group is used to parameterize and invoke $g(x|.)$, whose output is then aggregated.

**The Value of Declarative Processing.** There do exist systems that can be used to run versions of the join-and-co-group pattern. For example, a Parameter Server \cite{32, 12} allows workers (who would implement the function $g(x|.)$ for a specific data object $x$) to request a specific subset of the model parameters, which is then sent to the workers according to those requests. For another example, dataflow platforms such as Spark \cite{9} and Flink \cite{33} provide a large number of transformations (maps, reduces, joins, etc.) that can be used to build a join-and-co-group.

However, none of those systems are declarative. Parameter Server codes are fast, but the burden to a programmer is high. Such codes are low-level and intricate. For example, the LightLDA implementation \cite{34} of a version of the learning algorithm for the popular LDA model \cite{5} on top of a Parameter Server is between two and four thousand lines of fairly intricate C++ code \cite{35}, depending on how one counts. The burden on a Spark of Flink programmer is much lower, but the details of how the underlying system implements the pattern can have a huge impact on the running time of the optimization procedure, meaning the difference between a slow and a fast
implementation, or perhaps meaning the difference between an implementation that crashes and one that runs to completion. A Spark or Flink programmer must make a number of difficult decisions on what order to perform operations, which intermediate results to materialize, which to broadcast, and how to represent the data. Making the wrong choice can result in a very slow code, or one that fails, and it is not often obvious what the right choice is. Further, the fact that Spark and Flink require a programmer to operate at a higher level of abstraction than a Parameter Server programmer can actually be a hindrance, as many of the lower level implementation choices are left to the platform, which lacks the ability to make the correct choice on a case-by-case bases, in a cost-based fashion.

Implementing Declarative Join-And-Co-Group. Since realizing the join-and-co-group pattern is fundamentally a data processing task, in this chapter, we advocate a declarative, database-oriented approach to solving the problem. We argue that it makes sense to provide syntax for declaratively specifying a join-and-co-group computation to a programmer, and we consider the problem of efficiently implementing SQL-based specifications of join-and-co-group. We consider in detail the various logical and physical implementation choices available to a system that is tasked with actually executing a declarative join-and-co-group specification. For example, \( g \) is typically idempotent, in the sense that once \( g \) has been parameterized on \( \Theta_x \), it can be run many times with different values of \( x \) that use the same set of parameters. If \( g \) is idempotent, since parameterizing \( g \) can be expensive—for example, requiring a computation such as a matrix inversion—an can automatically choose to parameterize \( g \) once and then re-use the parameterized \( g \) for all \( x \in X \) having the same parameterization. For another example of a physical optimization that may be available, the result of the join-and-co-group can be huge: perhaps many thousands of
times the size of the original data set, since parts of the model can be associated with many data items. This means that it is often a necessity to pipeline the result of the join-and-co-group into the application of \( g \). However, this pipeline requirement may present a challenge because the co-group is implemented via a sort/hash: how does one pipeline the result of a join through a sort or hash operation?

**Our Contributions.** In the chapter, we carefully consider how a function such as \( g \) should be parameterized in the context of a large-scale optimization or machine learning computation, specified using a declarative programming language such as SQL. Most of the paper is concerned with the implementation of such a parameterization in SimSQL [36], an SQL-based database system with special facilities for iterative computation. Since SimSQL is a declarative, SQL-based system, it is possible for the system’s optimizer to automatically make the choices necessary to optimize the parameterization.

The specific contributions of this chapter are as follows:

- We identify and describe the join-and-co-group pattern for parameterizing user-defined functions in large-scale optimization and machine learning applications.

- We carefully describe how the naive implementation of this pattern, as a series of joins followed by a co-group that is pipelined into the user-defined function, can have poor performance.

- We identify two alternative implementations: the *local* method where parameterizations may be repeated all over a compute cluster, and the *global* method where a unique parameterization of the function will occur in just one location.

- We consider the optimization problem of how to choose between the local and global implementations.
• We show experimentally in the context of several machine learning applications how our ideas can lead to radical performance improvements.

2.2 The Join-And-Co-Group Pattern

In this section, we describe the join-and-co-group pattern in more detail, as well as the syntax for utilizing the pattern in SimSQL, which is a parallel database system with special support for statistical computing. While the syntax to support join-and-co-group has been available in SimSQL previously [36], the problem of implementing the pattern efficiently has not been studied previously, and is the subject of the paper.

2.2.1 Generic Form

The join-and-co-group-pattern is common in iterative optimization and machine learning. A diagram depicting the overall flow of an iterative program for large-scale optimization is given as Figure 2.1. Pseudo code for the pattern is given as Figure 2.2.
initialize $\Theta = \langle \Theta_1, \Theta_2, ..., \Theta_k \rangle$;
while $\Theta$ still changes:

// link params with data points
for $i \in \{1...k\}$:
    $\Theta_{X,i} \leftarrow X \bowtie_b \Theta_i$;

// groups params according to data points
$G \leftarrow \text{co-group} \ (X, \Theta_{X,1}, \Theta_{X,2}, ..., \Theta_{X,k})$ on $X$.id;

// apply UDF to each co-group
for each $\langle x, \Theta_{x,1}, \Theta_{x,2}, ..., \Theta_{x,k} \rangle$ in $G$:
    $\Delta_x \leftarrow g(x|\Theta_{x,1}, \Theta_{x,2}, ..., \Theta_{x,k})$;

// and update the model
aggregate all $\Delta_x$ to obtain $\Delta$;
update $\Theta$ using $\Delta$;

Figure 2.2: Pseudo-code for iterative optimization via the join-and-co-group pattern.

The pattern is fairly simple, though one point does bear a bit of explanation: the co-group operation. This operation takes a list of sets $(X, \Theta_{X,1}, \Theta_{X,2}, ..., \Theta_{X,k}$ in our case) and partitions each set in the list according to a key; in our case, this key is the identity of the tuple from $X$ that joined with each parameter value. This results in a set of lists of sets, where each list of sets contains all of the parameters associated with a particular $x \in X$. Once this set of lists of sets is created via the co-group, the UDF $g$ is applied to each item from it to produce a local update.

2.2.2 Realization in SimSQL

A central thesis of this chapter is that the join-and-co-group pattern is so central to large-scale optimization that any platform supporting large-scale machine learning or other, similar pattern should provide support for join-and-co-group. In this section, we discuss how, from a programmer’s perspective, join-and-co-group can be supported. We will subsequently discuss how it might be supported in a dataflow
platform such as Spark.

In SimSQL, the join-and-co-group pattern is realized via a `CREATE TABLE` statement with a special `FOR EACH` comprehension. In general, this statement takes the following form:

```sql
CREATE TABLE tableName (att1, att2, ...) AS
  FOR EACH x IN X
    WITH temp AS g (  
      (Subquery to compute \(\Theta_{x,1}\))  
      (Subquery to compute \(\Theta_{x,2}\))  
      ...  
      (Subquery to compute \(\Theta_{x,k}\)))  
  Query to assemble \(\Delta_x\) from temp
```

Logically, this code “loops” over all of the tuples in the table \(X\). For each tuple \(x\) in \(X\), a series of parameter tables are computed via a list of user-specified subqueries. These parameter tables are then used to parameterize the user-defined function \(g\) (called a `VG function` in SimSQL). A VG function is a special table function taking in one or more sets of tuples, and returning a set of tuples. It is declared as:

```sql
CREATE VGFUNCTION g
  (dataParam1 type, dataParam2 type, ... |  
   modelParam1 type, modelParam2 type, ...)  
RETURNS
  (output1 type, output2 type, ...)
LOAD FROM myFile.so;
```

Input parameters are divided into model and data parameters. As mentioned previously, the VG implementation must be idempotent in the set that once the VG function is parameterized with model parameters, it can be re-parameterized with data parameters and re-used. This idempotency can be taken advantage of by the system. VG functions are tables functions, returning a set of tuples corresponding to the output schema. VG functions in SimSQL are implemented as C++, and loaded dynamically as shared libraries.

One important aspect of VG function parameterizations is that a VG function does not accept just a single tuple or row as a parameterization. Indeed, the subqueries that
compute $\Theta_{x,1}, \Theta_{x,2}, \text{etc.}$ can each return multiple rows. In SimSQL, a VG function is implemented as a C++ class with a special `takeParams()` method. For each tuple in each of the sets $\Theta_{x,1}, \Theta_{x,2}, \ldots$, `takeParams()` is called. This is important, because it allows quite complex parameterizations.

Likewise, a VG function can produce multiple rows as outputs. In SimSQL, the VG function has a `outputVals()` method that is called multiple times, with each call producing another record, until a `null` is returned. Once a VG function has been parameterized and invoked for a particular $x$, a final query is used to assemble one or more output tuples that are inserted into the table named by `tableName`. This final query essentially post-processes the results of the VG function.

### 2.2.3 Examples

In this section, we give two examples of the join-and-co-group pattern, described using the SimSQL syntax.

**Gradient Descent.** For a simple (and somewhat contrived) example of this, imagine that we have a very high-dimensional and sparse set of feature vectors, and the goal is to learn a regression model via gradient descent. Data are stored in the following tables:

- `data (dataID, response double)`
- `features (dataID int, dimID int, value double)`

And the current vector of regression coefficients is:

- `coefs (dimID int, regCoef double)`

For each data point $x = \langle x_1, x_2, \ldots, x_d \rangle$ with response $y$, the gradient descent algorithm requires that we compute a set of regression coefficient updates of the form $(i, \Delta_i)$ where the update to coefficient $r_i$ is $\Delta_i = x_i(-2y + \sum x_j r_j)$. In SimSQL this is:
Figure 2.3: Naive join-and-co-group for LDA. The entire topics portion of the model is broadcast to each data point. For a large model, this is unrealizable.

CREATE TABLE deltas (dimID, value) AS
  FOR EACH x IN data
    WITH res AS ComputeGrad (
      (SELECT x.response)
      (SELECT f.dimID, f.value, c.regCoef
       FROM features AS f, coefs AS c
       WHERE f.dataID = x.dataID AND
       f.dimID = c.dimID))
    SELECT r.dimID, r.val
    FROM res AS r;

This code considers each data point d in sequence. For each, it selects the response (the first subquery) and the set of (dimID, value, regCoef) triples relevant to the data point. From this, \( x_i(-2y + \sum x_jr_j) \) is computed by the ComputeGrad VG function for each value of \( i \) (\( i \) is the dimID). The result is output as a set of (dimID, value) pairs. A subsequent aggregation over the resulting deltas table will complete the gradient update.

**Latent Dirichlet Allocation.** Gradient descent for linear regression is illustrative
first example, but it could be implemented in SQL, without using the join-and-co-group pattern.

For a more interesting computation that is unrealistic without using the join-and-co-group pattern, consider the task of implementing a very large LDA model [5] (also known as latent Dirichlet allocation) over a document corpus using a Gibbs sampler.

