D-AIDA: A Distributed Framework for Unified Query Processing and Machine Learning

Yu Jia He

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Faculty of Science

McGill University
Montréal, Québec, Canada

October 2023

Abstract

The advent of machine learning in the present day computer science landscape invites the collection of larger and larger datasets by interested entities. These datasets are often stored in a distributed fashion, across different computer nodes instead of in one monolithic system for multiple reasons. Moreover, much of the data is stored in relational databases, one of the most popular data management systems. The relational database system offers many tools for data analytics and processing, spawning its own realm of research topics.

Much research has gone into performing machine learning on such distributed data. Traditional methods may include amassing all the data on a singular machine, which is unlikely to be feasible in terms of available resources. Some frameworks have arisen allowing computation to instead be pushed to the data nodes, but these still involve extracting data from the database. This inhibits further operations using the database engine.

To this end, we create a unified framework that can perform these machine learning operations without removing data from the database – thus allowing a user to interweave operations required for machine learning and relational operations optimized by the database engine efficiently. Using a previous project designed for in-database analytics, we expand it to perform distributed operations across several database nodes, each hosting a partition of data. This extended project allows distributed relational queries and user-defined machine learning algorithms to be executed near data and by the database engine where applicable.

In this thesis, we present a proof of concept of such a framework designed using a middleware architecture. We present implementations of basic distributed relational operations, as well as frameworks provided for users to write distributed machine learning algorithms. We explore

the comprehensiveness of these frameworks by implementing several distributed machine learning algorithms that are common for relational data. We additionally examine the scalability of this system by testing the performance times of such algorithms in our system, and comparing them against pre-existing, database external distributed machine learning frameworks. Through this, we are able to identify the current limitations of this framework in its architectural and communicative sub-components.

Abrégé

L'avènement de l'apprentissage automatique dans le paysage de l'informatique contemporaine invite les entités intéressées à collecter des ensembles de données de plus en plus vastes. Ces ensembles de données sont souvent stockés de manière distribuée, sur différents nœuds informatiques plutôt que dans un système monolithique, pour de multiples raisons. De plus, une grande partie des données est stockée dans des bases de données relationnelles, l'un des systèmes de gestion de données les plus populaires. Le système de base de données relationnelles offre de nombreux outils pour l'analyse et le traitement des données, donnant naissance à son propre domaine de sujets de recherche.

De nombreuses recherches ont été menées pour réaliser l'apprentissage automatique sur de telles données distribuées. Les méthodes traditionnelles peuvent inclure l'accumulation de toutes les données sur une seule machine, ce qui est peu probable en termes de ressources disponibles. Certains frameworks ont émergé permettant au calcul d'être délégué aux nœuds de données, mais ils impliquent toujours l'extraction de données de la base de données. Cela limite les opérations ultérieures utilisant le moteur de base de données.

À cette fin, nous créons un framework unifié capable d'exécuter ces opérations d'apprentissage automatique sans retirer les données de la base de données, permettant ainsi à un utilisateur d'entrelacer efficacement les opérations requises pour l'apprentissage automatique et les opérations relationnelles optimisées par le moteur de base de données. En utilisant un projet précédent conçu pour l'analyse en base de données, nous l'étendons pour effectuer des opérations distribuées sur plusieurs nœuds de base de données, chacun hébergeant une partition de données. Ce projet étendu permet l'exécution de requêtes relationnelles distribuées et d'algorithmes d'apprentissage automatique définis par l'utilisateur à proximité des données et par le moteur de base de données lorsque

cela est applicable.

Dans cette thèse, nous présentons une preuve de concept d'un tel framework conçu à l'aide d'une architecture intermédiaire. Nous présentons des mises en œuvre d'opérations relationnelles distribuées de base, ainsi que des frameworks fournis aux utilisateurs pour écrire des algorithmes d'apprentissage automatique distribués. Nous explorons l'exhaustivité de ces frameworks en mettant en œuvre plusieurs algorithmes d'apprentissage automatique distribués courants pour les données relationnelles. Nous examinons également l'extensibilité de ce système en testant les temps de performance de ces algorithmes dans notre système et en les comparant aux frameworks d'apprentissage automatique distribués externes à la base de données existants. Ainsi, nous sommes en mesure d'identifier les limitations actuelles de ce framework dans ses sous-composants architecturaux et communicatifs.

Acknowledgements

This thesis would not have been possible without the support of my supervisors, Professor Bettina Kemme and Joseph D'Silva. Their hard work, advice and oversight has been invaluable in the writing of this thesis. In particular, Dr Kemme has provided funding and editing in support of this thesis, and Dr D'Silva the technical aid and knowledge required for the implementation of several aspects.

I am also grateful for my friends and peers who have supported and taught me along the way. Through them, I was introduced to many concepts outside my scope of research, enriching my time at school.

Lastly, I would like to thank my parents for their never ending support and patience throughout my time working on this thesis.