LDA is a ubiquitous Bayesian model, used for mining large document archives. In LDA, the learned statistical model has two distinct components: a set of topics, and a set of topic prevalence. Each topic in an LDA model is a dense vector of word probabilities, where the $k$th entry in the vector associated with topic $j$ gives the probability that an arbitrary word produced by topic $j$ will correspond to the $k$th entry in the dictionary. Each topic prevalence is a dense vector of topic probabilities, where the $j$th entry in the topic prevalence vector associated with document $i$ gives the probability that an arbitrary word in document $i$ was produced by topic $j$.

```java
*topics (topicID int, wordProbs vector[]) topicProbs (docID int, topicProbs vector[]) The data are then stored in the following database table:

docs (docID int, wordCnts vector[])

Here, the $k$th entry in a given wordCnts vector records the number of times that dictionary words $k$ appears in the document.

---

*Note that Gibbs sampling, which we consider here, is the typical way that an LDA model is learned from a text corpus. Since the “collapsed” LDA Gibbs sampler first appeared in the journal *Science* in 2004 [21], it has been the standard inference algorithm for LDA. Here “collapsed” means that one or more variables have been integrated out in the derivation of the Gibbs sampler—in the case of LDA only the assignments of words to topics are left—which will typically result in an MCMC simulation that converges more quickly. We choose, however, consider the non-collapsed sampler here for two reasons. First, it is more interesting as a benchmark because it is a bit more complicated, with multiple parameter sets (topic prevalences and word prevalences). Second, there is the issue of correctness. It is very challenging to parallelize the collapsed LDA Gibbs sampler correctly because of the complex correlation structure that the collapsing induces among the updates to the various word-to-topic-assignments. Most parallel/distributed LDA Gibbs samplers described in the literature simply ignore these correlations and update the vectors in parallel, disregarding the effect of the (incorrect) concurrent updates of the word-to-topic assignment vectors.
Join-and-co-group in this case requires that we take a cross product of topics and docs, then join topicProbs and docs on docID = docID to obtain the topicProbs vector for each document. The two resulting tables are co-grouped on the identity of each document, and then the resulting sets are sent into a UDF that associates each word in the document with a particular topic. The ultimate result of all of these UDF calls is a set of wordCnts matrices, where wordCnts[j][k] is the number of times that topic j produced words k for a given document.

One problem is that for a very large model, computing the cross product of topics and docs is going to be totally unmanageable, no matter the implementation of join-and-co-group. This is depicted in Figure 2.3; the entire contents of topics are broadcast to each document. This is problematic because the topics table can easily be hundreds of gigabytes in size; consider the realistic case where the dictionary (that is, the set of words in the corpus) contains ten million symbols, and the goal is to learn ten thousand topics. The corresponding 100 billion double precision values stored in the topics table will require at least 800 GB to store. This means that the co-group computed will contain nearly a TB of data. It is likely impossible to feed this much data into a UDF.

We can address this by de-normalizing topics and docs into a set of blocks:

topics (topicID int, wordBlockID int,
  wordProbs vector[])
docs (docID int, wordBlockID int,
  wordCnts vector[])  

Thus, the join-and-co-group can be run on a per-wordBlockID-basis. The resulting join-and-co-group is implemented in SimSQL as:
Figure 2.4: Join-and-co-group for LDA. In this case, topics is broken into blocks (subsets of words) so that the entire model need not be broadcast to each and every document during join-and-co-group.

CREATE TABLE produced (docID int, wordBlockID int, produced matrix[][][]) AS
FOR EACH d IN docs
WITH res AS AssignTopics (  
(SELECT d.wordCnts)  
(SELECT tp.topicProbs  
FROM topicProbs AS tp  
WHERE tp.docID = d.docID)  
(SELECT t.topicID, t.wordProbs  
FROM topics AS t  
WHERE t.wordBlockID = d.wordBlockID))
SELECT d.docID, d.wordBlockID, res.produced
FROM res

The data flow for this implementation is depicted in Figure 2.4. The first sub-query obtains a sub-block of the word counts associated with the document identified
by docID. The second obtains the complete set of topic probabilities for that document. And the third obtains the complete set of word probabilities for each topic, for the sub-block of words in question. The AssignTopics UDF then uses those model parameters to assign words to topics, as encoded by the produced matrix. Specifically, the $k$th column of the produced matrix is sampled from a Multinomial distribution: $\text{produced}[*][k] \sim \text{Multinomial}(\pi, \text{wordCnts}[k])$, with the $j$th entry in the probability vector $\pi$ proportional to

$$\text{topicProbs}[j] \times \text{wordProbs}[k]$$

for the wordProbs associated with topic $j$.

### 2.3 Implementation

Building an efficient implementation of the join-and-co-group pattern can be a challenge. In this section, we consider three different implementations of the join-and-co-group pattern, which we call naive, local, and global. Our focus here is building join-and-co-group into a system providing a declarative, domain specific language such as SQL that makes it possible to perform some analysis and re-writing of the computation. In a subsequent section of the chapter, we consider implementation on a dataflow platform such as Spark.

#### 2.3.1 Naive Implementation

A schematic for the most straightforward implementation of the join-and-co-group pattern is depicted above in Figure 2.5. Though we call the straightforward implementation the “naive” implementation, it is actually quite intricate, and offers many
opportunities for optimization, as we will describe in detail subsequently. The steps involved in the naive implementation are as follows:

**Step 1: Seeding.** The data set $X$ is first “seeded”. *Seeding* here refers to the process of appending each data item from the set $X$ with a unique identifier. Seeding is necessary because the subsequent set of joins depicted in Figure 2.5 will associate each data item $x \in X$ with a set of parameter sets $\Theta_{x,1}, \Theta_{x,2}, \ldots$. The joins $\Theta_1 \Join X$, $\Theta_2 \Join X$, $\ldots$ that are used to compute those parameter sets can produce an arbitrary data ordering and/or partitioning in a distributed system. Appending each parameter $\theta \in \Theta_i \Join X$ with the seed value from the $x \in X$ that produced $\theta$ ensures that all parameters can subsequently be co-grouped in such a way that each $\theta$ in a parameter
set can be associated with the data item \( x \) that was used to compute it.

To implement seeding in a distributed and/or parallel system, each compute core is given a unique identifier, which is then concatenated with a counter maintained by the compute core to produce a unique seed.

**Step 2: Preparing parameters.** After seeding, the seeded data set \( X \) is joined with each parameter set \( \Theta_1, \Theta_2, \ldots \) to produce, for each data item, its own parameter set. This step effectively associates the relevant subset of the parameter set with each data point. In the most general case, each join is pipelined into a projection operation that contains an arbitrary computation whose task is to compute the final, prepared parameter. But regardless of this computation, each of the parameters produced by the set of joins should be appended with the seed value from the data item \( x \in X \) used to compute the parameter.

**Step 3: Co-group.** After joining, all parameters as well as the data set \( X \) are co-grouped, using a sort- or a hash-based algorithm. That is, each of the sets \( \Theta_1 \bowtie X, \Theta_2 \bowtie X, \ldots, \Theta_k \bowtie X \) is partitioned into a set of subsets \{\( \Theta_{x_1,1}, \Theta_{x_1,2}, \ldots \}\), \{\( \Theta_{x_2,1}, \Theta_{x_2,2}, \ldots \}\), \ldots \{\( \Theta_{x_k,1}, \Theta_{x_k,2}, \ldots \)\} where the grouping happens in such a way that each parameter in a subset \( \Theta_{x_i,j} \) has \( x_i \)'s seed. These partitions are then re-organized into a set of lists of the form \{\( \langle \Theta_{x_1,1}, \Theta_{x_1,2}, \ldots, \Theta_{x_1,k} \rangle, \langle \Theta_{x_2,1}, \Theta_{x_2,2}, \ldots, \Theta_{x_2,k} \rangle, \ldots \)\} via a merge, which completes the co-group.

**Step 4: UDF invocation.** Next, for each data item \( x \), the user-defined function \( g() \) at the core of the join-and-co-group is first parameterized with \( \langle \Theta_{x,1}, \Theta_{x,2}, \ldots, \Theta_{x,k} \rangle \), and then evaluated with \( x \). \( g() \) can then produce zero or more outputs per input co-group.

**Step 5: Final output assembly.** Finally, depending on the particular computa-
tion, it may be necessary to append each data item output by \( g() \) with information computed using the data set \( X \). This is implemented via a final join between the output of \( g() \) and the seed, performed at the end of the join-and-co-group.

While this is the basic outline of the naive implementation, we have found that there are several implementation details that can make the difference between a running time that is unacceptably long, and one that is very fast. We consider the most important details now.

First and foremost is the need, if at all possible, to avoid a sort or hash (or, more generically in a distributed system, a “shuffle”) of any of the data sets taken as input into the co-group. In fact, our naive implementation of the join-and-co-group on SimSQL never shuffles the input data set \( X \) to implement the co-group. This is possible since the co-group requires partitioning/ordering each of \( \Theta_1 \Join X \), \( \Theta_2 \Join X \), ..., \( \Theta_k \Join X \) as well as partitioning \( X \) based upon the seed value—each unique seed value than defines a group. Since the seeds are arbitrary values whose only requirement is that they be unique, as long as they are generated and attached to the data items in \( X \) in already-sorted or already-hashed order, then no work is required to prepare \( X \) for the co-group, since \( X \) will already be in the correct order.

Along the same lines, the join at the top of the join-and-co-group, where one or more attribute values from \( X \) are attached to the data items output from the user-defined function \( g() \), can be implemented as a simple and inexpensive merge without a sort or a hash, since \( X \) is already ordered based upon the seed value, and by definition, so are the outputs from the user-defined function calls. This is the case because the inputs to \( g() \) must be co-grouped by the seed value in order to correctly parameterize \( g() \), and hence the outputs are already correctly ordered.

Most importantly, each of \( \Theta_1 \Join X \), \( \Theta_2 \Join X \), ..., \( \Theta_k \Join X \) can be huge. This makes
it imperative to avoid materialization of the join result, if at all possible. It is not uncommon to attach an entire statistical model that is kilobytes or even megabytes in size to each and every data point in $X$, which can produce a huge join output. For example, imagine a Gaussian mixture model (a statistical model used for clustering data) learned over one million, one kilobyte data points having 100 dimensions each. This moderately-sized data set is a GB in size. However, with 20 clusters and a full $100 \times 100$ covariance matrix for each, the complete statistical model is going to be approximately 2MB in size for most reasonable implementations. Taking the cross product of this model with the data set results in a join result that is (one million points) $\times$ 2MB = 2TB of data. This sort of 1000X blow-up in size (or more) after parameterization is not uncommon.

To handle this, assuming that $X$ is distributed around a compute cluster, our prototype implementation aggressively attempts to implement each of those joins as so-called scan joins, where complete copies of $\Theta_1$, $\Theta_2$, ... are broadcast around the compute cluster, and a local copy is maintained at each machine. If the local copy can be stored as RAM, then the local subset of $X$ can be scanned and joined in-RAM with the cached, local copy of each $\Theta_i$. There are two benefits to this. First, the result of the join will already be ordered on the seed, assuming that $X$ is already ordered on the seed. Hence, the co-group of $\Theta_i \bowtie X$ is costless. Second, the result of the join can be pipelined directly into the user-defined function $g()$, without every being materialized and written to secondary storage. In our Gaussian mixture model example, the complete statistical model would be broadcast to every machine, buffered in RAM, and a copy of the model pipelined into $g()$ in order to parameterize it for every $x \in X$. This means that the 2TB parameterized model is never actually materialized; it exists only ephemerally as it is pipelined into the
user-defined function.

2.3.2 Local Implementation

The naive implementation has one obvious problem: it does not make use of the idempotency of $g(x|\Theta_x)$. That is, if a large number of data points $x_1, x_2, \ldots$ all share the same parameter set $\Theta$, then one can first parameterize the UDF with $\Theta$, and then use the same, parameterized UDF to compute $g(x_1|\Theta)$, $g(x_2|\Theta)$, and so on.

For some, almost-trivial computations (for example, linear regression via gradient descent), the parameterization of $g(.)$ is essentially costless. For other computations, parameterization of $g(.)$ can require, for example, expensive matrix computations involving the items in $\Theta$. In this case, it makes sense to re-use the parameterization, if at all possible.

This brings us to the so-called “local” implementation. In the local implementation, the locally-parameterized UDF is saved, and re-used across invocations whenever possible—hence the name “local”.

At the highest level, this re-using is accomplished by caching all of the parameterized UDFs that have been computed at a particular compute node. If and when too much RAM is consumed by those cached UDFs, then UDFs can be evicted from the cache using a standard cache replacement policy, such as LRU. While this idea is quite simple, there are a few important details to consider.

**Determining Re-use Opportunities.** One key question is: How do we recognize when a cached UDF can be re-used for a particular UDF invocation $g(x|\Theta_x)$? That is, we must determine when two tuples $x_1, x_2 \in X$ must have the same set of parameters. Assuming an SQL-based system with an extended SQL syntax as described in the previous section, we can do this by making a few changes to the query compiler and
optimizer.

First, during compilation of the SQL and transformation into an algebraic representation, the query compiler determines all of the attributes from \( X \) that are referenced during the set of joins \( \Theta_1 \bowtie_{B_1} X, \Theta_2 \bowtie_{B_2} X, \) and so on, and which provide input to the UDF’s so-called “model parameters”. Call this set of attributes \( A_\Theta \).

Two tuples \( x_1, x_2 \in X \) will have exactly the same parameterization if and only if, for each \( a \in A_\Theta \), \( x_1.a = x_2.a \). In classical database theory, all of the model parameter attributes will have a so-called “multi-valued dependency” on the set \( A_\Theta \) [37].

For example, consider LDA’s \texttt{CREATE TABLE produced} from Section 2.2.3 of the thesis. The \texttt{AssignTopics} UDF has the input schema:

\[
\text{AssignTopics (wordCounts vector[]} | \\
\text{topicProbs vector[]}, \text{topicID int}, \\
\text{wordProbs vector[]})
\]

so \texttt{topicProbs}, \texttt{topicID}, and \texttt{wordProbs} are the model parameters. In this case, \( X \) is the set \texttt{docs}, and \( A_\Theta \) will contain the attributes \texttt{docID} and \texttt{wordBlockID}. Thus, two tuples from \texttt{docs} will have the same parameterization if and only if \( x_1.docID = x_2.docID \) and \( x_1.wordBlockID = x_2.wordBlockID \).

Once \( A_\Theta \) has been determined, the query compiler/optimizer must make sure not to project away those attributes after seeding \( X \). Consider Figure 5. Typically, the query compiler/optimizer will choose not to include any attribute from \( X \) along the direct path to the UDF unless they will be used to parameterize the UDF. In our LDA \texttt{CREATE TABLE produced} example, only \texttt{docs.wordCnts} is used to provide data parameters to the UDF, and so all other attributes are projected away. However, in order to determine whether two tuples \( x_1, x_2 \in X \) have the same values for all parameters in \( A_\Theta \) (and hence will have the same parameterization for their respective UDFs) we must not project away \texttt{docs.docID} and \texttt{docs.wordBlockID}.
Then, during the parameterization of the UDF, when a new tuple \( x \in X \) is processed, the first thing is to hash all attributes in \( A_\Theta \) to determine whether we have an already-parameterized UDF that can be used. In this case, the already-parameterized UDF is used, and any model parameters that have been computed for \( x \) can be discarded.

**Further Optimizations.** The above optimizations consider how to avoid re-parameterizing a UDF when an appropriate, already-parameterized UDF is already cached. But it is desirable to do even better, and avoid computing those parameters altogether.

As described in the previous subsection, if one of the input parameter tables \( \Theta_i \) can be buffered in RAM, then it can be broadcast around the cluster and joined on-the-fly with tuples from \( X \) as they are pipelined, already ordered based on the seed, into the merge that implements the join-and-co-group pattern’s co-group.

In the local implementation, we can go a step further. If, before we perform the scan join, we first check for a particular \( x \in X \) whether we can use a previously-parameterized UDF for \( x \), then if we find that we *can* use a previously-parameterized UDF, we can avoid pushing \( x \) through any scan joins that produce model parameters. In this way, for an \( x \) that can make use of a previously-parameterized UDF, we do not even compute the set of model parameters for \( x \). This can save a considerable amount of computation.

### 2.3.3 Global Implementation

The local implementation strives to re-use parameterized UDFs, in order to avoid parameterizing UDFs whenever possible. If it is applicable, it is going to be an excellent option.

**Limitation of the Local Implementation.** However, the local implementation
has a key weakness that it inherits from the naive implementation: the join-and-co-
group must run a co-group by ordering the data resulting from the joins $X \bowtie \Theta_1$, $X \bowtie \Theta_2$, ... on the value of the seed. If the result of each join is huge—as can often be the case—this is simply not feasible.

Section 2.3.1 of the chapter outlined a set of optimizations that can mitigate the problems associated with huge parameterizations. The most important optimization is pipelining a copy of $X$ into each join. If a copy of the parameter set $\Theta_i$ can be stored in RAM, then there is no need to actually materialize the join result—it can simply be piped into the UDF, and, as long as this pipelining does not change the ordering of $X$, there is no need to re-order the result of the join in order to implement the co-group.

The problem with this optimization is that there will often be cases where it is not possible to store all of a parameter set $\Theta_i$ in RAM. In Section 2.2 we outlined the realistic case where information about the current topics in an LDA implementation can be near one TB in size. In such a case, there is no way to avoid running the complete join $X \bowtie \Theta_i$ and then re-ordering the join output based upon the seed in order to perform the co-grouping. And the join output can be enormous. Consider, for example, an LDA implementation with 100 million documents, 1000 topics, and a vocabulary of $10^5$ words split into 100 word blocks. The LDA parameterization requires that we compute $\text{docs} \bowtie \text{topics}$ on $\text{wordBlockID} = \text{wordBlockID}$. Assuming that 10% of the word blocks are non-empty, encoding the 100 million documents will require $10^9$ records; each record will join with a word block from each of the 1000 topics, for a total of $10^{12}$ tuples. Since each record resulting from the join will contain a dense, length $10^5/100 = 10^3$ vector taking approximately 10KB to store, the total size of the join will be $10^{12} \times 10^5 = 10^{17}$ bytes, or 10PB. Materializing and re-ordering
this set of parameters is going to require a fully shuffle of the data, and is simply not possible.

**The Global Implementation.** The global implementation avoids this by first identifying, through standard query optimization techniques, a potentially very large parameterization if it exists; docs $\bowtie$ topics in the case of LDA. Efficiently handling this parameterization is the target of the global implementation.

Just as in any distributed join algorithm for two large input sets, the set of parameters $\Theta_i$ input into this parameterization (topics in this case) is partitioned around the compute cluster based upon the join key (wordBlockID in this case), and hashed/sorted on each local machine using that key. The data table $X$ is then also partitioned around the cluster in the same way, so that the join $X \bowtie \Theta_i$ can be run by merging the local, already-sortedhashed subsets of $X$ and $\Theta_i$.

This looks a lot like a standard, distributed join algorithm. But a key requirement here is that we must avoid the need to re-order—or to even materialize—$X \bowtie \Theta_i$ to implement the subsequent co-group. The output of $X \bowtie \Theta_i$ must be pipelined directly into the merge that implements the co-group, so that the UDF can be parameterized with the join results as they are created. This presents a problem, because the join results are ordered based upon the join key (wordBlockID in our example) and the co-group requires an ordering based upon the seed.

Fortunately, it is possible to avoid the re-order by incorporating the join key (wordBlockID in our example) into the seed value generation, so that the result of the join is *already* ordered on the seed value. This is possible because the seed is an arbitrary value, used only to link parameters back to the data object that was used to generate the parameter set, and as long as it is unique, it can be generated using any arbitrary algorithm. Imagine that the seed is a 16-byte value. We could use the
upper-order eight bytes of the seed to store a hash of the wordBlockID, while the lower-order eight bytes are generated in such a way as to be unique for each seeded $x \in X$—the first four uniquely encode the compute core at which the seeding is taking place, and the last four bytes count the tuples seeded at the compute core.

Assume that we compute $X \bowtie \Theta_i$ by (1) partitioning $X$ across machines by hashing wordBlockID, (2) dividing by the number of machines to determine the identifier of the machine that each $x \in X$ will be mapped to, and then (3) sorting the local portion of $X$ at each machine by this hash. If seeds are generated as described above, we can obtain a functionally equivalent ordering by instated dividing the seed rather than the join key by the number of machines, and then sorting locally the seed. The resulting sorted and hashed $X$ can then be merged with $\Theta_i$, which has been hashed and sorted using the join key. But now, the output of the merge will already be ordered by the seed. This means that the output of the merge can be pipelined directly into the merge that implements the co-group, without any shuffling or re-ordering, and we avoid ever materializing $X \bowtie \Theta_i$.

**Limitations of the Global Implementation.** There are a few limitations of this approach that are worth mentioning.

First, in the case where a copy of each and every parameter set $\Theta_i$ can be stored in RAM locally at each machine, the local implementation will likely be far faster. Global performs a full shuffle of the data $X$ as well as a parameter set. This should be avoided if it all possible. But if a parameter set is large, it will make a lot of sense.

Second, it is unclear how to make global work in the case where there are multiple joins that produce very large parameterizations from multiple parameter sets, where each join uses a different key. Global relies on creating the seed in such a way that the join key ordering and the seed ordering are one-in-the-same. It is unclear how
this can be accomplished for two different join keys.

2.4 Optimization

It is unlikely that a well-designed local implementation of the join-and-co-group pattern is going to be outperformed by the naive implementation. The big difference between the two implementations is that local stores and re-uses UDF parameterizations in RAM, saving the CPU time associated both with computing the parameterization, and the system cost of moving data from the system and into the UDF. In the worst case, where there is no RAM available to save the parameterized UDF, local can be designed to effectively default to naive. No parameterized UDFs are saved, and so each and every UDF invocation must result in a re-parameterization. Our experiments will demonstrate the superiority of local over naive.

However, the question of when to use global remains. Global partitions the model—so there is one global copy—and moves the data to the model. This is potentially a terrible approach, requiring a re-partitioning of the entire data set, which may be debilitating. On the other hand, global is potentially the only possible way to run the computation if the model is large.

Fortunately, it is possible to make this choice automatically, with at least a bit of accuracy, using a cost-based query optimizer. The key question is whether the joins \( \Theta_1 \bowtie X, \Theta_2 \bowtie X, \ldots \) used to compute the parameterization of the model for each data point can guarantee that their output will be partitioned across machines and ordered by the seed value. If this is the case, then the co-group becomes computationally easy—requiring only a (typically pipelined) local merge of the output of each \( \Theta_1 \bowtie X, \Theta_2 \bowtie X, \ldots \), and local is going to be the preferred method since the result of the merge can be pipelined directly into the UDF invocation.
An individual join $\Theta_i \bowtie_{B_i} X$ is going to be ordered on the seed if the join does not require a re-ordering/re-partitioning of $X$. Recall that seeds are unique values, generated and associated with tuples in $X$ as $X$ is pipelined into the join-and-co-group. $X$ will come out of the join $\Theta_i \bowtie_{B_i} X$ still ordered on the seed, in two different cases:

1. If $X$ happens to be ordered on the join key encoded by the predicate $B_i$, so that the join $\Theta_i \bowtie_{B_i} X$ can be implemented by partitioning and ordering $\Theta_i$ on the join key, and then merging the two tables.

2. If $\Theta_i$ is small enough that a copy of it can be maintained at each node in the system, and so the join $\Theta_i \bowtie_{B_i} X$ can be implemented by pipelining the local portion of $X$ into a join with the local copy of $\Theta_i$.

Case (1) can be recognized during planning by making use of the “interesting sort order” information that has classically been maintained by query optimizers for decades [23]. For a join $\Theta_i \bowtie X$ that does not conform to case (1), the optimizer must estimate the size of $\Theta_i$—which is easy if $\Theta_i$ is materialized on disk, and possibly difficult if it is itself the result of a complicated computation—and add up the size of each non-conforming $\Theta_i$. If it is possible to buffer all of those in RAM, then local should be run. Otherwise, global should be chosen.

2.5 On a Dataflow Platform

In this section, we consider the implementation of the join-and-co-group pattern on Apache Spark [9], which is a dataflow platform. While the previous few sections went into great detail considering how join-and-co-group could be implemented within a
parallel database system, our focus in this section is how join-and-co-group could be implemented as an API on top of a data flow platform such as Spark.

2.5.1 Key Design Considerations

**Programmatic API.** For ease of use, the join-and-co-group pattern is implemented as a Scala API function `joinNCoGroup`. The model and data are provided as RDDs, and the interface is general enough to support the various machine learning computations described in the experimental section of the chapter. For the naive version of the `joinNCoGroup` function, a curried function with signature `g : Model => Data => Iterable[Delta]` is accepted as input. This function takes as input the model, producing a parameterized model that can then accept the data.

To implement the local and global approach, a slightly different API is required. Rather than the user applying a single curried UDF, we instead used two functions. The signatures of these functions are `g1 : Model => Object` and `g2 : (Object, Data) => Iterable[Delta]`. The function `g1` takes as an input the Model and creates a temporary object. This temporary object, when given to `g2` produces the same result as one would obtain by computing `g(Model, Data)` directly. This extension is required because Spark treats the UDF as a black box and depends on Scala for partial-application; it is unclear how to realize the two-step application under Spark. One could imagine applying Spark’s `map` transformation on an RDD storing the model, performing a partial application: `Model => (Data => Iterable[Delta])`, and then joining the result with the data. Though this approach will produce correct results, in practice it would be impossible to realize the performance benefits of the local approach compared to naive. This is because the partial application: `Model => (Data => Iterable[Delta])` is a syntactic sugar that simply stitches the `Model` and the
function $g$ together and only evaluates them when $\text{Data}$ is provided.

**Representing the data $X$ and the model $\Theta$.** Another consideration is how to represent data. In SimSQL, it is possible to implement mathematical programs using mathematical objects like vectors and matrices. Unlike SimSQL, Spark does not treat such objects as first-class citizens. Hence, instead of tables of vectors/matrices, the user is expected to implement the $\text{Model}$ and the $\text{Data}$ trait/interfaces himself or herself and provide the input as $\text{RDD}[\text{Model}]$ and $\text{RDD}[\text{Data}]$ respectively. To reduce the impact of serialization overhead, we use native Java arrays to implement both Model and Data objects.

We also explored implementing the data and model as $\text{DataFrame}$ rather than $\text{RDD}$, but ran into several issues. Since Spark 2.0 discontinued support for user defined types, we stored both Model and Data as serialized objects. This has no additional overhead when the model and data to be joined are on different executors. However, if both model and data are on the same executor, we pay the additional penalty of serialization and deserialization. Also, since SparkSQL does not natively support linear algebra, the Catalyst optimizer [38] is unable to infer the statistics of intermediate results and hence chooses a suboptimal plan for the join operation. By adding linear algebra extensions and incorporating the optimization strategies discussed in the section 2.4 into the Catalyst optimizer, the join-and-co-group pattern could be supported directly by the $\text{DataFrame}$.

**Type of “join”.** For models like linear regression and Gaussian mixture models, the map-side join outperformed alternative join implementations, such as Sort-Merge-Join (using SparkSQL), RDD’s join API and Cartesian-Product followed by filter. This is because the size of the model for linear regression and Gaussian mixture models is relatively small and can easily fit into the broadcast budget of our executors.
For LDA with 1250 topics, the resultant model fit into the broadcast memory and hence the map-side join outperformed alternative join implementations. We noticed that the output partition of UDF invocation (i.e. step 4 below) for LDA was much larger than the input partition. As an example, an average task took as an input partition of size 2.4 MB and generated an output partition of size 170 MB. This imbalance caused the executor JVM to thrash and we had to repartition the RDD before UDF invocation to avoid thrashing. Also, we increased the broadcast budget on each executor, reduced the storage memory fraction and removed any unnecessary \texttt{persist()} calls. We validated through Spark UI that RDDs that were persisted were never evicted to the disk.

However, for LDA with 5000 topics, the resultant model was of size 40 GB. Since the model was too large to broadcast or even fit into the driver memory, the map-side join failed. Both the Sort-Merge-Join (using SparkSQL) and Cartesian-Product followed by filter were too slow and we killed the job after 24 hours. Then, we tried to run LDA local implementation (described later) using RDD’s \texttt{join} API, which successfully joined the data and the model. In step 5 (described below), we apply \texttt{reduceByKey} to the output of the join. Spark uses \texttt{ExternalAppendOnlyMap} data structure to perform \texttt{reduceByKey}, which throws ‘GC overhead error’ due to huge amount of data. Moreover, as our LDA global implementation on Spark is based on customized model and data partitioning, joining the repartitioned model and data fails due to the 2GB frame size limit in Spark’s \texttt{TransportFrameDecoder} class.

2.5.2 Naive implementation

Some key details of our naive implementation of \texttt{joinNCoGroup} are as follows.

\textbf{Step 1: Seeding.} The seeding of the data RDD is implemented using \texttt{zipWithIndex}
API, which associates the seed with each data item. If the RDD has more than one partition (which is the case in our experiments), then this method triggers a Spark job.

**Step 2: Preparing the parameters.** After seeding, we compute the hash for every model and data item using the user-defined functions `model_hash: Model => Long` and `data_hash: Data => Long`. The RDD obtained after seeding the model is then collected to the driver node as an `Array`. This `Array` data structure is then sent to all nodes using Spark’s `broadcast` API. The implicit assumption of this approach is that the collection of all the `Model` objects requires less than half of the driver memory budget. This assumption holds for many machine learning models, but not for those where the learned model is very large. For larger models, one should use global approach.

**Step 3: Join-and-Co-Group.** We perform a map-side join using below Spark code:

```scala
val model = sc.broadcast(modelRDD.collect)
val joinedModelData = dataRDD.flatMap(
  d => model.filter(m => m.id == d.id)
  .map(m => d.id, m.model, d.data))
```

A map-side join ensures that the data RDD is not shuffled and is always processed locally. Though a map-side join may fail on a very large input (a large model, for example), a large model is better handled via the global implementation.

**Step 4: UDF Invocation.** The above step outputs `RDD[ID, Model, Data]`. We apply a `map` transformation on this RDD where we apply the function `g` to each value of the given RDD. We also output the ID, as well as the data item along with the output of the `g` function (type `Iterable[Delta]`). It is important to note that Spark
is able to pipeline Step 3 and Step 4 into single stage, thereby reducing the amount of data shuffled across the cluster.

Step 5: Output assembly. The output assembly is done by using RDD’s reduceByKey API. We aggregate the Delta associated with each data item and then perform aggregation using the user-defined function \((\text{Iterable}[\text{Delta}], \text{Data}) \Rightarrow \text{Output}\). Spark introduces an implicit combiner to aggregate the Delta values across the cluster. This helps in network communication by batching the Delta available in a given node together. As opposed to the groupByKey API, the reduceByKey API also avoids Out-Of-Memory errors which could occur in case of improper load balancing of reducers.

2.5.3 Local implementation

As intimated above, the local implementation takes as an input two user-defined functions \(g_1: \text{Model} \Rightarrow \text{Object}\) and \(g_2: (\text{Object}, \text{Data}) \Rightarrow \text{Iterable}[\text{Delta}]\) instead of a single \(g\) function. Unlike the naive implementation, after seeding we apply a map transformation using function \(g_1\) and then prepare the parameter and perform join-and-co-group similar to the naive implementation. However, we only apply \(g_2\) in UDF invocation step after the join. Assuming the number of datapoints is \(n_d\) and number of models is \(n_m\) and time to process \(g_1\) is \(t_{g_1}\), then we save \((n_d - n_m) \times t_{g_1}\) time units using the local implementation as opposed to naive implementation.

2.5.4 Global implementation

The global implementation is similar to the local implementation, except for two key differences. First, while preparing the parameters, both the model and data are partitioned using custom partitioner, which assigns the partition id using the formula: \(\text{hash mode} \text{numPartitions}\), where \(\text{hash}\) is computed using the user-defined
functions `model_hash` and `data_hash` respectively instead of using Java Object’s hash-Code. As an example, in LDA, both `model_hash` and `data_hash` output a word block identifier. Hence, after using the above mentioned partitioner, the model and data items corresponding to a given word block identifier are co-located on the same node. To ensure that this partitioning is not lost, we apply `persist` to the RDD using the `MEMORY_AND_DISK` storage level. Second, instead of using map-side join, we use RDD’s `join` API. This API first co-groups the two RDDs based on their keys and then performs a sort-based join. Since the model and data are already partitioned, no model or data item is shuffled over network while performing the sort, instead Spark performs a disk-based sort.

The key drawback of this approach is that degree of parallelism available to Spark is bounded by the cardinality of the hash functions. Since the hash functions are also treated as black-box, no inference about the cardinality of the hash functions can be made. If Spark SQL’s UDF framework (and similarly RDD’s `join` API) is extended to infer the cardinality of UDFs, Catalyst can be extended to choose the global implementation for a given machine learning workload instead of naive or local implementation.

2.6 Experiments

In this section, we benchmark our declarative implementation of the join-and-co-group pattern on top of SimSQL—a parallel relational database system. We also try to implement several applications of join-and-co-group on top of Spark, a popular dataflow platform.

**Experimental goals.** There are two major goals of our benchmark efforts. First, we wish to test the efficacy of the three different methods for implementing join-and-
co-group: naive, local, and global.

Second, we wish to see if there is value to the fully declarative approach compared to an implementation on top of a dataflow system. Specifically, in the introduction to the chapter, we asserted that it can be difficult to make a complicated computation such as join-and-co-group work (and work efficiently) on top of a dataflow system for two reasons. First, implementing join-and-co-group on a dataflow system requires a large number of difficult decisions on order of operations (maps, reduces, joins, etc.), data materialization and broadcast, and data representation. Second, even if the programmer can make the right choices, a dataflow system may not give a programmer enough control over low-level decisions to easily make the implementation efficient. If a dataflow-based join-and-co-group is more efficient than a declarative implementation, the declarative implementation may have value if it is simpler. If the dataflow-based join-and-co-group is not more efficient, it provides even stronger evidence for the declarative approach.†

2.6.1 Machine Learning Tasks

We considered three different distributed machine learning tasks over relatively large data sets in our experiments.

**Linear regression.** The data set that we used consisted of 4,135,219 Wikipedia documents. We use the 10,000 most common words in the corpus to create a feature

†It is worth noting that we do not experimentally compare the special-purpose, distributed LDA implementations (typically in C++ or Java) with our SimSQL-based, declarative LDA implementation. The reason is that a hand-coded, C++ version would undoubtedly be faster, in the same way that a C++ implementation of a database table scan is undoubtedly faster than an SQL database. But that would be an uninformative and potentially misleading comparison. Our goal is to provide a highly-performant, high-level, declarative interface to join-and-co-group that makes coding the pattern simple and flexible; the goal is not to provide the fastest implementation of join-and-co-group possible.
vector. The \(i\)th entry in feature vector \(x\) is the value of a log-likelihood (LLR) ratio statistic \([39]\) measuring the difference between the number of occurrences of the \(i\)th word in the dictionary, and the number of expected occurrences in the associated document. Let \(n\) be the total number of word occurrences in the corpus, \(m\) be the number of words in the current document, \(n_i\) be the number of occurrences of the \(i\)th dictionary word in the corpus, and \(m_i\) be the number of occurrences of this word in the document. So the frequency of word \(i\) in the corpus is \(p_i = n_i/n\) and the frequency in the document is \(p'_i = m_i/m\). Then:

\[
x_i = m_i(\log p_i - \log p'_i) + (m - n_i)[\log(1 - p_i) - \log(1 - p'_i)]
\]

Using feature vectors constructed in this way is challenging because the vectors are dense; each vector consists of 10,000 doubles, meaning the entire corpus is 330GB of dense feature vectors. The task is to learn a model (using gradient descent) to predict the last edit date of the document. This task is notable because of its simplicity, and the need to move a very large amount of data through the system quickly.

**Gaussian mixture model (GMM) imputation.** We again use the Wikipedia data set to extract 100 million 5-grams, using a 10,000 word dictionary. Each 5-gram can be treated as a point in a \(5 \times 10,000\) dimensional space. We use a random projection \([40]\) to map these points to a 100-dimensional space, and then censor 50\% of the entries in each 100-dimensional vector. The resulting data set is then represented as a set of 100 million records containing two, 100-dimensional vectors (160GB total). The first vector is the point, the second is the censored dimensions. The task is to use a MCMC algorithm to simultaneously impute the missing data, and cluster the data points. See \([7]\) for a description of the statistical model and algorithm. This task is
notable because parameterizing the model requires a linear algebra calculation that can be re-used across many different data points.

**Latent Dirichlet allocation (LDA).** For this task, we learned an LDA model over the 4,135,219 Wikipedia documents using a non-collapsed Gibbs sampler. However, we used a very large dictionary size (one million words) and a large model size (1,250 topics) and an even larger size (10,000 topics). Thus, storing the complete model—a matrix of word probabilities for each topic—requires $8 \times 1,250 \times 10^6 = 10\text{GB}$ for the smaller model, and $80\text{GB}$ for the larger model. Using the implementation outlined earlier in the chapter, we used 100 word blocks (of 10,000 dictionary entries each) to store each document. Empty word blocks were not stored. The total data set size was approximately $450\text{GB}$. Note that as described earlier in the chapter, we implement a non-collapsed Gibbs sampler as more interesting, theoretically justified—and difficult—example, which is different from most of the parallel LDA samplers described in the literature [41, 42, 34].

### 2.6.2 Join-and-Co-Group Specifics

**Linear regression.** Linear regression is a very simple computation. In the join-and-co-group, $X$ is the set of data points, and there is only one parameter “set” $\Theta_1$, consisting of the single regression vector. Hence, the join $\Theta_1 \bowtie \text{true } X$ is a simple cross product.

Both GMM imputation and LDA are rather complex computations and require multiple join-and-co-groups to implement, and so in our experiments, we focused on one particular join-and-co-group for each:

**GMM imputation.** For GMM imputation, we considered the imputation step,
which samples estimated values for each missing value in a data point, using the
mean and covariance of the Gaussian distribution that point has been assigned to,
conditioned on the observed dimensions. In the join-and-co-group pattern, $X$ consists
of (1) the identifier for a data point, and (2) the current assignment of a data point to
a cluster. $\Theta_1$ consists of the two vectors for each data point, and $\Theta_2$ is the Gaussian
model parameters (means and covariances for each cluster). Therefore, $\Theta_1 \Join_{B_1} X$
represents the data input joined on the data id, and $\Theta_2 \Join_{B_2} X$ the model input joined
on the identity of assigned Gaussian distribution that produced the data point.

**LDA.** As described previously, we benchmark the LDA implementation described
earlier in the chapter. We de-normalize all the wordProbs by wordBlockID. For
every word in a word block, the join-and-co-group that we evaluate takes in wordCnts
and topicProbs as data parameters, wordProbs as model parameter, and outputs
possible topicIDs that have generated that word (the AssignTopics UDF introduced
in Section 2.3.2 of the chapter). Here, $X$ and $\Theta_1$ stand for the de-normalized docs,
$\Theta_2$ the topicProbs of documents and $\Theta_3$ the de-normalized wordProbs. $\Theta_1 \Join_{B_1} X$
therefore gives the data input, $\Theta_2 \Join_{B_2} X$ gives the per-document topic distribution
and $\Theta_3 \Join_{B_3} X$ gives the per-topic word distribution joined on wordBlockID, serving
as the model input.

### 2.6.3 Parallel Database Results

For the database experiments, ten Amazon EC2 m2.4xlarge machines were used,
each of which had 68.4 gigabytes of RAM. All SimSQL UDFs were written in C++
using the GNU Scientific Library.

Results (wall clock run times) are shown in Figure 2.6. For LDA, both naive and
local fail to run to completion.
Figure 2.6: Total join-and-co-group time for parallel database system. Format is HH:MM:SS.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Naive</th>
<th>Local</th>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Reg.</td>
<td>00:05:05</td>
<td>00:04:17</td>
<td>04:22:01</td>
</tr>
<tr>
<td>GMM</td>
<td>00:55:03</td>
<td>00:16:24</td>
<td>01:32:24</td>
</tr>
<tr>
<td>LDA (Small Model)</td>
<td>Fail</td>
<td>Fail</td>
<td>01:10:19</td>
</tr>
<tr>
<td>LDA (Large Model)</td>
<td>Fail</td>
<td>Fail</td>
<td>05:33:13</td>
</tr>
</tbody>
</table>

We also show the breakdown of each total runtime into a series of pipelines in Figures 2.7, 2.8, 2.9, 2.10. For example, consider Figure 2.10, which shows how the global implementation is broken down into a series of computational steps by SimSQL. Seeding is performed as a distributed scan and write of the input document set $X$. The join of $X$ with $\Theta_1$, which attaches the seed to each word block, and the join of $X$ with $\Theta_2$, which attaches the topic distribution to each word block, are performed as distributed merge joins. The join of $\Theta_3$ and $X$, which adds the per-topic word distribution to each document, is pipelined into a distributed shuffle/sort of all of these results, on the seed value. Then, a fifth operation performs a distributed merge, whose results are pipelined into the UDF, and then into the join.

### 2.6.4 Spark Results

We spent a lot of time tuning Spark, and in the end, we built successful implementations of several of the join-and-co-group instances. Specifically, we attempted—and were successful—at building Spark implementations of join-and-co-group for logistic regression (local) and GMM imputation (local). We did not attempt global implementations of these algorithms, because our declarative implementations showed that the local implementations for both were clearly superior. We also successfully built a LDA local implementation for the smaller, 1,250 topic model. This implementation
Figure 2.7: Breakdown of SimSQL execution time; linear regression. The figure shows the grouping of operations into pipelines. The implementation constructs two pipelines for naive/local. Three are constructed for global (at right).

failed on the larger, 10,000 topic model, despite approximately one person-month of effort by two experienced Spark programmers. Likewise, we were unable to implement a working global join-and-co-group for either LDA problem. All tests were run on a cluster with the same configuration as the cluster used for SimSQL. The results are shown in Figure 12.

2.6.5 Discussion

As one might expect, there is little difference between naive and local for linear regression. Recall that local re-uses the a parameterized UDF. There is little cost associated with local processing of the parameters in a regression model learned via gradient descent, and so there is little advantage to local.

However, the advantage in GMM learning is significant. Processing the GMM model requires expensive matrix calculations that can be re-used across data points.
Figure 2.8: Breakdown of SimSQL execution time; GMM. The implementation constructs three pipelines for naive/local. Four are constructed for global (at right).

Figure 2.9: Breakdown of SimSQL execution time; LDA (smaller model). The implementation constructs five pipelines. Both naive/local fail.
Figure 2.10: Breakdown of SimSQL execution time; LDA (larger model). The implementation constructs five pipelines. Both naive/local fail.

As a result, we see nearly a $3.4 \times$ speedup for GMM local compared to GMM naive on SimSQL. Given that there is typically little additional cost associated with local compared to naive, any industrial-strength implementation of join-and-co-group should start with the local implementation.

LDA can only be run using global. The problem, as expected, is that the LDA model itself is many gigabytes in size. The join of the model with the data cannot be pipelined directly into the the UDF application due to model size. But running a fully distributed join of the model with the data results in a data set that is petabytes in size. The only option is to run global: partition a single copy of the model globally, and then move the data to the relevant portion of the model.

For smaller models, as expected, global is a terrible option. For a simple model such as linear regression, it takes $60 \times$ as long to run global compared to local (on SimSQL). It is interesting that the problem with global is made worse with simpler
<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of Items</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Reg.</td>
<td>$4.1 \times 10^6$ documents</td>
<td>328 GB</td>
</tr>
<tr>
<td>GMM</td>
<td>$10^8$ 5-grams</td>
<td>4 GB</td>
</tr>
<tr>
<td>LDA</td>
<td>$4.1 \times 10^6$ documents</td>
<td>40GB</td>
</tr>
</tbody>
</table>

Figure 2.11: Data set sizes.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Regression Local</td>
<td>03:00:24</td>
</tr>
<tr>
<td>GMM Local</td>
<td>00:56:04</td>
</tr>
<tr>
<td>LDA (Small Model) Local</td>
<td>03:34:22</td>
</tr>
<tr>
<td>LDA (Large Model) Local</td>
<td>Fail</td>
</tr>
<tr>
<td>LDA (Small Model) Global</td>
<td>Fail</td>
</tr>
<tr>
<td>LDA (Large Model) Global</td>
<td>Fail</td>
</tr>
</tbody>
</table>

Figure 2.12: Total running time for Spark (data flow platform).

models, because global works by bringing the data to the model. If the model is simple, consisting of only one component (as in linear regression) then that component is by definition mapped to one worker—destroying any opportunity to for parallelism.

One could perhaps imagine a scheme that gracefully transitions between local and global. For very large models, a full global implementation is used. For smaller and simpler models, local is run. For models of intermediate complexity, enough copies of the model are sent around the cluster that all available CPU cores are used to run the UDF. Such an implementation would be challenging, however, because it is unclear how to make such decisions at runtime, and it would be difficult to predict at compilation/optimization time the number of copies to distribute.

**Challenges of Join-And-Co-Group On Spark.** Despite a lot of effort, and a high-level of expertise in Spark, we had some difficulty getting join-and-co-group to run effectively on Spark, particularly in the case of LDA.
That said, we did have some moderate success getting LDA local to work on the smaller model, whereas our declarative implementation of LDA local failed. The declarative, SimSQL implementation of LDA local was not successful because SimSQL’s scan join implementation (which Spark would call a broadcast join) requires a separate copy of the broadcast table for each processor. Since the size of the smaller model is approximately 10GB, the 8 cores on each machine required 80GB of RAM, which the machines tested did not have, and so SimSQL failed.

With Spark, however, with 1,250 topics, the 10GB model is shared by all processors and hence it could fit into the broadcast memory. This makes a feasible map-side join feasible. However, we had a difficult time achieving reasonable performance. We noticed that the output partition of UDF invocation for LDA was much larger than the input partition. As an example, an average task took as an input partition of size 2.4MB and generated an output partition of size 170MB. This imbalance caused the executor JVM to thrash, and we had to repartition the data RDD into smaller RDDs before UDF invocation to avoid thrashing. During tuning, we also increased the broadcast budget on each executor, reduced the storage memory fraction and carefully tuned the persist() calls to reduce thrashing. Still, performance was inferior to SimSQL’s LDA global implementation.

As the LDA model size gets larger, it becomes impossible to fit the model into the driver memory and broadcast to the executors. Thus, for the larger LDA model, in the local implementation we used Spark’s join operation directly over model and data, instead of a map-side join. The join and the subsequent UDF invocation succeeded. However, the amount of intermediate data (the Deltas) generated by the UDF was huge—several terabytes. When Spark used the ExternalAppendOnlyMap data structure to apply reduceByKey to these Deltas, it threw a ‘GC overhead error’
due to the amount of data and we were unable to remedy this.

We were unable to make LDA global work for either model. As our LDA global implementation must make use of customized model and data partitioning (by \texttt{wordBlockID}), tuning was difficult. We were unable to repartition data in a way that Spark could process them (as in the LDA local implementation). Therefore, joining the relatively large repartitioned model and data failed due to the 2GB frame size limit in Spark’s \texttt{TransportFrameDecoder} class.

**Closing Thoughts.** Does all of this mean that Spark cannot possibly run join-and-co-group efficiently? Absolutely not—the specialized implementation of join-and-co-group described in this chapter could absolutely be ported to Spark, and in that case, it may very well outperform our SimSQL-based implementation. Or, it is entirely possible that there is a Spark-based implementation for join-and-co-group that we missed, that could outperform our SimSQL-based implementation right now. But we have shown that a careful implementation of join-and-co-group on a declarative system can export an interface that offers performance that is comparable to (and often superior to) a careful, expert implementation on top of Spark. Given the obvious benefits of a declarative approach, this seems to validate the ideas in this chapter.
Chapter 3

Declarative Recursive Computation on a RDBMS

3.1 Motivation

Most popular tools for building machine learning (ML) algorithms (such as TensorFlow [11], Caffe2 [43], and PyTorch [44]) are essentially numerical optimization engines bolted on top of a compute engine. In the ideal case, to use such a tool, the programmer specifies a model in the form of a loss function, and the tool generates a numerical optimization procedure (typically, gradient decent) by automatically differentiating the model. This optimization procedure is then run on top of the system’s compute engine.

**Parameter Servers for Machine Learning.** Most widely-used ML algorithm-generation tools (such as TensorFlow [11]) use a so-called parameter server [45, 46] as the underlying distributed compute engine for executing automatically-generated algorithms.

A parameter server is essentially a distributed key-value store. In a parameter server, a set of machines are designated as servers. Servers typically store fragments of the model, addressable via a unique key. The remainder of the machines are designated as workers. Workers repeatedly load a small portion of the training data, request some relevant part of the model, use the data to update the part of the model, and then push the updated model to the servers.

In the common case that the numerical optimization procedure to minimize the
loss function is generated automatically and the end user is more skilled in mathematics than in writing Big Data codes, it is desirable for the compute engine to run the numerical optimization procedure well, no matter how large or small the data set and model, and no matter how many machines are used to run the computation, or the amount of resources (CPU and RAM) available on each machine. To learn a model twice as quickly, just requisition twice the machines (and hence twice the CPUs/G-PUs) and run the same code. This is the “one implementation, any model/data size and compute hardware” ideal.

Unfortunately, this is not the reality for today’s distributed deep learning tools. It is true that a parameter server can be used to implement any distributed computation—just as a tool such as MPI [47] can be used to implement any distributed computation—but the problem is that getting many distributed computations to work on top of a parameter server will typically require a large amount of human effort. For anything other than relatively simple computations, a programmer must manually break the computation into discrete units and then make choices about which servers serve which data, which workers run which computations, which machines workers and servers are housed on, how to synchronize the servers and workers, etc. The result is that in practice, parameter servers such as TensorFlow are most often used to execute computations on a single machine. For distributed learning, parameter servers are typically used for running asynchronous, stochastic gradient descent algorithms [48] in the simple case where the model is small enough to be easily broadcast in its entirety to each worker. The various workers each have a small subset of the data, which they use to update the model, and then asynchronously (or synchronously) send their updates back to the parameter server. This is the so-called data parallel implementation. In practice data parallel implementations are preferred because they
are easy. Each processing unit performs the same computation over different subsets of the data.

In contrast, in many situations, a *model parallel* implementation—where various workers work on different parts of a huge model—would be desirable. Depending upon the application, the model may be so large that it cannot fit into the RAM of any particular machine and so model parallelism is required. Or, the model may be complicated and difficult to compute, and as we will argue, data parallelism is limited in its ability to speed convergence of distributed gradient descent algorithms. Model parallelism can help.

Unfortunately, various claims both in the folklore and in research papers, parameter servers *do not* support easy model parallelism, where “easy” model parallelism would insulate a programmer from the details of the distributed implementation, the same way in which SQL insulates a database programmer from the details of how a distributed query is executed. In the ideal case, the gradient descent algorithm automatically generated by the ML platform’s front end would automatically run well on any model/data size and compute hardware. However, this is far from the reality of the situation.

**Adapting RDBMS Technology for ML.** In contrast, the goal of “one implementation, any model/data size and compute hardware” coincides almost perfectly to the relational database management system (RDBMS) concept of *data independence*, which has been a guiding principle of RDBMS design since their inception in the 1970’s. To support data independence, databases are fundamentally based upon the declarative programming paradigm [19]: the programmer (or algorithm generator) specifies what he/she/it wants, and not how to compute it, and the computation is automatically optimized to match the data and hardware. We will argue that this
makes an RDBMS a great platform for distributed machine learning.

Besides the principle of data independence, RDBMSs have many other characteristics to recommend their use as a replacement for a parameter server as the compute platform for distributed ML computations. Databases are fast, robust, distributed query optimization is now well-understood and highly effective [49]. It is not an accident that competing distributed compute platforms such as Spark [9] (which now promotes the use of relational-style DataFrames [50] and DataSets [51] interfaces) are beginning to look more like a parallel RDBMSs.

Our central thesis in this chapter is that it makes little sense to reinvent the wheel—that ideas such as declarative programming and data independence have made RDBMSs so successful make a lot of sense for distributed machine learning as well. We will show that it is possible to express deep learning computations in a few lines of SQL, and have those computations automatically parallelized in a truly model-parallel way. We will argue that there is nothing unique about modern ML computations that make them particularly amenable to the key-value style of programming required by a parameter server. In fact, such computations look at lot like the classical analytics queries that have been successfully evaluated by distributed RDBMSs for the past two decades. Hence, tools such as TensorFlow should be using RDBMS-like backends, rather than those based upon parameter serves.

However, there are a couple of reasons that a modern RDBMS cannot be used out-of-the-box as a platform for most large-scale machine learning algorithms. Crucially, such systems lack sufficient support for recursion. In deep learning, for example, it is necessary to “loop” through the layers of a deep neural network, and then “loop” backwards through the network to propagate error and update the model. All of this is repeated within another “loop” that repeatedly runs the forward-backward
passes. Such “looping” could be expressed declaratively via recursive dependencies among tables, but RDBMS support for recursion is typically limited (if it exists at all) to computing fixed-points over sets—specifically with an eye towards computing transitive closures [52]. And crucially, even if one adds sufficient support for recursion to an RDBMS, there is the problem that the “query” plan for a typical deep-learning computation may run to tens of thousands of operators, which no existing RDBMS optimizer is going to be able to handle.

**Our Contributions.** In this chapter, we consider exactly how database support for recursion should be enhanced to handle large-scale machine learning computations, and then address the question of how a typical query optimization framework can be modified so that it can handle the massive query plans that result from complicated recursive computations. We implement our ideas on top of SimSQL, which is a prototype distributed RDBMS that is specifically designed to handle large-scale statistical computation.

For two distributed deep learning problems (a feed-forward neural network and an implementation of Word2Vec [53, 20]) we show that declarative SimSQL codes of the kind that could be auto-generated by a ML algorithm-generation tool outperform corresponding TensorFlow parameter server implementations, and trivially scale to huge model sizes that are beyond the capabilities of the implementations that ship with TensorFlow. This is despite the fact that SimSQL has the performance handicap of being a Java-based prototype RDBMS implemented on top of MapReduce. A high-performance RDBMS would likely do even better. We also show that a distributed LDA implemented on top of SimSQL generally scales better and outperforms the same implementation on top of TensorFlow’s parameter server (as well as Apache Spark).
Taken together, these results call into question the current practice of developing state-of-the-art machine learning platforms on top of a distributed parameter server, and instead suggest that a more RDBMS-like backend is perhaps a more suitable choice.

3.2 Parallelism in ML

In the general case, when solving a ML problem, we are given a data set $T$ with elements $t_j$. The goal is to learn a $d$-dimensional vector ($d \geq 1$) of model parameters $\Theta = (\Theta^{(1)}, \Theta^{(2)}, \ldots, \Theta^{(d)})$ that minimize a loss function of the form $\sum_j L(t_j|\Theta)$. To this end, most machine learning perform a simple update repeatedly until convergence:

$$\Theta_{i+1} \leftarrow \Theta_i - F(\Theta_i, T)$$

Here, $F$ is the update function. Each update marks the end of a processing epoch. Many learning algorithms are decomposable. That is, if $T$ has elements $t_j$, the algorithm can be written as:

$$\Theta_{i+1} \leftarrow \Theta_i - \sum_j F(\Theta_i, t_j)$$

For example, consider gradient descent, the quintessential learning algorithm. It is decomposable because $F(\Theta_i, T) = \sum_j \nabla L(t_j|\Theta_i)$.

If it is possible to store $\Theta_i$ in the RAM of each machine, decomposable learning algorithms are trivially made data parallel. One can simply broadcast $\Theta_i$ to each site, and then compute $F(\Theta_i, t_j)$ for data $t_j$ stored locally. All of these values are then aggregated using standard, distributed aggregation techniques.

However, data parallelism of this form is often ineffective. Let $T_i$ be a small sample of $T$ selected during epoch $i$. Since for decomposable algorithms, $F(\Theta_i, T) \approx$
\[
\text{\frac{|T_i|}{|T|} F(\Theta_i, T_i),}
\]
in practice only a small subsample of the data are used during each epoch (for example, in the case of gradient descent, *mini-batch gradient descent* is typically used). Adding more machines can either distribute this sample so that each machine gets a tiny amount of data (which is typically not helpful because for very small data sizes, the fixed costs associated with broadcasting \(\Theta_i\) dominate) or else use a larger sample (which is also not helpful because the estimate to \(F(\Theta_i, T)\) with a relatively small sample is already accurate enough).

One idea to overcome this is to use *asynchronous data parallelism* [54], where recursion of the form \(\Theta_{i+1} \leftarrow \Theta_i - F(\Theta_i, T)\) is no longer used. Rather, each site \(j\) is given a small sample \(T_j\) of \(T\); it requests the value \(\Theta_{\text{cur}}\), computes \(\Theta_{\text{new}} \leftarrow \Theta_{\text{cur}} - F(\Theta_{\text{new}}, T_j)\) and registers \(\Theta_{\text{new}}\) at a parameter server. Any subsequent requests for \(\Theta_{\text{cur}}\) will return this new value. All requests for \(\Theta_{\text{cur}}\) are asynchronous and happen to obtain whatever the last value written was, leading to stochastic behavior. To aide in convergence, it is possible to use policies that put limits on (among other things) the staleness of a synchronous update. The problem is that data parallelism of this form is often ineffective for more than a few machines. For a large cluster, either the policy regarding staleness is so restrictive as to throw out most of the updates, or most of the computation is done using stale data.

An alternative is *model parallelism*. In model parallelism, the idea is to stage \(F(\Theta_i, T)\) (or \(F(\Theta_i, T_i)\)) as a distributed computation without assuming that each site has access to all of \(\Theta_i\) (or \(T_i\)). There are many forms of model parallelism, but in the general case, model parallelism is “distributed computing complete”. That is, it is as hard as “solving” distributed computing. Existing systems for machine learning have very limited support for automated model parallelism. Further, the distributed key-value stores (or parameter servers) favored by most existing Big Data
ML systems (such as TensorFlow and Petuum [55]) seem to be a questionable choice for supporting automated model parallelism, at least compared to relational-style systems. As we will argue in this chapter, given that SQL-based relational systems have historically been among the most successful distributed computing systems, supporting implicit parallelism and automatic optimization. Further, typical forms of $F()$ map very nicely into distributed, relational computations. Given this, we will argue that relational-style systems demand serious consideration as the backend for distributed, model parallel machine learning.

### 3.3 Deep Learning on an RDBMS

We begin this section by describing the computation required to learn the weights within a standard deep neural network. We then describe how this maps to a relational-style computation, and point out some of the dangers associated with implementing such a computation on top of an existing RDBMS. This will serve as motivation for the technical ideas presented later in the chapter.

#### 3.3.1 A Simple Deep Learner

A deep neural network is nothing more than a differentiable, non-linear function, typically conceptualized as a directed graph. Each node in the graph (often called a “neuron”) computes a continuous activation function over its inputs (standard activation functions are sigmoid, ReLU, etc.). The neuron then sends the result of the computation over an edge in the graph to another neuron.

One of the simplest and most commonly used artificial neural networks is a so-called *feed-forward neural network* [56]. The graph associated with this type of computation is acyclic, and the neurons are organized into layers. Neurons in one layer
Figure 3.1: Structure of a feed-forward neural network.

are connected only to neurons in the next layer, hence the name “feed-forward”. Consider the feed-forward network in Figure 3.1. To compute a function over an input (such as a text document or an image), the input vector is fed into the first layer, and the output from that layer is fed through one or more hidden layers, until the output layer is reached. If the output of layer \( l - 1 \) (or “activation”) is represented as a vector \( a_{l-1} \), then the output of layer \( l \) is computed as:

\[
a_{l} = \sigma(a_{l-1}W_{l} + b_{l})
\]

Here, \( b_{l} \) is the bias vector associated with layer \( l \), \( W_{l} \) is the weight matrix associated with the layer, and \( \sigma(\cdot) \) is the activation function. If the goal is classification, a softmax filter is often added on the output layer to compute a weight (often viewed as a probability) associated with each of the possible output class labels.

**Learning.** Learning is the process of customizing the weights for a particular data set and task. Since learning is by far the most computationally intensive part of using
a deep network, and because the various data structures (such as the $W_l$ matrix) can be huge, this is the part we would typically like to distribute across machines.

Two-pass mini-batch gradient descent is the most common learning method used with such networks. Each iteration takes as input the current set of weight matrices $\{W_1^{(i)}, W_2^{(i)}, \ldots\}$ and bias vectors $\{b_1^{(i)}, b_2^{(i)}, \ldots\}$ and then outputs the next set of weight matrices $\{W_1^{(i+1)}, W_2^{(i+1)}, \ldots\}$ and bias vectors $\{b_1^{(i+1)}, b_2^{(i+1)}, \ldots\}$. This process is repeated until convergence.

In one iteration of the gradient descent, each batch of inputs goes through two passes: the forward pass and the backward pass.

**The forward pass.** In the forward pass, at iteration $i$, a small subset of the training data are randomly selected and stored in the matrix $X^{(i)}$. The activation matrix for each of these data points, $A_1$, is computed as $A_1^{(i)} = \sigma \left( X^{(i)} W_1^{(i)} + B_1^{(i)} \right)$ (here, let the bias matrix $B_1^{(i)}$ be the matrix formed by replicating the bias vector $b_1^{(i)}$ $n$ times, where $n$ is the size of the mini-batch). Then, this activation is pushed through the network by repeatedly performing the computation $A_l^{(i)} = \sigma \left( A_{l-1}^{(i)} W_l^{(i)} + B_l^{(i)} \right)$.

**The backward pass.** At the end of the forward pass, a loss (or error function) comparing the predicted set of values to the actual labels from the training data are computed. A common error function is the cross entropy between the predicted and actual labels. To update the weights and biases using gradient descent, the errors are fed back through the network, using the chain rule. Specifically, the errors back-propagated from hidden layer $l + 1$ to layer $l$ in the $i$-th backward pass is computed as

$$E_l^{(i)} = \left( E_{l+1}^{(i)} \left( W_{l+1}^{(i)} \right)^T \right) \odot \sigma' \left( A_l^{(i)} \right),$$

where $\sigma'(\cdot)$ is the derivative of the activation function. After we have obtained the
errors (that serve as the gradients) for each layer, we update the weights and biases:

\[ W_l^{(i)} = W_l^{(i-1)} - \alpha \cdot A_{l-1}^{(i-1)} E_l^{(i-1)}, \]

\[ b_l^{(i)} = b_l^{(i-1)} - \beta \cdot \sum_n e_l^{(i-1)}, \]

where \( \alpha \) and \( \beta \) are the learning rates, and \( e_l \) is the row vector of \( E_l \).

### 3.3.2 Deep Learning on an RDBMS

Perhaps surprisingly, a model parallel computation of this algorithm is easy on top of an RDBMS. We begin by assuming an RDBMS that has been lightly augmented to handle matrix and vector data types as described in [57], and assume that the various matrices and vectors have been “chunked”, so that (for example) we have the following database table to store the various weight matrices:

\( W \) (ITER, LAYER, ROW, COL, MAT)

Here, ITER and LAYER have the obvious meaning. MAT is of type matrix \((1000, 1000)\) and stores one “chunk” of the corresponding weight matrix. For example, a \(10^5 \times 10^5\) matrix chunked in this way would have \(10^4\) entries in the \( W \) table, with one sub-matrix for each of the \(100 = 10^5/10^3\) possible ROW values combined with each of the \(100 = 10^5/10^3\) possible COL values. Also, the activations and errors are stored in table A and E with similar schemas:

\( A \) (ITER, LAYER, COL, ACT)

\( E \) (ITER, LAYER, COL, ERR)

A final table AEW is used to store the errors that are back-propagated along each of the links for each of the data points:
AEW (ITER, LAYER, ROW, COL, ERR)

Given this, a fully model parallel implementation of the backward pass can be implemented using the SQL code in Figure 3.2. This code is surprisingly simple, implementing the matrix multiplications in distributed fashion using SQL SUM and GROUP BY. Assuming a careful implementation of the matrix package on the underlying RDBMS (including support for using GPUs if they are available) there is no reason in theory that this implementation could not approach the performance of carefully hand-crafted implementation of a data-parallel learning algorithm using a tool such as MPI. Further, the entire data parallel backward-pass code is around twenty lines long and could easily be generated by the front end of a learning tool such as TensorFlow.

In Figure 3.2, note that there is a looping over the hidden layers indexed by 1. For each value of 1, two things are done. First, the weight matrices of layer $1+1$ are updated using the joins of tables A, W and E; the SQL for this looks a lot like its math form. Second, the error matrices of layer $1$ are joined with A and W under the constraints that $A._{\text{LAYER}}=1$ and $W._{\text{LAYER}}=1$, for the purpose of preparing for the next value of 1. The SQL for error calculation is similar to its math form, except that it has to do a further aggregation (through SUM and GROUP BY) due to model parallelism.

3.3.3 So, What’s the Catch?

This is excellent in theory, but there is, however, a significant problem, related to the unavoidable inclusion of imperative control flow in this code.

In writing a loop, the programmer used a database table to pass program state from one iteration of the loop to the next. This is done by through the AEW table, which stores the error being back-propagated through each of the connections from layer $l+1$
-- backward pass (starting from the output layer)
SELECT W.LAYER, W.ROW, W.COL, A.ACT, E.ERR, W.MAT
BULK COLLECT INTO AEW
FROM A, W,
(SELECT A.COL,
crossentropyderiv(A.ACT, DO.VAL) AS ERR
FROM A, DATA_OUTPUT AS DO
WHERE A.LAYER=9) AS E
WHERE A.COL=W.ROW AND W.COL=E.COL
AND A.LAYER=8 AND W.LAYER=9
AND A.ITER=i AND W.ITER=i;

for l = 8, ..., 1:
SELECT i+1, LAYER, ROW, COL,
MAT - matmul(t(ACT), ERR) * 0.00000001
BULK COLLECT INTO W
FROM AEW
WHERE LAYER=l+1;

SELECT W.LAYER, W.ROW, W.COL, A.ACT, E.ERR, W.MAT
BULK COLLECT INTO AEW
FROM A, W,
(SELECT ROW AS COL,
SUM(matmul(ERR, t(MAT))
* reluderiv(ACT)) AS ERR
FROM AEW
WHERE LAYER=1
GROUP BY ROW) AS E
WHERE A.COL=W.ROW AND W.COL=E.COL
AND A.LAYER=l-1 AND W.LAYER=l;
AND A.ITER=i AND W.ITER=i;
end for

SELECT i+1, LAYER, ROW, COL,
MAT - matmul(t(ACT), ERR) * 0.00000001
BULK COLLECT INTO W
FROM AEW
WHERE LAYER=1;

Figure 3.2: SQL code to implement the backward pass for iteration $i$ of a feed-forward deep network with eight hidden layers.
to layer $l$ in the network, for each of the data points in the current learning batch. If there are 100,000 neurons in two adjacent layers in a fully-connected network and 100 data points in a batch, then there are $(100,000)^2$ such connections for each of the 100 data points, or $10^{12}$ values stored in all. If each of these are stored as a double-precision floating point value, this is 8TB of data that must be materialized. The amount of data to materialize could be even larger with a larger number of connections, or a larger batch size. Materializing an 8TB table is problematic; materializing an 80 or 800TB table may be debilitating.

Storing the set of per-connection errors is a very intuitive choice as a way to communicate among loops iterations, especially since the per-connection errors are subsequently aggregated in two ways (one to compute the new weights at a layer, and one to compute the new set of per-connection errors passed to the next layer). But forcing the system to materialize this table can result in a very inefficient computation. Further, this computation is needlessly inefficient, precisely because the per-connection errors will immediately be aggregated, and hence need not be materialized. This could be implemented by pipelining the computation creating the new data for the AEW table directly into the two subsequent aggregations, but this possibility has been lost when the programmer asked that the new data be BULK COLLECTed into AEW.

Note that this is not merely a case of a poor choice on the part of the programmer—in order to write a loop, state has to be passed from one iteration to another, and it is this state has made it impossible for the system to realize an ideal implementation.

How might this problem be addressed? One could imagine moving control flow from the code entirely—asking the programmer to write a single query for the entire back-propagation. Without control flow, no opportunities for realizing the “correct”
implementation will be lost. This, however will itself create two problems. First, it means that somehow a single massive query of perhaps 150 lines must be composed, with nesting levels dozens of levels deep. Such a query would not be human readable. And second, for a large, deep network, the resulting compilation and optimization problem could be orders of magnitude larger than what could be handled by current relational database systems.

In the next section of the chapter, we address the programmability problem: how can complex recursive computations be expressed succinctly in SQL? In subsequent sections of the chapter, we address the execution problem: how can the resulting, very large query plans be compiled and executed?

3.4 Extensions to SQL

In this section, we consider a couple of simple extensions to SQL that make it easy for a programmer (either a human or a deep learning tool chain) to declaratively specify recursive computations such as the back-propagation of the last section—thus side-stepping the need for a programmer to force the materialization of the state of the program.

3.4.1 The Extensions

We introduce these SQL extensions in the context of a classic introductory programming problem: implementing Pascal’s triangle, which recursively defines binomial coefficients. Specifically, the goal is to build a matrix such that the entry in row \( i \) and column \( j \) is \( \binom{i}{j} \) (or \( i \) choose \( j \)). The triangle is defined recursively so that for any integers \( i \geq 0 \) and \( j \in [1, i - 1] \), \( \binom{i}{j} = \binom{i-1}{j-1} + \binom{i-1}{j} \):
Our extended SQL allows for multiple versions of a database table; particular
versions are accessed via one or more array-style indices. For example, we can define
a database table storing the binomial coefficient \(^\binom{i}{j}\):