Table of Contents

Acronyms

AIDA Advanced In-Database Analytics

D-AIDA Distributed AIDA

DBDC Density Based Distributed Clustering

DBSCAN Density-Based Spatial Clustering of Applications with Noise

MSE Mean Squared Error

ORM Object Relational Mapping

RDBMS Relational Database Management System

RMI Remote Method Invocation

RPC Remote Procedure Call

SGD Stochastic Gradient Descent

SQL Structured Query Language

SSP Stale Synchronous Parallelism

List of Figures

List of Programs

1 Introduction

In this first chapter, we present the motivation behind this thesis, introduce our contributions and give a brief overview of the organization of this thesis.

1.1 Problem Statement

Machine learning has become an integral part of new technologies being researched and deployed in the modern era. Problems too large and complex to be solved by traditional algorithms can be approached through use of various machine learning techniques.

Moreover, much of the data used in machine learning exists in relational form. This model of data is popularized by its ease of use, scalability, and the powerful query language supporting it. Indeed, many popular datasets used for machine learning can be found in relational form – either as separate tables or as a result of a relational query. In fact, much of the data available

to enterprises is already stored in relational form in a Relational Database Management System (RDBMS). This system serves as the powerhouse for the management of relational data natively. As businesses expand and more data is collected, data storage becomes naturally distributed to cope with increasing storage and retrieval costs. Now, it is more essential than ever that such systems are able to support machine learning and data analytic tasks alongside their traditional relational operations.

However, organizing and managing such a distributed database system is a non-trivial task. Much effort has gone into distributed processing of relational queries and implementing systems to manage large-scale distributed databases. Kossman [26] offers a survey of various optimizations and methodologies for traditional relational database operations in a distributed setting. However, scalable platforms that support advanced analysis and maching learning are mostly built outside the database execution environment. Many assume that the data resides in a distributed shared file system and data is then retrieved from this data storage to worker nodes to perform the analytic computation. The MapReduce programming paradigm [9], for instance, assumes a distributed file system, but can optimize performance by executing the map phase on the same nodes on which the data partitions reside. Spark [1], a general purpose data processing platform, is also based on a shared file system, but can also retrieve data from relational database systems. However, should further relational operations be needed to be performed on the retrieved data, they must be executed within Spark. Although Spark has its own query execution system, it is not as sophisticated as those offered by the traditional RDBMS. Some machine learning tools such as PyTorch or Tensorflow support distribution, but all take data out of the database system initially and generally do not support any relational operations on the data.

As such, we are not aware of a unified framework where distributed relational query processing, data analytics and machine learning tasks are carried out in a distributed relational database ecosystem. In this thesis, we aim to provide such a framework. This framework should be able to use features of the RDBMS for efficient query processing as well as offer convenient methods for distributed data processing and machine learning tasks, such as those provided by Spark.

1.2 Thesis Methodology and Contribution

In this thesis, we present D-AIDA, a user-friendly framework for conducting distributed data processing and machine learning for data existing in relational database systems. D-AIDA supports embedding of these processes within the database system in order to take advantage of the querying capabilities provided by the underlying engine. This framework allows the user to perform relational operations on distributed data as well as allowing easy deployment of distributed machine learning algorithms across several data servers. It offers a user friendly programming API with a unified data abstraction for relational and data analysis tasks.

D-AIDA builds off of previous work presented in [12]. AIDA is a framework that provides an agile programming environment that allows data analytic computations to be pushed into a RDBMS to be executed near data. This thesis seeks to expand on this idea by scaling AIDA up to allow data processing and computation to occur across multiple nodes. The AIDA framework is extended to provide global views of data partitioned across multiple AIDA servers, distributed query processing on such partitioned data, and infrastructure for supporting user-defined data processing and machine learning tasks in such a system. The API of the framework provides a unified data abstraction and allows for use of popular machine learning libraries such as PyTorch. The middleware-based distributed architecture we propose enables the client to connect to and execute programs across the overall system through a single point of contact. It also provides the user with an easy to understand architecture for global views and models.

We also identify several weak points in the D-AIDA architecture that may impede more performant results. The sole reliance on AIDA Remote Method Invocation (RMI) as a communication paradigm over potentially faster ones causes D-AIDA to perform worse in cases where there is a lot of communication between nodes. As D-AIDA is coded in Python, it is subject to the limitations of Python as a programming language – namely, the global lock interpreter which prevents true parallelism, and the relatively slower speed of Python compared to compiled languages. While D-AIDA can access libraries such as PyTorch or NumPy, which are natively written in C, the communication process must happen in Python. Additionally, users must be more involved when writing models using the D-AIDA frameworks – they must be able to separate parts of the algorithm according to how they wish to distribute it.

Despite this, D-AIDA shows more flexibility in the distributed algorithms that can be implemented and executed in it. D-AIDA implements three distributed machine learning frameworks in order to increase the scope of distributed algorithms that can be implemented in D-AIDA. We then test these frameworks against those offered by PyTorch and present the results. Furthermore, we show that distributed relational algorithms can be implemented in D-AIDA through use of a middleware architecture, through implementing a subset of relational operators as a proof of concept. The middleware architecture also provides an intuitive site for accessing a global view of the data, and storing shared variables.