```
CREATE TABLE pascalsTri[0][0] (val) AS
SELECT val FROM VALUES (1);
```

The table `pascalsTri[0][0]` can now be queried like any other database table, and
various versions of the tables can be defined recursively. For example, we can define
all of the cases where \(j = i\) (the diagonal of the triangle) as:

```
CREATE TABLE pascalsTri[i:1...][i] (val) AS
SELECT * FROM pascalsTri[i-1][i-1]
```

And all of the cases where \(j = 0\) as:

```
CREATE TABLE pascalsTri[i:1...][0] (val) AS
SELECT * FROM pascalsTri[i-1][0]
```

Finally, we can fill in the rest of the cells in the triangle via one more recursive
relationship:

```
CREATE TABLE pascalsTri[i:2...][j:1...i-1](val) AS
SELECT pt1.val + pt2.val AS val
FROM pascalsTri[i-1][j-1] AS pt1,
pascalsTri[i-1][j] AS pt2;
```
Note that this differs quite a bit from classical, recursive SQL, where the goal is typically to compute a fix-point of a set. Here, there is no fix-point computation. In fact, this particular recurrence defines an infinite number of versions of the \texttt{pascalsTri} table. Since there can be an infinite number of such tables, these are materialized on-demand. A programmer can issue the query:

\begin{verbatim}
SELECT * FROM pascalsTri[56][23]
\end{verbatim}

In which case the system will unwind the recursion, writing the required computation as a single relational algebra statement. Since a programmer will often want to ask questions about multiple versions of a table at the same time (without having each one be computed separately) all of the required results can be computed at once:

\begin{verbatim}
EXECUTE ( 
    FOR j IN 0...50:
        SELECT * FROM pascalsTri[50][j]
)
\end{verbatim}

By definition, all of the queries/statements within an \texttt{EXECUTE} command are executed as part of the same query plan. Thus, this would be compiled into a single relational algebra statement that produces all 51 of the requested tables, under the constraint that each of those 51 tables must be materialized (without such a constraint, the resulting physical execution plan may pipeline one or more of those tables, so that they exist only ephemerally and cannot be returned as a query result). If a programmer wished to materialize all of these tables so that they could be used subsequently without re-computation, s/he could use:

\begin{verbatim}
EXECUTE ( 
    FOR j IN 0...50:
        MATERIALIZE pascalsTri[50][j]
)
\end{verbatim}

which materializes the tables for later use. Finally, we also introduce a multi-table \texttt{UNION} operator that merges the contents of multiple, recursively-defined tables. This makes it easy to define recursive relationships that span multiple tables. For example,
a series of tables storing the various Fibonacci numbers (where $Fib(i) = Fib(i - 1) + Fib(i - 2)$ and $Fib(1) = Fib(2) = 1$) can be defined as:

CREATE TABLE Fibonacci[i:0...1] (val) AS SELECT * FROM VALUES (1)

CREATE TABLE Fibonacci[i:2...] (val) AS SELECT SUM (VAL) FROM UNION Fibonacci[i-2...i-1]

In general, UNION can be used to combine various subsets of recursively defined tables. For example, one could refer to UNION pascalsTri[i:0...50][0...i] which would flatten the first 51 rows of Pascal’s triangle into a single multiset.

### 3.4.2 Learning Using Recursive SQL

Now, with our SQL extensions, we can rewrite the aforementioned forward-backward passes to eliminate imperative control flow. We begin by initializing the activations during the forward pass at each iteration of the learning with the input data:

CREATE TABLE A[i:0...][j:0](COL, ACT) AS SELECT DI.COL, DI.VAL FROM DATA_INPUT AS DI;

We then send the activation across the links in the network:

CREATE TABLE WI[i:0...][j:1...9](COL, VAL) AS SELECT W.COL, SUM(matmul(A.ACT, W.MAT)) FROM W[i][j] AS w, A[i][j-1] AS A WHERE W.ROW = A.COL GROUP BY W.COL;

Those links are then used to compute subsequent activations:

CREATE TABLE A[i:0...][j:1...8](COL, ACT) AS SELECT WI.COL, relu(WI.VAL + B.VEC) FROM WI[i][j] AS WI, B[i][j] AS B WHERE WI.COL = B.COL;

And finally used to perform the prediction:

CREATE TABLE A[i:0...][j:9](COL, ACT) AS SELECT WI.COL, softmax(WI.VAL + B.VEC) FROM WI[i][j] AS WI, B[i][j] AS B WHERE WI.COL = B.COL;

Then we begin the code for the backward pass with the initialization of the error:
CREATE TABLE E[i:0...][j:9](COL, ERR) AS
    SELECT A.COL, crossentropyderiv(A.ACT, DO.VAL)
    FROM A[i][j] AS A, DATA_OUTPUT AS DO;

At subsequent layers, the error is:
CREATE TABLE E[i:0...][j:1...8](COL, ERR) AS
    SELECT W.ROW, SUM(matmul(E.ERR, t(W.MAT))
    * reluderiv(A.ACT))
    FROM A[i][j] AS A, E[i][j+1] AS E,
    W[i][j+1] AS W
    WHERE A.COL = W.ROW AND W.COL = E.COL
    GROUP BY W.ROW;

Now we use the error to update the weights:
CREATE TABLE W[i:1...][j:1...9](ROW, COL, MAT) AS
    SELECT W.ROW, W.COL,
    W.MAT - matmul(t(A.ACT), E.ERR) * 0.00000001
    FROM W[i-1][j] AS W, E[i-1][j] AS E,
    A[i-1][j-1] AS A
    WHERE A.COL = W.ROW AND W.COL = E.COL;

And the biases:
CREATE TABLE B[i:1...][j:1...9](COL, VEC) AS
    SELECT B.COL,
    B.VEC - reducebyrow(E.ERR) * 0.00000001
    FROM B[i-1][j] AS B, E[i-1][j] AS E
    WHERE B.COL = E.COL;

We now have a fully declarative implementation of neural network learning.

3.5 Handling Very Large Query Plans

The recursive specifications of the last section address the problem of how to succinctly specify complicated recursive computations so as to avoid any control flow that may force bad decisions upon the underlying system. Yet the question remains: How can the very complex computations associated with such specifications be executed?

To demonstrate the very high complexity of such a computation, consider the feed-forward network learner given in Section 3.3, and the SQL code:
EXECUTE (  
    FOR i IN 0...99:  
        EXPORT (ERRORS [i][9]) TO 'errors' + i + '.csv'
    FOR l in 1...8:  
        MATERIALIZE (WEIGHTS[99][l])
)  

As a result of this computation, 100 iterations of training will be completed. Each of 
the prediction errors incurred during those 100 training iterations will be written to a 
text file. Also, the complete set of weights learned at the end of the 100 iterations will 
be saved to the database, so that they can be used to make predictions, or improved 
upon by subsequent training.

The obvious way to execute this computation would be to start with the 108 
different tables that are requested by this execution, and then completely unroll the 
recursion, computing the complete set of tables that would need to be computed 
in order to supply the 108 user-requested tables. The complete set of dependencies 
between those tables could then be compiled into a relational algebra computation, 
and then optimized and executed using a (mostly) standard, RDBMS-like engine.

The problem with this is that the resulting graph would be huge. Each of the 
100 training iterations for an eight-layer network consists of 584 relational algebra 
operations, including 56 joins and 15 aggregations. Thus, 100 training iterations 
results in a query plan having 56,900 operations in all. No existing RDBMS engine 
could possibly optimize and execute such a complex computation. Analyzing and 
optimizing the plan itself is a particular bottleneck. Many different approaches to 
built query optimizers have been suggested over the years, but all of them assume 
a plan that is at most a few dozen operations in size. None of them are meant to 
cale into plans consisting of thousands of operations. Further, the situation here is 
unique in that the application is very unforgiving. Doing the wrong thing (such as
deciding to materialize the errors being back-propagated through the links in a deep net) can very easily result in a computation plan that can bring down the system.

**Frame-based execution.** To deal with this, an obvious idea is to partition the query plan into *frames*, and to optimize and execute those frames independently. A frame is simply a small sub-computation of the entire unrolled recursive computation. To execute the overall plan, one frame is optimized and executed, and its output materialized. Then a second frame is optimized and executed, and its output materialized. Frames are only optimized and executed once all of their inputs have been materialized. This is repeated until the entire computation is complete.

Frame-based computation is an attractive option because as long as each frame is small enough that an existing query optimizer and execution engine handle the frame, the RDBMS optimizer and engine need not be modified in any way. Further, this iterative execution results in an engine that resembles engines that perform re-optimization during runtime [58], in the sense that frames are optimized and executed only once all of their inputs have been materialized. Accurate statistics can be collected on those inputs, meaning that classical problems associated with errors propagating up a query plan can be avoided.

However, the problem with this approach is that cutting the computation into frames necessarily commits the system to a certain set of choices regarding the implementation and execution of the plan. Specifically, a cut across a long between one operation and the next means that the data flowing across that operation must be materialized. Hence, in a sense we are back to where we started in Section 3.3 of the chapter, where the iteration-based computation forced a programmer to materialize data in order to communicate between iterations, and the amount of data to materialize was debilitating. Here, we are again forced to materialize data in order
to communicate between frames. A poor choice of cut can bring us back to where we were in Section 3.3, where it does not matter how well we optimize and execute each frame, because the cost of moving data between frames is debilitating.

Hence, in the remainder of the chapter, we turn our attention to a central question: How can we efficiently cut a very large, unrolled computation into a set of frames, in such a way that the overall execution time is not significantly greater than had we somehow been able to optimize and execute the entire computation as a whole?

### 3.6 Problem Definition

In this section, we define the problem of computing the best cut of a very large plan into a set of frames.

#### 3.6.1 Intuition

The cost incurred when breaking a large plan into frames comes from the fact that the contents of already-executed frames have to be communicated to the frames that are utilizing them. To mitigate this cost, we would like our intermediate results to be as small as possible. Therefore splitting the plan so that the contents of small tables are saved would be desirable. Unfortunately, it is unreasonable to base the problem of cutting a plan on any notion of table size; in a very large plan, we are unlikely to have reliable statistics over intermediate results, as estimation errors propagate through the plan.

Instead, we choose to split the plan into frames by associating costs with operator-operator pairs whose link has been cut. For example, we seek to reduce the number of *pipeline breakers* induced. An induced pipeline breaker is one that would not have been present in an optimal physical plan, but was forced by the cut. For example,
cutting across the output from a selection operation induces a pipeline breaker—selection operations can always be pipelined. We discuss issues related to developing costs later in this section.

There is a secondary goal as well. If the cut results in a large number of very small frames, it will tend to restrict the ability of the system’s logical and physical optimizer to find optimization opportunities. For example, if the logical plan \(((R \bowtie S) \bowtie T)\) is optimal but the input plan \(((R \bowtie T) \bowtie S)\) is cut into frames \(f_1 = (R \bowtie T)\) and \(f_2 = (f_1 \bowtie S)\) it is impossible to realize this optimal plan. Thus, we wish to avoid having a large number of small frames.

### 3.6.2 Cost-Based Cutting

We assume that a query plan to be cut into frames is represented as a directed graph \(G = (V, E)\). If a query plan has \(N = |V|\) operators and \(D\) is the desired number of operators per frame, we would like to split the graph into \(M = \left\lfloor \frac{N}{D} \right\rfloor\) frames. In order to simplify the formulation, we define the split of a query plan to be a matrix \(X = (x_{ij})_{M \times N}\), where each row would be one frame so that \(x_{ij} = 1\) if operator \(j\) is in frame \(i\), and 0 otherwise.

Define \(c_{ij}\) to be the cost of operator \(j\) being in frame \(i\). Then we want to minimize the value \(\text{cost}(X) = \sum_{i=1}^{M} \sum_{j=1}^{N} x_{ij} c_{ij}\).

One question is: how do we define \(c_{ij}\)? We begin by defining \(\text{opCost}(o_i, o_k) = \frac{b(o_i, o_k)}{e^{T(o_i) - 1}}\), where \(b(o_i, o_k)\) is the base cost associated with operator \(o_i\) being in a different frame from operator \(o_j\), and \(T(o_i)\) counts the number of operations that consume the output of \(o_i\). We will discuss possible realizations of the function \(b\) subsequently. The exponential is introduced to lower the cost of operators with multiple consumers since their output cannot be fully pipelined.
The total cost \( c_{ij} \) of putting operator \( j \) in frame \( i \) then sums \( \text{opCost}(o_j, o_k) \) for each \( o_k \) consuming the output of \( o_j \):
\[
c_{ij} = \sum_{k=1}^{N} \text{opCost}(o_j, o_k) \cdot e(o_j, o_k) \cdot r(i, o_k),
\]
where \( e(o_j, o_k) \) is 1 if \((j, k) \in E\) and 0 otherwise, and \( r(i, o_k) \) is 0 if operator \( o_k \) is in cut \( i \), 1 otherwise.

Substituting, this gives us:
\[
\text{cost}(x) = M \sum_{i=1}^{M} N \sum_{j=1}^{N} x_{ij} \sum_{k=1}^{N} \text{opCost}(o_j, o_k) \cdot e(o_j, o_k) \cdot r(i, o_k)
\]
The term \( b_{ik} \) can be expressed as \( r(i, o_k) = 1 - x_{ik} \) since \( x_{ij} \) will be 0 if operator \( o_j \) is not in frame \( i \). Thus:
\[
\text{cost}(X) = \sum_{i=1}^{M} N \sum_{j=1}^{N} x_{ij} \sum_{k=1}^{N} \text{opCost}(o_j, o_k) \cdot e(o_j, o_k) \cdot (1 - x_{ik})
\]
\[
= \sum_{i=1}^{M} N \sum_{j=1}^{N} x_{ij} \sum_{k=1}^{N} \text{opCost}(o_j, o_k) \cdot e(o_j, o_k)
\]
\[
- \sum_{i=1}^{M} \sum_{j=1}^{N} \sum_{k=1}^{N} x_{ij} \cdot x_{ik} \cdot \text{opCost}(o_j, o_k) \cdot e(o_j, o_k)
\]
Then substitute \( d(o_j, o_k) = -\text{opCost}(o_j, o_k) \cdot e(o_j, o_k) \) and
\[
t(o_j) = \sum_{k=1}^{N} \text{opCost}(o_j, o_k) \cdot e(o_j, o_k):
\]
\[
\text{cost}(X) = \sum_{i=1}^{M} N \sum_{j=1}^{N} x_{ij} \cdot t(o_j) + \sum_{i=1}^{M} \sum_{j=1}^{N} \sum_{k=1}^{N} x_{ij} \cdot x_{ik} \cdot d(o_j, o_k)
\]
If we define the function \( f(i, o_h) \) so that \( f(i, o_h) = 1 \) if \( i = h \) and \( f(i, o_h) = 0 \) if \( i \neq n \),
we can add the sum \( \sum_{h=1}^{M} \) to the quadratic component, giving \( \text{cost}(X) = \)
\[
\sum_{i=1}^{M} \sum_{j=1}^{N} x_{ij} t(o_j) + \sum_{i=1}^{M} \sum_{j=1}^{N} \sum_{h=1}^{M} \sum_{k=1}^{N} x_{ij} f(i, o_h) x_{hk} d(o_j, o_k)
\]

The trivial solution to the minimization of the previous equation is to put all operators in the same frame, but that would result in a query plan that is not split. Therefore we need to add a constraint on the upper bound of operators on each frame. The following equation models that constraint:

\[
\forall j \in \{1, \ldots, M\}, \left( \sum_{ij=1}^{N} x_{ij} \leq D \right)
\]

Where \(D\) is the desired number of operators per frame. Another problem is that minimizing the value could result in a sequence of frames that may not be executable because they contain circular dependencies. In order to ensure that we have no circular dependencies, we have to make the intermediate value that a frame uses available before it is executed. Thus for a given operator \(o_j\) in frame \(i\) we introduce the following constraint:

\[
x_{ji} = 1 \Rightarrow \forall (k, j) \in R, \exists m \in \{1, 2, \ldots, i\}, \text{s.t.} x_{km} = 1
\]

Aside from the constraint that there cannot be circular dependencies among constraints, this is an instance of the generalized quadratic assignment problem [59].

### 3.7 Heuristic Solution

Generalized quadratic assignment is a very difficult problem [60]. Some even report exact solution procedure completed in a few days of running time [61]. Therefore, we turn to heuristics that may work for our application. There exist previous efforts
on solving the generalized quadratic assignment problem in seconds with various heuristics [59, 62, 63, 64].

One simple idea for our application is a greedy algorithm that builds frames, one at a time. Given a graph, the greedy algorithm starts from a source operator and adds operators to the frame iteratively, always adding the operator that directly depends on an operator in the current frame that would yield the smallest increase in the cost of the whole frame (where the cost is the label of each link which is cut to separate the frame from the graph). In order to ensure that we do not build frames that have circular dependencies, when we add a new operation \( o_i \) to the frame where \((o_i, o_j) \in E \) but \( o_j \) is not yet part of any frame, we add \( o_j \) to the frame. This may in turn trigger the recursive addition of new operators to the frame.

An illustration of how the algorithm works is given in Figure 3.3(a). We begin with operation \( o_1 \), adding operation \( o_2 \). Then we add \( o_4 \) since it has a lower cost than \( o_3 \). Since \( o_4 \) requires that we have computed \( o_6 \), we add \( o_6 \) and all of its un-added dependencies to the frame.

There are some problems with this algorithm. First, it is highly dependent on the chosen starting point; choosing a bad start can lead to a poor cut. Consider Figure 3.3(b). Operation \( o_{k+4} \) is near the end of a very long computation. If we choose this operation to start with, we will next add \( o_{k+5} \) which will cause the entire query plan to be recursively added into the frame. This makes it impossible to hit the target number of operators in the frame, \( D \).

We can remedy this by running the greedy algorithm repeatedly, starting with each possible operation. For each run, we stop growing the frame when its size meets or exceeds \( D \). We then choose the frame with the lowest possible cost.

In practice, we find that while re-running the greedy algorithm many times allows
(a) The greedy algorithm. The values in red represent the costs. The operators in green are selected as part of the frame. The operators in yellow are under consideration.

(b) Greedy if the source operator is badly chosen. Adding the source operator adds the rest of the graph to the frame.

(c) Greedy where the upper bound is set to $D = 7$.

(d) Trying three different frame sizes. The middle frame will be selected.

Figure 3.3: Greedily cutting a frame from a compute plan.
us to choose a suitably-sized frame, the problem definition itself can have issues. Consider Figure 3.3(c). If we choose $D = 7$, no matter where we begin the greedy algorithm, we will end up with the frame depicted in this figure, where we cut across two different operations. It would probably be better to simply include all operations in the frame.

Thus, we use both a lower bound on frame size and an upper bound. In practice, we use quite a wide range (a lower bound of ten operations and an upper bound of 100). To construct the next frame, for each possible starting operation, we greedily grow the frame until its size exceeds the lower bound, and keep growing the frame until the size exceeds the upper bound. We then choose the best (lowest cost) frame out of all of the various frames that were observed. This is illustrated in Figure 3.3(d), with a lower bound of three and an upper bound of five. In this case, the frame of size four is chosen.

3.7.1 Choosing Costs

Using trial and error, over time we have found a set of costs that tend to work well for our platform. For example, the cost of placing a frame boundary between an aggregation operation and any other operation is set to be zero. Aggregation results tend to be small (aggregations reduce data size) and so we want to encourage cuts across aggregation results. On the other hand, placing a frame boundary after a selection tends to have a very high cost. Data are always pipelined directly through selections, and so placing a frame boundary after a selection has a high cost since it forces a write of a potentially large data set that would not have been written otherwise. If the operation coming after the selection is an aggregate, we give a cost of two; this is the highest possible cost we use, assigned to this cut because we far
prefer to place the cut after the aggregation (forcing a write of a small result) rather than after the selection. If the next operation is a join, we give a cost of 1.5. A join result may be large, so it may be an error to lose the opportunity to pipeline its output—so choosing to cur across the selection (rather than the join) is not such an indefensible choice.

3.8 Experiments

In this section, we detail set of experiments aimed at answering the following questions:

*Can the ideas described in this chapter be used to re-purpose an RDBMS so that it can be used to implement recursive, model parallel machine learning computations? And, if so, are the resulting computations reasonably performant, and do they scale to large and complex models?*

To answer these questions, we implement the ideas in this chapter on top of the SimSQL distributed database system [36]. Though SimSQL is a research prototype system, it has a cost-based optimizer, an assortment of implementations of the standard relational operations, the ability to pipeline those operations, and the ability to make use of “interesting” sort orders. SimSQL also has some enhancements specifically for machine learning, such as native matrix and vector support [57].

As we will describe in detail subsequently, our benchmarking utilizes model-parallel, distributed implementations of three machine learning algorithms: (1) a multi-layer feed-forward neural network, (2) the Word2Vec algorithm [20] for learning embeddings of text into a high-dimensional space, and (3) a distributed, collapsed Gibbs sampler for LDA [5] (LDA is a standard text mining model).
3.8.1 Overview

In our benchmarking, we run three different sets of experiments. The experiments are designed to evaluate:

(1) **Utility of Plan Cutting.** One of the core technical ideas in this chapter is that frame-based execution can be used to scale a standard RDBMS optimizer and query execution engine to very large computational plan sizes. In this set of experiments, we use the two neural learning algorithms (feed-forward and Word2Vec) to evaluate the various algorithms we have proposed for cutting a very large computational plan into frames.

(2) **Scalability with Respect to Model Size.** In order to provide some idea of the relative scalability of our model parallel SimSQL implementations, for each algorithm, we experiment with a number of different model sizes, including models that are too large to fit into the RAM of any of the machines in our compute cluster. The goal is to determine how an RDBMS-based, model parallel implementation scales with increasing model size.

For the first two neural learners, we compare against the standard, data parallel feed-forward and Word2Vec implementations that are shipped with TensorFlow. For the collapsed LDA sampler, we compare with bespoke implementations on top of TensorFlow and Spark (custom implementations are required because TensorFlow ships with no LDA implementation, and Spark’s MLlib LDA implementation relies on variational inference rather than Gibbs sampling).

We stress that this is not meant to be a “which system is faster?” comparison. SimSQL is implemented in Java and runs on top of Hadoop, with all of the high fixed costs that implies. TensorFlow, on the other hand, is implemented natively in C++,
does not use Hadoop, and for that reason alone is likely to be considerably faster than SimSQL, at least for learning smaller models (when SimSQL’s high fixed costs will dominate). Rather than determining which system is faster, the specific goal is to study how an RDBMS-based, data-parallel learner implemented declaratively scales with model size, and to compare the relative scaling with TensorFlow and Spark.

(3) Scalability with Respect to Cluster Size. Finally, for fixed (but very large) model size, we evaluate how an RDBMS-based, data-parallel learner is able to gain performance by simply adding additional compute resources in the form of additional machines in a cluster.

3.8.2 Learning Algorithms

In this subsection, we describe the three different learning algorithms used in the benchmarking.

(1) A Feed-Forward Neural Network. This is the learning algorithm that we have used as a running example throughout the chapter. This is the most difficult algorithm to make data parallel due to the dependencies created by the fully-connected layers. Our RDBMS-based implementation has already been described extensively. We use the data parallel, synchronous, feed-forward network implementation that ships with TensorFlow as a comparison point.

We use a Wikipedia dump of 4.86 million documents as the input to the feed-forward learner. The goal is to learn how to predict the year of the last edit to the article. There are 17 possible labels in total. We pre-process the Wikipedia dump, representing each document as a 60,000-dimensional feature vector, where each feature corresponds to the number of times a particular unigram or bigram appears in the document. The network learned has two fully-connected hidden layers.
(2) **Word2Vec.** Word2Vec is a two-layer neural network used to generate word embeddings such that words embedded close to one another are somehow similar. We use the skip-gram version of Word2Vec. To decrease the number of model parameters to train, we use negative sampling, with 64 negative samples per document. The loss function used is the average noise contrastive estimation (NCE) loss. We train our Word2Vec model using the same Wikipedia dump described above, embedding the 1 million most frequent tokens in the corpus. The input and output layers in our experiments both have one million neurons. The neurons of the input layer are connected to the neurons of an intermediate embedding layer, which are further connected to the neurons of the output layer. Therefore, there are two weight matrices of size $V \times D$ and size $D \times V$ respectively, where $V$ is one million in our experiments, and $D$ is the dimension of embeddings used. For each batch, one document is randomly selected and pre-processed with skip window size of 1. For example, a sentence of “this hidden layer has ten neurons” will be converted to (hidden, this), (hidden, layer), (layer, hidden), (layer, has) and so on. On average, each batch has 1240 such word pairs.

Briefly, our Word2Vec SimSQL implementation works as follows. We have three recursive schemas. First, for aforementioned weight matrices we have $\text{weights}[i:0\ldots][j:1\ldots2]$ with attributes tokenID and $\text{embedVec}$. By storing the embedding of each token as a vector, we automatically have a model parallel representation. Second, $\text{embeds}[i:0\ldots][j:1\ldots3]$ stands for the embedding vectors from the weight matrices, where $j = 1$ gives the embeddings corresponding to input labels in a batch, $j = 2$ gives those corresponding to output labels, and $j = 3$ gives those corresponding to what we call “negative samples” from a Zipfian distribution via a user-defined function. Moreover, $\text{errors}[i:0\ldots][j:1\ldots2]$ represents the delta updates to be applied back to $\text{weights}[i][j]$. Both $\text{errors}[i][1]$ and $\text{errors}[i][2]$ are cal-
culated from $\text{embeds}[i][1]$, $\text{embeds}[i][2]$ as well as $\text{embeds}[i][3]$.

We compare our SimSQL implementation with the Word2Vec implementation that ships with TensorFlow. TensorFlow’s implementation is “lightly” model parallel, in that while data are partitioned across workers in a data parallel way, each worker requests only the portion of the model relevant to the data it is processing.

(3) Latent Dirichlet Allocation. LDA is a standard text mining algorithm, and collapsed Gibbs sampling is a standard learning algorithm for LDA. The goal when learning LDA is to learn a set of topics, which are sets of words that tend to co-occur with one another. Collapsed LDA requires maintaining counts of (1) the number of words assigned to each topic in a document, and (2) the number of words assigned to each topic in the corpus. Workers must repeatedly load a document, then cycle through the words in the document, re-assigning them to topics, updating both sets of counts. In distributed LDA, since local updates change the global topic counts—and these updates cannot be distributed globally in an efficient manner—the effect of local updates is typically ignored [45] until a synchronization step. In our LDA implementation, we divide all input documents into ten different subsets. All of the documents in the subset are processed, and then in a synchronized aggregation, the number of words assigned to each topic is updated.

In SimSQL, this is implemented by annotating the documents with a partitionID. To create the table $\text{WORDS_TO_TOPICS}[i]$ with attributes docID, wordID, topicID, and cnt, the query $\text{SELECT} * \text{ FROM docs WHERE partitionID} = i \text{ MOD } 10$ is used to select only the documents associated with the partition to be updated in the current iteration. Then, each of the words in all of the documents in partition $i \text{ MOD } 10$ are re-assigned to topics. To reference the complete set of assignments at the beginning of iteration $i$, the statement $\text{UNION WORDS_TO_TOPICS}[i-10...i-1]$ is used.
So, for example, to obtain the complete set of word-to-topic counts, our SimSQL implementation uses the SQL:

```sql
SELECT topicID, SUM(cnt)
FROM UNION WORDS_TO_TOPICS[i-10...i-1]
GROUP BY docID
```

We build an analogous model parallel implementation using Spark RDDs. We define classes `Document` and `TopicModel`. Each `Document` instance consists of the topic label for each word in a document, as well as a vector of the accumulated per-document topic counts. A `TopicModel` consists of a dense vector of the accumulated per-topic word counts, as well as an array of sparse vectors representing the counts for a `(topicID, wordID)` pair. At the beginning, the input data is parallelized as RDD[Document] and partitioned into ten RDDs with the `filter` method. Also, one instance of `TopicModel` is created on the driver. For each of the partitions, the `TopicModel` instance is broadcast to all the executors from the driver before being used to sample new topic labels for the words. After sampling, the changes in topic labels for that RDD partition are updated to the driver copy of `TopicModel`, which is further used in the next RDD partition.

TensorFlow's implementation is again “lightly” model parallel, in that while data are partitioned, requests to the parameter server pull only the required portion of the model. The topic-word counts (ntw) are stored on the parameter server as a matrix tensor. Due to the lack of support for indexing sparse variables, we choose to use a dense matrix tensor here, which can overfill the memory when the number of topics gets really large: `ntw = tf.get_variable("ntw", [V, T],
initializer=tf.constant_initializer(0.0), ...)`. The topic labels for all the words in that document are stored on each worker locally in Python dictionaries and are refreshed after each iteration. The topic sampling process goes by looping over
each word in a document with `tf.while_loop`. Since each document is of variable length, we store the sampled topics in a dynamic-sized `tf.TensorArray` passed within the `tf.while_loop`. The changes in sampled topics are updated to `ntw` on parameter server via `tf.scatter_add`. After each partition of documents is processed, barriers are added on each worker via `tf.FIFOQueue` for synchronization purpose.

To give the reader an idea of the relative complexity of coding for these systems, in Figure 3.4 we give line counts for each of the various implementations. Note that SimSQL’s implementation is broken into lines of SQL code, and into lines of C++ code necessary to implement user-defined functions.

### 3.8.3 Results

Most experiments were run on a cluster of ten Amazon EC2 `m2.4xlarge` machines, each with eight cores and 68GB of RAM.

To test the utility of plan cutting, we run our SimSQL feed-forward network implementation and our Word2Vec implementation using four different options for cutting the plan into frames. In each case, hidden layers containing 40,000 neurons are

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Feed-Forward NN</th>
<th>Word2Vec</th>
<th>LDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimSQL SQL</td>
<td>206</td>
<td>140</td>
<td>135</td>
</tr>
<tr>
<td>SimSQL C++</td>
<td>324</td>
<td>334</td>
<td>641</td>
</tr>
<tr>
<td>SimSQL total</td>
<td>530</td>
<td>474</td>
<td>776</td>
</tr>
<tr>
<td>TensorFlow total</td>
<td>331*</td>
<td>196*</td>
<td>227</td>
</tr>
<tr>
<td>Spark total</td>
<td>NA</td>
<td>NA</td>
<td>424</td>
</tr>
</tbody>
</table>
used. In the first two cases, we run the simple greedy and the full graph cut algorithms with all operator-to-operator costs each set to one, so the goal is to minimize the size of the cut. In the second two cases, we run the simple greedy and the full graph cut algorithms with operator-to-operator costs tuned as described in Section 3.6 of the chapter. Figure 3.5 shows the per-iteration running time for each of these options.

In the remainder of the experiments, the full graph cut algorithm with tuned operator-to-operator costs was used. To test scalability with respect to model size, we first run SimSQL and TensorFlow to learn a feed-forward network with hidden layer sizes of 10K, 20K, 40K, 80K, and 160K. Results are shown in Figure 3.6. We then run both to learn Word2Vec, varying the embedding dimensionality from 100 to 1K to 10K. Those results are shown in Figure 3.8. Finally, we run LDA learning on SimSQL, TensorFlow, and Spark using various topic counts: 1K, 5K, 10K, 50K, and 100K. Those results are shown in Figure 3.9. Note that these numbers are per-batch latencies. Since (for the feed-forward network and for Word2Vec) TensorFlow scales by running an independent, asynchronous batch on each machine, its throughput will be $10 \times$ greater than SimSQL, at the same latency.

Finally, to test scalability with respect to cluster size, we run the SimSQL feed-forward network implementation at 160K neurons per hidden layer (this was chosen since it the costliest per-iteration learning problem), but vary the cluster size, using one, five, ten, and 20 machines. Results are shown in Figure 3.7.

### 3.8.4 Discussion

As is clear from Figure 3.5, there is a clear benefit to the proposed graph cutting algorithm, algorithm. The proposed tuned full graph cut algorithm outperforms (by a factor of as much as $10 \times$) all other options.
### Figure 3.5: Latency of feed-forward neural networks and Word2Vec on 10 machines, using different graph cut algorithms in SimSQL. The hidden layers of feed-forward neural networks have 40K neurons each. The embedding dimensionality of Word2Vec is 100. Experiments over 2 hours are terminated. Format is HH:MM:SS.

<table>
<thead>
<tr>
<th>Graph Cut Algorithm</th>
<th>Feed-forward NN</th>
<th>Word2Vec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuned Full</td>
<td>00:10:01</td>
<td>00:10:26</td>
</tr>
<tr>
<td>Untuned Full</td>
<td>&gt;2h</td>
<td>&gt;2h</td>
</tr>
<tr>
<td>Tuned Greedy</td>
<td>&gt;2h</td>
<td>00:30:28</td>
</tr>
<tr>
<td>Untuned Greedy</td>
<td>01:35:43</td>
<td>00:11:19</td>
</tr>
</tbody>
</table>

### Figure 3.6: Latency of feed-forward neural networks on a cluster of 10 machines, as the size of hidden layers increases. Each mini-batch contains 100 documents. Format is HH:MM:SS.

<table>
<thead>
<tr>
<th>Hidden Layer Neurons</th>
<th>SimSQL</th>
<th>TensorFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000 (data parallel)</td>
<td>00:03:12</td>
<td>00:01:10</td>
</tr>
<tr>
<td>10000</td>
<td>00:09:18</td>
<td>00:01:11</td>
</tr>
<tr>
<td>20000</td>
<td>00:09:39</td>
<td>00:02:10</td>
</tr>
<tr>
<td>40000</td>
<td>00:10:01</td>
<td>00:05:22</td>
</tr>
<tr>
<td>80000</td>
<td>00:11:59</td>
<td>Fail</td>
</tr>
<tr>
<td>160000</td>
<td>00:37:05</td>
<td>Fail</td>
</tr>
</tbody>
</table>

It is also clear that for all three implementations, the SimSQL model parallel implementations were the only ones that were consistently able to scale. For the feed-forward network, TensorFlow was unable to scale past hidden layers with 40,000 neurons. The 80,000 neuron model requires 51GB of RAM simply to store the weight matrix—this is more than 75% of the memory available on our test machine, and so the computation fails. The 160K neuron model requires more than 200GB. Since a data parallel implementation must be able to store the model on a machine, it is impossible to handle this large model size with a purely data parallel implementation.
### Feed-forward Neural Networks

<table>
<thead>
<tr>
<th>Number of Machines</th>
<th>SimSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01:39:40</td>
</tr>
<tr>
<td>5</td>
<td>00:56:17</td>
</tr>
<tr>
<td>10</td>
<td>00:37:05</td>
</tr>
<tr>
<td>20</td>
<td>00:17:10</td>
</tr>
</tbody>
</table>

Figure 3.7: Runtime of feed-forward neural networks with a fixed hidden layer size of 160000, as the number of machines increases. Format is HH:MM:SS.

### Word2Vec

<table>
<thead>
<tr>
<th>Embedding Dimensions</th>
<th>SimSQL</th>
<th>TensorFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>00:10:26</td>
<td>00:08:03</td>
</tr>
<tr>
<td>1000</td>
<td>00:13:40</td>
<td>01:14:58</td>
</tr>
<tr>
<td>10000</td>
<td>00:20:39</td>
<td>Fail</td>
</tr>
</tbody>
</table>

Figure 3.8: Latency of Word2Vec on 10 machines, as the dimension of word embeddings increases. Format is HH:MM:SS.

In general, the per-iteration running time on SimSQL was slower than the per-iteration running time on TensorFlow, especially for the feed-forward network (though SimSQL was actually far faster than TensorFlow for the LDA implementation with a large number of topics, due to the very large number of small parameter requests to TensorFlow’s parameter server). However, we believe that much of this gap is related to two factors. First, a data parallel implementation is inherently faster *when it works*, since it generally has far less communication cost. In Figure 3.6, for example, we show the time required for a data parallel implementation on SimSQL, which is approximately 3× faster than the model parallel implementation. Second, SimSQL, with its reliance on Java and Hadoop, is simply not built for low latency.

Another interesting finding is that per-iteration running time—and the high throughput facilitated by TensorFlow’s asynchronous, model parallel implementation—is not
<table>
<thead>
<tr>
<th>Number of Topics</th>
<th>SimSQL</th>
<th>TensorFlow</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>00:10:49</td>
<td>00:05:06</td>
<td>00:00:39</td>
</tr>
<tr>
<td>5000</td>
<td>00:13:49</td>
<td>00:25:22</td>
<td>00:03:03</td>
</tr>
<tr>
<td>10000</td>
<td>00:14:08</td>
<td>00:52:35</td>
<td>00:06:39</td>
</tr>
<tr>
<td>50000</td>
<td>00:15:11</td>
<td>04:51:51</td>
<td>00:55:27</td>
</tr>
<tr>
<td>100000</td>
<td>00:16:10</td>
<td>Fail</td>
<td>01:42:35</td>
</tr>
</tbody>
</table>

Figure 3.9: Average sub-step runtime of collapsed LDA on 10 machines with varied numbers of topics. Format is HH:MM:SS.

Figure 3.10: Per-hour NCE loss of Word2vec with 100-dimensional word embeddings.

the only story. Consider Figure 3.10, where we plot the loss of the two Word2Vec learners as a function of time, for the 100-dimensional embedding. Even though TensorFlow’s per-iteration time is more than $10 \times$ faster than SimSQL, the loss decreases more quickly with SimSQL. This is likely because TensorFlow relies on asynchronous learning, which is often the only way to scale-up a data parallel computation. The problem with synchronous data parallel gradient-based learning is that more machines can only serve to increase the size of the batch used in each update—which is generally known to be ineffective past a relatively small batch size. The issue is that a larger batch makes the gradient estimate more accurate, but depending on the precise problem, past a small number of data points, the estimate is already so
accurate that the extra work is not worth it.

The solution is to *asynchronously* have each machine do its own concurrent model updates. This has the effect of having more updates per time unit as more machines are added. The problem here is that asynchronicity often leads to staleness, where a lot of work is being done using old parameters. This can slow convergence. This appears to be what we see in Figure 3.10.

On the other hand, model parallelism has no such problem. In theory, if a model parallel implementation is able to scale out, then faster convergence is achieved by simply adding more machines, allowing more iterations per time period, without complications such as staleness. Figure 3.7 seems to verify that the SimSQL implementation can achieve good scale out. Going from one machine to 20, the per-iteration time drops by a factor of $6 \times$—with a drop of more than $2 \times$ in per-iteration time when moving from ten to 20 machines.

Given that the model parallel implementation is quite simple on SimSQL (as shown in Figure 3.4, SimSQL’s model parallel, feed-forward network implementation requires only 206 lines of SQL code) we believe that these results suggest that an RDBMS-like system should receive strong consideration for use as a backend for a machine learning system. A commercial-quality, distributed RDBMS without SimSQL’s latency issues should be more competitive for the smaller, easier problems, and would also maintain SimSQL’s ability to scale to large clusters and model sizes.
Chapter 4

Overcoming Performance Bottlenecks in Distributed Machine Learning

4.1 Motivation

Various performance bottlenecks still exist when using distributed systems for large-scale machine learning applications. In this chapter, I take a closer look at several bottlenecks encountered during the physical execution of various machine learning tasks on top of SimSQL, and discuss how to eliminate these bottlenecks. Despite the fact that I focus on SimSQL, a lot of the ideas in this chapter will apply to other systems. The three bottlenecks I consider are:

Inefficient serial runtime compilation. Runtime compilation is a commonly used technique in distributed systems [24, 65, 66, 67]. It allows the systems to compile files dynamically and optimize adaptively in the middle of the execution. However, runtime compilation is usually done in a serial manner, where compilation of one computation will only start when execution of its previous computation has completed. This incurs a high overhead (e.g. 11–12 seconds per job in SimSQL).

Random partitioning of large matrices or vectors. Very often, a randomized hash function is used to partition the data objects as evenly as possible across machines. However, random partitioning works best when the number of data objects is large, and can be problematic in distributed linear algebra where we prefer to reduce the amount of communication by having a small number of matrix-type or vector-
type objects. Under this circumstance, random partitioning leads to an unbalanced distribution of data objects and is therefore inefficient.

**Suboptimal choices of physical plans.** A distributed system must avoid costly operations such as shuffling and sorting of large data sets. Unfortunately, if not designed and optimized carefully, a system may generate a suboptimal physical plan that includes expensive sorting and data materialization. I consider one such case in this chapter.

This chapter is organized as follows. For each of the aforementioned performance bottlenecks, I describe the problem in the context of a learning task, such as a feed-forward neural network, or a Gaussian mixture model. Then, via experiments, I show that addressing the bottleneck can lead directly to an improvement in efficiency.

### 4.2 Inefficient Serial Runtime Compilation

SimSQL [15] is a relational database system with a runtime built on top of Hadoop [8]. After a programmer writes SQL-like code for a machine learning application, the system converts the code into a series of MapReduce jobs that are run one by one. The computation inside one job is distributed and parallelized by Hadoop, but the jobs are executed in a serial manner, meaning that the initiation of the next job has to wait for the completion of the current job. The reason for serial execution is that, we need to know how many bytes of data are actually serialized (i.e. the size of the output relation) at the end of the current job, to make critical runtime decisions on the next job, such as whether some table from the current job is small enough to fit into the memory and so on. With different statistics on relation sizes, the physical plans generated by SimSQL will also be different.

The execution of a job is broken into three stages:
(1) Job preparation. This stage includes three sub-stages. First, a Java archive file (called a jar file) is compiled with all the necessary information about the operator under consideration. This process usually takes 11–12 seconds for every job. Second, a **Configuration** object is created with critical properties set, such as the maximum memory allowed for the mappers and the reducers. Third, a **Job** object is created with the customized **Configuration** object, and its properties are set, including the mapper and reducer classes, the number of reducers, the input and output paths, the input split size, the partitioner class and so forth.

(2) Job execution. The **Job** object above is deployed to the Hadoop cluster. This is the stage when parallelization actually takes place, based on the number of workers/cores available in the cluster.

(3) Statistics collection. Finally, we compute the size of the output relation. The preparation stage of the next job can then be started.

Ideally, we would like to overlap preparation of the next job with execution and statistics collection of the current job. However, without statistics, it is not possible to prepare the next job. Fortunately, since most machine learning applications are iterative, the statistics needed for the compilation of the next job can be guessed without seeing the data from the current job. Consider an iterative application such as a feed-forward neural network. The weight matrices or activations after the 100th iteration will be highly similar (if not identical) in size to those after the 1st iteration. Thus, if we can save all the statistics after the 1st iteration for future reference, we can run the current job by forking a new thread, while we start compiling the jar file for the next job immediately in the current thread. This is depicted in Figure 4.1.

Note that, we will only need one additional thread (called “Hadoop thread”) other than the main thread. That is because if some job is still running by the time we
are about to start off a new Hadoop job on the cluster, then we wait until that job finishes. Ideally, we would finish the preparation of the next job on main thread while the current job is running on Hadoop thread, and kick it off once the current
job completes. But it is possible for the current job to complete before the preparation for the next job is done.

Under this added constraint, the *singleton design pattern* serves as a suitable implementation. Since we only need one instance of the Hadoop thread and we want it globally accessible, it is natural to define the single-instance Hadoop thread as a static object `hadoopThread` that is initialized to `null` in the SimSQL runtime class `HadoopRuntime` along with a static boolean `firstIterationDone` to indicate whether we have the statistics obtained from earlier iterations to enable pre-compilation:

```java
// in HadoopRuntime.java
public static HadoopThread hadoopThread = null;
public static boolean firstIterationDone = false;
```

I define a new subclass of `Thread` named `HadoopThread` which takes as input the `Job` instance to run and submits that Hadoop job to the cluster when the thread becomes active.

```java
public class HadoopThread extends Thread {
    Job job;
    boolean verbose;
    int exitCode;
    public HadoopThread () {
    }
    public void setParams (Job job, boolean verbose) {
        this.job = job;
        this.verbose = verbose;
    }
}```
public void run() {
    try {
        exitCode = job.waitForCompletion(verbose) ? 0 : 1;
    } catch (Exception e) {
        e.printStackTrace();
        throw new RuntimeException("Unable to run the job", e);
    }
}

public int getExitCode() {
    return exitCode;
}

Finally, I implement my new logic in the operator class following jar file compilation and job preparation. The code here keeps only one HadoopThread instance to be accessed directly.

if (HadoopRuntime.firstIterationDone) {
    if (HadoopRuntime.hadoopThread != null
        && HadoopRuntime.hadoopThread.isAlive()) {
        try {
            // waiting for the previous thread to finish
            HadoopRuntime.hadoopThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
}
exitCode = HadoopRuntime.hadoopThread.getExitCode();
}
// starting a new thread
HadoopRuntime.hadoopThread = new HadoopThread();
HadoopRuntime.hadoopThread.setParams(job, verbose);
HadoopRuntime.hadoopThread.start();
} else {
    // running this op as before
}

To demonstrate the efficiency and efficacy of my pre-compilation implementation, I tested it on two iterative machine learning algorithms—feed-forward neural networks and Word2Vec. Both can be trained using mini-batch gradient descent. I evaluated each algorithm using various model sizes. The results are shown in Figure 4.2 and Figure 4.3.

From these results, one can see that the relative performance improvement from pre-compilation decreases with increase in model size. In general, the performance gain is proportional to the number of jobs per iteration, which tends to be constant for iterative applications. As was mentioned, each job preparation stage takes about 11 to 12 seconds, and this bottleneck can be eliminated nearly entirely by spawning one additional thread.

4.3 Random Partitioning in Distributed Linear Algebra

In a multi-machine environment, it is desirable for data objects as well as computation to be distributed as evenly as possible. In SimSQL, data records in one relation are
Figure 4.2: Per-minibatch latency (in seconds) of feed-forward neural networks with 40000 and 160000 neurons in the hidden layers respectively, on a cluster of ten EC2 m2.4xlarge machines.

Figure 4.3: Per-minibatch latency (in seconds) of Word2Vec with embedding dimensionalities of 100, 1000 and 10000 respectively, on a cluster of ten EC2 m2.4xlarge machines.
distributed according to the long-typed hash values of their sorting attributes. We call this hash function \( h(\text{byte[]}):\text{long} \), and assume the data of a record at its sorting attribute with a certain type (e.g. integer, integer array, double, double array, string, vector, matrix, etc.) has been written to its underlying byte buffer \( b \). The hash value can then be expressed as \( h(b.array()) \).

However, in certain cases, particularly in machine learning, the values of the sorting attribute are already uniformly distributed, and the total number of data objects is not that large, making it unnecessary, or even undesirable to hash these values with a randomized function, as randomization may lead to skew. For example, with 40 reducers, applying the existing hash function in SimSQL to a consecutive range of integers from 0 to 159 (160 integers in total) yields as many as 7 hash values on one reducer, and as few as just 1 hash value on another reducer. However, if uniformly distributed, each reducer should get \( \frac{160}{40} = 4 \) hash values to process. Assume the sorting attribute happens to be of type long, a better option for \( h \) is \( \text{bytes} \to \text{ByteBuffer} \text{.wrap(bytes)} \text{.getLong()} \), which does nothing but place the records on the workers in a round-robin manner. I have extended SimSQL to support this way of balanced partitioning, so that it is used when the system has detected matrix-type or vector-type data.

One case where this is useful is in the model parallel implementation of feedforward neural networks. In this case, all the weight matrices are partitioned in both dimensions, and the sub-matrices are indexed by \( \text{row} \) and \( \text{col} \). Assume we partition both dimensions by a factor of 160, and sort the sub-matrices by \( \text{row} \), then each value of \( \text{row} \) will map to 160 values of \( \text{col} \). Since values of \( \text{row} \) fall uniformly in the range of \([0,159]\), it makes sense to use these values of \( \text{row} \) directly as the hash. As long as 160 is a multiple of the number of reducers, all the 25600 sub-matrices will be evenly
distributed across the cluster.

Besides experiments on feed-forward neural networks, I also tested the new partitioning method on block matrix multiplication, where two input matrices are partitioned and distributed before their sub-matrices are multiplied and aggregated. Given input matrix $X$ of size $a \times b$ with $m$ row partitions and $p$ column partitions and input matrix $Y$ of size $b \times c$ with $p$ row partitions and $n$ column partitions, the product matrix $Z$ of size $a \times c$ with $m$ row partitions and $n$ column partitions can be expressed as

$$Z_{ij} = \sum_{k=1}^{p} X_{ik} Y_{kj}.$$  

This way of partitioning matrices resembles that in a model parallel implementation of a machine learning application. Each value of $\text{col}$ in $X$ maps to $m$ values of $\text{row}$ and each value of $\text{row}$ in $Y$ maps to $n$ values of $\text{col}$. It makes sense to apply the new identity hash function to the $\text{col}$ index of $X$ and the $\text{row}$ index of $Y$, since values of both indices fall uniformly in the range of $[0, p - 1]$. In SQL, the block matrix multiplication computation can be expressed as

```sql
SELECT X.ROW, Y.COL, sum(matmul(X.MAT, Y.MAT))
FROM X, Y
WHERE X.COL = Y.ROW
GROUP BY X.ROW, Y.COL;
```

Note that the attributes ($X.\text{COL}$ and $Y.\text{ROW}$) referenced in the join predicate are the attributes by which the records in $X$ and $Y$ are placed round-robin over the workers. Note that in this problem, there is another variable affecting the performance, which is the value of $p$. Therefore, I performed experiments not only in different values of $b$, but also in different values of $p$ under the same $b$ value.

Figure 4.4, Figure 4.5 and Figure 4.6 show the results for various matrix sizes in feed-forward neural networks and block matrix multiplication. A decrease of 2%
Figure 4.4: Per-minibatch latency (in seconds) of feed-forward neural networks with 40000 and 160000 neurons in the hidden layers respectively, on a cluster of ten EC2 m2.4xlarge machines using different partitioning schemes.

to as much as 57% in runtime was seen when the previous random partitioning was replaced by the new balanced partitioning. The improvement was especially obvious when \( p \leq \) the number of worker cores (which was 80 in my cluster setting). This is not surprising. With a round-robin placement, when the value of \( p \) is not greater than 80, one worker will be assigned at most one partition, ensuring a balanced distribution. This is not the case for a randomized hash function, where one worker may be assigned more than one partition and become a straggler.

### 4.4 Poor Choices of Physical Plans in Three-way Joins

Shuffling and sorting are both time and compute resource consuming. Fortunately, physical plans can be optimized in order to avoid expensive data shuffling. In the case of a join operator, if data partitions from two input tables are joined using the
attributes they are already sorted on, they can be merge-joined directly without first sorting before being merged. Assume two tables $A$ and $B$ are sorted by attributes $keyA$ and $keyB$ respectively, and table $A$ has some other attribute $nonKeyA$, the following join can be executed as a merge join without a sort:

```sql
SELECT *
FROM A, B
WHERE A.keyA = B.keyB;
```

But the one below cannot:

```sql
SELECT *
FROM A, B
WHERE A.nonKeyA = B.keyB;
```

Determining when sorted orders can be used to enable merges without sorting is
tricker when there are three tables involved in a join, each with a sorting attribute, and there may be multiple join predicates. Consider the following example from feed-forward neural networks, when the weight matrices are updated using the gradients:

```sql
CREATE TABLE W[i:1...][j:1...9](ROW, COL, MAT) AS
    SELECT W.ROW, W.COL,
           W.MAT - matmul(t(A.ACT), E.ERR) * 0.00000001
    FROM W[i-1][j] AS W, E[i-1][j] AS E, A[i-1][j-1] AS A
    WHERE A.COL = W.ROW AND W.COL = E.COL;
```

Three tables are involved in this three-way join: \( W(ROW, \text{COL}) \), \( E(\text{COL}) \) and \( A(\text{COL}) \). The underscore denotes the sorting attribute of that table. With 60000 and 80000 neurons in two adjacent layers \( j-1 \) and \( j \) as well as 100 data points in a mini-batch, the size of \( W \) is about 60000 \times 80000 \times 8 = 38GB, the size of \( E \) is 100 \times 80000 \times 8 = 63MB, and the size of \( A \) is 100 \times 60000 \times 8 = 47MB. It is obvious that \( W >> E > A \) in terms of size. Because of this, we can speed up the whole join by hashing one of the smaller tables (\( E \) or \( A \)) in memory.

In the SQL code above, the three-way join is done in a way such that \( A.\text{COL} = W.\text{ROW} \) and \( W.\text{COL} = E.\text{COL} \). \( W \) and \( A \) can be merge-joined in a map-only job since they are joined over their sort attributes. This is not the case for \( W \) and \( E \). But if we can hash the 63MB table \( E \) in memory for the purpose of a hash join, we end up with a good physical plan where we pipeline the other two tables (\( W \) and \( A \)) and the three-way join can still finish within just the map phase as a merge join.

However, SimSQL failed to do this, because the size of \( E \) is larger than \( A \) (in our neural network example, the size of \( E \) is slightly larger than that of \( A \) but both are quite small compared to \( W \)). Due to the heuristics used by SimSQL (and also by many other distributed systems), the system always hashes the smaller table in a hash join and pipelines the other, so SimSQL chooses to hash \( A \) in memory and pipeline \( E \). Then, it is not able to merge-join \( W \) and \( E \) without sorting them first, because \( W \) is not
The remedy for this bottleneck in a three-way join is to hash in memory one of the two smaller tables which if pipelined, cannot be directly merge-joined with the largest table without additional sorting. I extended the system to optimize over a three-way join as one unit when determining the execution plans, and implemented this remedy.

Besides the weight update example commonly seen in feed-forward neural networks, I also evaluated a three-way join in Gaussian mixture models. Given a number of Gaussian distributions as well as their means and covariances, Gaussian mixture models compute the membership of each data point in these Gaussian distributions (or, “clusters”) and update cluster means and covariances based on the memberships. Consider the following SQL code that calculates the per-data, per-cluster covariances to be used later when we update the covariance matrix of each cluster:

```sql
SELECT ME.DATAID, ME.CLUSTERID, 
      outerproduct(DA.POINT-MO.MEAN, DA.POINT-MO.MEAN) 
FROM DATA[i] AS DA, MEMBERSHIP[i] AS ME, MODEL[i] AS MO 
WHERE DA.DATAID = ME.DATAID AND MO.CLUSTERID = ME.CLUSTERID;
```

Three relevant tables here are: `DA(DATAID, POINT)`, `ME(DATAID, CLUSTERID, MEMBERVECTOR)` and `MO(CLUSTERID, MEAN, COVAR)`. Since `DA` and `ME` are both sorted on `DATAID`, based on our previous discussion, `MO` should be hashed as long as it can fit into the memory. This way, `DA` and `ME` can be pipelined and merged in the same order, which avoids expensive data shuffling.

Figure 4.7 and Figure 4.8 are the performance comparisons using different join heuristics in physical plan generation, for feed-forward neural networks and Gaussian mixture models respectively. Note how the values of $N$, $D$ and $C$ in Figure 4.8 affect the order of table sizes. For instance, given $N = 100k$, $D = 1000$ and $C = 10$, `DA` is approximately $8ND = 800$MB, `ME` is $8NC = 8$MB and `MO` is $8CD^2 = 80$MB. The new join heuristic chose to hash `MO` in memory despite the fact that `MO` > `ME`
4.5 Discussion

In this section, I briefly discuss how the aforementioned remedies for various performance bottlenecks can be generalized and applied to other large-scale machine learning platforms.

**Parallel runtime compilation.** For any platform involving a series of critical jar files or other type of files generated on the fly, it helps to have some kind of approximate inference on the ongoing job, so that the generation of the file next in line won’t be contingent upon the completion of the current one. The approximate inference makes sense due to the iterative nature of most machine learning applications.
Balanced partitioning for matrices and vectors. Balancing workloads among worker nodes is almost always a good idea to avoid stragglers, even if not in a MapReduce-like system. Some system takes care of this automatically, but some does not. For example, in Spark, programmers can control the number of partitions to cut the input dataset into when parallelizing it. Increasing that number can effectively relieve the problem caused by stragglers.

Three-way join optimization. It is good practice to keep sorting orders and avoid sorting to the greatest extent possible. As is known, sorting has $O(n \log n)$ average time complexity and can get worse in practice for really large applications. This is the reason why merge-joins or hash-joins easily outperform sort-merge joins, and why map-only jobs are preferred on Hadoop.

Last but not least, materializations of data should only be done when extremely
necessary, in particular for very large data. Possible alternatives in the existence of very big data are (1) pipelining it to its consumer(s) so that it becomes transient and phantom; (2) distributing it among the memory of worker nodes so that it can be accessed and concatenated in a fast manner. The latter is equivalent to an in-memory model parallel solution.
Chapter 5

Related Work

In this chapter, I study the important related work in the field of large-scale and distributed machine learning.

5.1 Dataflow and Synchronous Systems

The most well-known modern data analytics system is probably Hadoop [8]. Hadoop is a distributed dataflow framework that contains both storage (HDFS [68]) and computing (MapReduce [69]) modules. A typical MapReduce program consists of a map phase that performs some operations on the input data (in the form of key-value pairs) in parallel, a shuffle phase that shuffles or sorts the intermediate data generated from the map function, and a reduce phase that aggregates the intermediate data with the same keys. One drawback is that at the end of each MapReduce job, Hadoop always writes the output back to the disk, incurring I/O overhead and making it hard to re-use the same data across different operations. Hadoop achieves fault tolerance by checkpointing the master node, and re-running mapper or reducer tasks in case of worker failures. Hadoop is synchronous in the sense that a job is not considered “done” and no output is given until all workers have completed.

There are two categories of problems Hadoop cannot handle well: iterative jobs and interactive tasks. Iterative parameter estimation is commonly used in machine learning and in Hadoop, it requires the same data to be loaded from disk again
and again. Interactive tasks always load in the same data and query over it multiple times. These drawbacks were the main reason for the development of Spark [9]. RDDs (Resilient Distributed Datasets [10]) are the key abstraction of Spark, and they are essentially immutable, parallel collections of data with various APIs. An RDD doesn’t need to be materialized each time it is used and is usually cached in memory for reuse by multiple operations. It is read-only and stored across multiple machines. An RDD can either be created through a rich set of lazily-evaluated transformations, or via a persist method that specifies which data to be reused. No output can an RDD return until an action is called. The lineage feature makes it easy to recover failed RDDs, as lineage records information on how one partition of an RDD is generated by others. Spark also provides checkpointing functionality for RDDs with long lineages that might require a long time to recover. RDDs make Spark quite usable as well as scalable on large data.

A popular graph-based synchronous system is Pregel [70], which is built on the Bulk Synchronous Parallel model [71]. It consists of a sequence of iterations, called supersteps. Each superstep handles the incoming and outgoing messages, as well as updates the state of the vertices, and requires synchronization at the end. The synchrony of each superstep makes Pregel free of data races naturally. The master node is responsible for notifying the workers that it is time to checkpoint their state when a superstep starts. GraphX [72], Spark’s graph-parallel module, is built on the Pregel API. Giraph [73] is the counterpart implementation of Pregel, and is extensively used at Facebook. Vertices and edges in Giraph are both stateful. Giraph uses a vertex-centric approach that is similar to MapReduce, where each vertex runs a local UDF in parallel, before its output is aggregated. Edges do not have their own computations. Recently, Ching et al. [74] added graph processing features beyond the
basic Pregel model and reported successful support for trillion edge graphs. Their improvements include allowing edge data loading from different sources along with vertex data, adding support for local worker multithreading computation, sharding aggregators across workers and adding centralized computation on the master before distributed computation on workers.

A distributed system specialized in streaming dataflow computing is Flink [33]. It aims to fill the void between MapReduce-like systems and database systems, by providing a UDF-centric data-parallel programming model equipped with a declarative coding interface and a cost-based optimizer. The support for continuous data sources (streaming) in addition to support for processing static files makes Flink stand out.

Because these systems often borrow ideas from distributed database systems and integrate recursive training for machine learning models as well as provide transparent management for low-level operations, such as job scheduling, logical/physical graph optimization, task communication and networking, etc., these systems can be used as the backend for training deep neural networks. For example, Spark provides native libraries for deep learning, and there exist several deep learning frameworks running on top of Spark, such as Deeplearning4j [75] and BigDL [76].

5.2 Parameter Servers and Asynchronous Systems

The parameter server architecture [45, 46] was proposed to provide scalable, parallel training for machine learning models. The basic architecture consists of two parts: a parameter server and a set of workers. The parameter server stores and updates the parameters for the model to be trained, where these parameters are accessible by all workers. The workers are responsible for the computation for the parameters, and interact with the parameter server by sending updates of the parameters, as well as
fetching the current parameter values. Parameter server programming is usually not declarative. Although it is possible to write fast parameter server codes, the burden on programmers is high.

Support for data parallelism in a parameter server is straightforward. Each worker is supplied with different subsets of the training data; the workers fetch the model parameters, perform computations over the data and the model, and send the updates back to the parameter server. However, model parallelism seems a little more difficult to implement in parameter server. I now discuss some of the common parameter server-based systems in use.

DistBelief [1] is a framework that targets the training of large deep neural networks on a large number of machines. It utilizes a parameter-server-like architecture, where model parallelism is enabled by distributing the nodes of a neural network across different machines. The parameters are also partitioned and stored in a group of machines in the parameter server. While the efficacy of this architecture was tested and verified on two optimization algorithms (Downpour SGD and Sandblaster L-BFGS), it is unclear precisely what support DistBelief provides for declarative or automated model parallelism. For example, for SGD, the computation of the activation for the current layer involves a matrix multiplication between the activation from the last layer and the weight matrix of the current layer. However, it is unclear how this matrix multiplication is implemented when the two matrices are partitioned across a set of machines. As we have argued, an RDBMS can easily handle such issues, as a distributed matrix multiplication (optimized by the system optimizer) requires only a few lines of SQL code.

Tensorflow [11, 77] utilizes a similar strategy for model parallelism. Although it provides some functions (e.g., \texttt{tf.nn.embedding\_lookup}) that allow parallel model
updates (this function is used in Word2Vec), support for more complex parallel model updates is limited. For example, TensorFlow does not supply a distributed matrix multiplication.

It is notable that either of these system could supply a distributed matrix multiplication as a library call—there is nothing preventing the use of a tool such as ScaLAPACK [78] in either system—but this is a very different approach than the sort of end-to-end optimizable computations described in this paper.

Project Adam [79] applies a similar architecture for use in learning convolutional neural networks. The models are partitioned vertically across a set of workers, where fully-connected layers and the convolutional layers are separated. This partition works for the convolutional layers as the volume of weights is low for those layers. However, it is again unclear what sort of support Project Adam supplies for automated computation over very large fully-connected layers.

MXNet [80] is another recent system that employs a parameter server to train deep neural networks. The authors of the MXNet paper claim the system supports model parallelism. However, its model parallelism support is similar to TensorFlow in that this parallelism can only be applied easily mostly when the structure of the neural network itself makes model parallelism easy. More complex computations that require model parallelism (such as distributed matrix multiplication) require using low-level APIs and manual management of the computations and communications.

Petuum [55] is a general-purpose framework that provides data parallelism and model parallelism support for large-scale machine learning. The architecture of Petuum consists of a scheduler, a set of parameter servers, and a group of workers. The parameter servers aid in realizing model parallelism, while the scheduler provides the support for data parallelism. Petuum provides model parallel implementations
for a large number of machine learning models. However, for a learning algorithm to be amenable to model parallelism, the algorithms for learning these models need to be able to be easily partitioned. It is unclear how Petuum could handle the large feed-forward network tested in this paper.

There are several other systems exploring model parallelism for deep learning [81]. AMPNet [82] adds control flow to the execution graph, and supports dynamic control flow by introducing a well-defined intermediate representation. This framework proves to be efficient for asynchronous model-parallel training by the experiments. Coates et al. [83] built a distributed system on a cluster of GPUs based on the COTS HPC technology. This system achieved model parallelism by carefully assigning the partial computations of the whole model to each GPU, and utilized MPI for the communication.

Another asynchronous system worth mentioning, especially for graph-based computation, is GraphLab [84, 85]. Like Pregel, GraphLab adopts a vertex-centric model where graph-parallel computations (in the form of an update function or UDF) are defined on each vertex. Different from Pregel’s push-style message passing strategy, GraphLab uses a pull-style execution model that focuses on the sequential procedure each vertex performs on its state, along with all its adjacent vertices and edges. The support for asynchronous iterative computation in GraphLab leads to higher efficiency while synchronous computation suffers from stragglers and has to be blocked until all nodes finish. GraphLab ensures serializability on asynchronous computation through full, edge and vertex consistency models. Because a vertex can access all the adjacent vertices and edges, it can implement the Gather-Apply-Scatter paradigm in C++ explicitly. GraphLab is also more flexible and expressive than Pregel in that a GraphLab vertex is highly “privileged” and can access data on all neighboring ver-
tices while a Pregel vertex can only use the data passed via messages. One notable limitation of GraphLab is that it requires the whole graph to fit in memory, which may cause problems when the models are large. GraphLab achieves fault tolerance via asynchronous incremental checkpointing.

5.3 Declarative Systems for Machine Learning

Declarative programming frees machine learning programmers and data scientists from difficult decisions such as the proper representations of data and the optimal choice of execution plans, and it provides them with the convenience of writing a single machine learning code that may be used regardless of model/data sizes, or available compute resources.

There are some mature efforts focused on providing distributed machine learning in a declarative manner. For example, SystemML [86, 87] performs optimization of distributed, matrix-based computations. Its cost-based compiler converts high-level specifications of machine learning algorithms into execution plans that can run on back-end data-parallel frameworks such as MapReduce or Spark. Further, with ideas borrowed from database compression techniques, SystemML explores the memory and I/O friendly compressed linear algebra [88] representations for matrices, so that linear algebra operations can be executed on compressed matrices directly. However, while matrix operations are treated declaratively, non-matrix computations are not.

The programming language for SystemML is an R-like, imperative language. Mahout Samsara [89] and SciDB [90] likewise give some support for optimization of distributed computations using large arrays and matrices. Boehm and co-authors give a very nice description of existing, declarative systems for machine learning [91].

Some other efforts target adding declarative support on top of mature non-declarative
platforms for more efficient data access. Spark SQL [92] is a module on top of Spark to query structured data in almost standard SQL. It is equipped with a cost-based optimizer. Hive [93] is built on top of Hadoop and allows SQL-like queries for data stored on HDFS or the underlying file system. Its compiler is responsible for converting the SQL-like queries to a series of MapReduce jobs.

The SQL-like language constructs used by SimSQL[15] for user-specification of the join-and-co-group pattern were first proposed in the paper describing MCDB (Jampani et al. [94]). That paper gave the basic SQL syntax and proposed the template implementation depicted in the join-and-co-group pattern, but it did not consider implementation issues past the template. The proposed naive, local, and global implementations are unique to this thesis, as is the assertion that join-and-co-group is, in fact, a general pattern useful for statistical computation and performing large-scale optimization. Also, before being extended for multi-dimensional recursive computations, SimSQL only provided one-dimensional recursion, assuming the Markov property; that is, in a database-valued simulation, relations at the current iteration are only dependent on those at the previous iteration. Therefore, the previous frame-cutting problem was easier than the one presented in this thesis.

Some papers [95, 96] show that relational database systems can provide effective support for big data analytics and machine learning. SAP HANA [97] and Hyper [98, 99] are two in-memory database systems that support both OLTP and OLAP.

5.4 Special-Purpose Deep Learning Tools

In recent years, many frameworks and libraries have been developed specifically for learning deep neural networks. Theano [100] is a Python library that facilitates computations on multi-dimensional arrays, and provides an easy support for writing deep
learning algorithms. Caffe [101] is one of the earliest specialized frameworks for deep learning, and mainly focuses on applications to computer vision. Caffe2 [43] extends Caffe to provide a better support for large-scale, distributed model training, as well as the support for model learning on mobile devices. Torch [102] is a computational framework in which users can interact with it with the language Lua. Its python version, PyTorch [44], applies dynamic computation graphs. Similar ideas are adopted by DyNet [103] and Chainer [104] as well. The Microsoft Cognitive Toolkit (previously known as CNTK) [105] is a toolkit that can help people use, or build their own deep learning architectures. In addition, higher level APIs are developed on top of those aforementioned frameworks to provide more flexibility for programmers. For example, Keras [106] supports TensorFlow and Theano as its backend, and Gluon [107] is run on MXNet. While most of these frameworks and libraries support model training in a distributed environment, training typically uses data parallelism, not model parallelism. Theano does support putting independent computations on different GPUs, but it does not provide a complete framework for developing general-purpose model parallel computations. Torch provides an example of (manually) putting different layers of a convolution neural network on multiple GPUs. But this model partitioning is limited as the components inside of a layer cannot be partitioned. The model parallelism over multiple GPUs in Chainer is similar in the sense that programmers need to specify the computations for each GPU, and take care of synchronizations and communications between GPUs.

5.5 UDFs in Declarative Systems

There is a significant amount of related work considering the implementation and execution of UDFs in systems for data analytics. Most of this work differs from what
I present in this thesis, in that it does not consider methods for efficiently matching different sets of parameters with different UDF invocations. I briefly mention a few of the papers now.

There is classical work in database systems that is focused on optimizing queries that contain UDFs [108, 109]. Most focus on the difficulties resulting from the fact that a user-defined function is a black box. Hence, it is difficult for an optimizer to know how expensive its execution is going to be [110, 111]. Despite the fact that the classical work on UDFs appeared before large-scale machine learning became prevalent, such papers shed light on the issues associated with optimizing over UDF executions.

In recent years, various big data processing systems with a declarative (or semi-declarative) front-end language, such as Hive [93] and Pig [112], provide extensive support for UDFs. One could imagine incorporating the ideas from this thesis into such systems. Popular systems for data analytics, including Hadoop [113], Spark [9], and Flink [33] are fundamentally based on the idea of processing UDFs. These systems typically manipulate user-defined host language objects (specified in Java, Python, Scala, etc.) where the semantics of each manipulation are specified by defining methods on the objects, or lambdas that are used to process the objects. Join-and-co-group could be implemented as an API on top, without changing the internals of the systems. This could significantly simplify the implementation, while at the same time it is in-keeping with the philosophy behind the design and implementation of such platforms.

Tupleware (Crotty et al. [114]) is a recently-proposed data analytics system designed to efficiently support UDFs by compiling their execution into an SQL-specified computation using LLVM [65]. Tupleware’s design is based on the assertion that an
emerging challenge in data analytics lies in the UDF computation itself, not in the data volume. The core ideas in the Tupleware system could easily be used in conjunction with the ideas presented in this thesis. Friedman et al. [115] describe a framework for implementing and executing UDFs in a database system, where the focus is on supporting a form of run-time polymorphism for UDFs. Kwon et al. [116] describe how to handle computation skew in a Hadoop-like setting, where different UDF invocations can take radically different times to execute. The key innovation is a query planning framework that is able to generate skew-tolerant computational plans.

Ordonez considered how database UDF frameworks could be used to process statistical computations [117]. He claims that UDFs integrated inside a DBMS are more efficient than C++ code running outside and only require one table scan. While I believe C++ (with its scientific libraries) is still the best solution to high-complexity statistical computations, the focus of this thesis is co-grouping the data points with the model parameters before fully invoking the UDFs. MadLib (Hellerstein et al. [118]) is a library of UDFs and SQL codes for implementing various statistical and analytics computations on top of a parallel DBMS. Both of these efforts are focused on utilizing existing database UDF APIs.

5.6 Recursive Queries and Transitive Closure Computations

Papers on recursive queries as well as transitive closure computations in relational database systems have been around for thirty years. Among the efforts is the well-known Datalog language [119], which is a declarative, recursive query language for logic programming. Users of Datalog specify a finite chain of rules, usually including a recursive rule that has a fact both in the head and the body. Since Datalog is
declarative, users do not need to worry about the implementation details of the rules.

Much effort has been devoted to improving algorithms for transitive closure computations over database relations. These algorithms largely fall into two categories. *Iterative* algorithms aim to solve the recursion by applying a relational algebraic expression, until no more new tuples are generated in the relation and a fixpoint is reached. In the naive implementation [52], the operator needs to be applied to all the tuples (including the known ones) in every step, which slows down the computation a lot. Semi-naive [120] and logarithmic [121] algorithms can address this issue by only applying the operator on the tuples generated in the previous iteration. The other category is the so-called direct algorithms, which are more specific to transitive closure and process each node or edge directly. Direct algorithms may be matrix-based such as Warshall’s Algorithm [122] and Warren’s Algorithm [123], or graph-based such as Schmitz’s Algorithm [124] and Schnorr’s Algorithm [125]. Other variants [126, 127] of direct algorithms exist, but are mostly dependent on the aforementioned matrix-based or graph-based algorithms. Rakesh Agrawal [128] even studied the parallelization of these algorithms in a multiprocessor setting via both shared memory and message passing.

Recently, there has been a renaissance of Datalog implementations in large-scale applications. Myria [129] is a distributed analytics system that supports recursive Datalog-based parallel synchronous/asynchronous query evaluation in a shared-nothing engine. It has various failure handling mechanisms, but does not embody a cost-based optimizer as of yet. SociaLite [130] is a Datalog implementation in distributed social networking. It enables graph analysis in social networks by generating the parallelized code automatically with its compiler, and programmers only need to specify how data are sharded. Both Myria and SociaLite support recursive monotone
aggregate functions. DeALS [131, 132] explores adding aggregate functions to traditional Datalog on a single machine. LogicBlox [133] is a Datalog-driven commercial platform that focuses on enterprise applications. REX [134] focuses on propagating the changes, or delta, across iterations. It supports an SQL-extended programming model called RQL, and enables fixpoint recursion with implicit termination, similar to the Datalog semi-naive algorithm described earlier.

Other recent works on recursive query evaluation include implementing iterative transitive closure algorithms on a multi-core shared memory machine [135] and on computing clusters [136]. The authors assert that a transitive closure algorithm that outperforms others on a multi-machine cluster is not necessarily faster on a single multi-core machine, due to memory bandwidth limitations. Since later iterations tend to generate few new facts, recursive algorithms that finish in fewer iterations are favored in a distributed setting due to file transmission cost.

Much effort has been made to add Datalog recursion to existing programming models. MapReduce [69] and Hadoop [8] are very inefficient for iterative applications. Several extensions aim to resolve this issue. Haloop [137] extends MapReduce by supporting synchronous iterative applications, so that recursion can be handled by a set of MapReduce iterations. It delivers a loop-aware scheduler which tends to assign mappers and reducers that access the same data to the same physical machines. To facilitate data re-use across iterations and reduce I/O, Haloop caches the inter-iteration data on the physical machines’ local disk. Twister [138] is another iterative MapReduce system which identifies the static and dynamic data and supports static data caching to avoid repeated loading. Both mappers or reducers in Twister can read from or write to the distributed memory. But this assumes that all intermediate data fit in the distributed memory of the cluster. PrIter [139] is a distributed
framework which prioritizes iterations that benefit convergence most. It maintains a StateTable to specify the priority value of each data point and only extracts data with top priority values for iterative updates. The top-$k$ algorithm used to select the most important data is approximate and makes use of sampling. Afrati et al. [140] studied recursion implementations in a distributed cluster as well as failure recovery for recursive tasks on Hadoop. BigDatalog [141] incorporates declarative, recursive Datalog rules into the Spark programs and evaluates the Datalog programs through parallel semi-naive evaluation. BigDatalog’s compiler borrows the idea from the sequential DeALS compiler [131] and extends it to support parallel evaluation. With its recursive-syntax-based compiler and optimizer, BigDatalog turns the Datalog programs into physical plans and makes decisions on RDD caching automatically. It even adds a recursion operator specifically for semi-naive computation. Bu et al. [142] explore the idea of leveraging recursive queries in various machine learning applications and argues that Datalog is a natural choice for machine learning programming due to its iterative characteristics. Their main contribution is a declarative representation in Datalog for the Pregel and MapReduce programming models. However, a fully functional recursive-query-driven declarative system specialized in machine learning is yet to be completed.
Chapter 6

Conclusions

Large-scale optimization has become an important application for data management systems, particularly in the context of statistical machine learning. In this thesis, I have identified the *join-and-co-group* pattern, which is a ubiquitous pattern in iterative, large-scale optimization. I have carefully considered the alternatives for implementing the join-and-co-group pattern, as well as how the best alternative can be selected automatically. I have implemented all of our ideas within the SimSQL parallel database system, and on top of the Spark dataflow platform. My experiments have shown that it is perhaps more feasible to produce a high-performance implementation of the join-and-co-group pattern *within* a parallel database system where it is possible to control the low-level details of the implementation, than it is to implement the join-and-co-group pattern *on top of* a dataflow platform where key implementation choices are out of reach.

I also have considered the problem of building large-scale, model parallel machine learning applications. Much of the cutting-edge work in the area assumes that a parameter server is the right compute platform for distributed machine learning, and popular ML platforms such as TensorFlow use a parameter server to execute their computations. I have given strong evidence that a parameter server is *not* the right platform—that an RDBMS with an extended SQL API is a better choice. I have extended SQL so that it can support recursive, ML-like computations without using imperative constructs, and I have proposed a frame-based execution strategy.
for running the very large plans (perhaps tens of thousands of relational operations) that result.

Furthermore, I have studied three performance bottlenecks that exist in many distributed systems. To resolve these bottlenecks, I have engineered various machine learning-specific improvements including balanced partitioning of large vector-type or matrix-type data in distributed linear algebra, order-preserving physical plan generation for three-way joins, and parallel runtime compilation using multithreading.
Bibliography


analysis & transformation,” in *Code Generation and Optimization, 2004. CGO

[66] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, “Merge: a program-
ning model for heterogeneous multi-core systems,” in *ACM SIGOPS operating

for heterogeneous computing systems,” *Computing in science & engineering*,

file system,” in *2010 IEEE 26th symposium on mass storage systems and tech-

[69] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large

[70] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and
G. Czajkowski, “Pregel: a system for large-scale graph processing,” in *Proceed-
ings of the 2010 ACM SIGMOD International Conference on Management of

[71] L. G. Valiant, “A bridging model for parallel computation,” *Communications

distributed graph system on spark,” in *First International Workshop on Graph


[142] Y. Bu, V. Borkar, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer,
and R. Ramakrishnan, “Scaling datalog for machine learning on big data,”