1.3 Thesis Overview

The rest of the thesis is organized as follows:

- Chapter 2 pertains to the background of data processing and machine learning techniques referenced in this thesis, as well as the AIDA system and related research in this area.
- Chapter 3 contains a description of the architecture and data structures used in D-AIDA.
- Chapter 4 provides an in-depth overview of the distributed relational operations provided by D-AIDA.
- Chapter 5 delves deeper into the frameworks provided for distributed machine learning tasks in D-AIDA.
- Chapter 6 contains experimental results for execution of various distributed machine learning algorithms written using the frameworks provided by D-AIDA, comparing it with distributed implementations offered by PyTorch [34].
- Chapter 7 concludes this thesis, presenting a summary of the results and expands on future work.

2

Background and Related Work

2.1 Relational Data and Databases

First introduced by E. F. Codd in 1970 [5], relational databases are an often used method of storing data. In a relational database, data is stored in tabular format. Each row in the table signifies a data record (or tuple) and each column describes an attribute of that data. Tuples are unordered and unique in the table. Data records can be inserted, deleted or updated in their table. The collection of data tables is managed by the RDBMS, which also provides various operations used for managing and querying the data. The type of data stored in relational databases is referred to as relational data and it, along with the relational operations available on it, will be the focus of this thesis.

The relational model is so called because each table represents the mathematical n-ary relation between the n attribute sets which compose the columns of the table. The attribute or set of

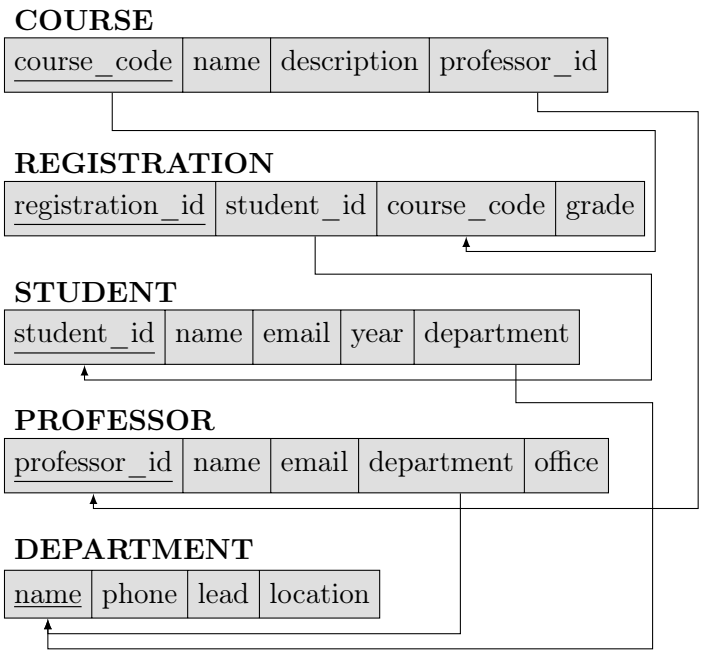

Figure 2.1: Example of relational database schema

attributes which define a unique tuple in the table is called the primary key and is used for identifying and accessing the tuple. Tuples in relations can also be linked to tuples in other relations by having one or more attributes of the first relation refer to the key attributes of the second — this is known as the foreign key.

Figure 2.1 shows an example of a relational database that one might find in a school server. Each of the tables has multiple attributes describing the data contained within. The underlined attributes are the primary keys of the table, and must be unique within the table. Furthermore, the arrows represent foreign keys referencing other tables. These relationships can be varied – for instance, a student may belong only to one department, but may register for many courses.

This setup allows definition and usage of various operations to be performed on the data. These operations are called relational operations and can be written through the use of Structured Query Language (SQL) [4], a standardized and powerful language used across different relational database implementations. Many database implementations not only fully support SQL queries, but include many optimizations in their engine to rewrite and optimize such queries. The queries themselves are constructed from a set of base relational operators that can be combined in various ways:

1. Projection (π_a): The projection operation on a table T, denoted $\pi_a(T)$ returns a subset a

of columns of all the data records in T. This operations focuses on displaying only certain attributes of interest in a table, and discarding the rest. For example, the operation $\pi_{\texttt{student\_id, email}}(\texttt{STUDENT})$ returns the student id and name of all students in the table, with nothing else. In SQL, this operation can be expressed as `SELECT column_1, column_2 FROM table_name`.

2. Selection (σ_c): The selection operation on a table T, denoted $\sigma_c(T)$ returns the set of all records in T fulfilling the condition c. This operation filters the output data to a limited amount of records realizing the condition. An example of this would be $\sigma_{\texttt{department='math'}}(\texttt{PROFESSOR})$, which returns all the professors in the math department. In SQL, this operation is expressed as `SELECT * FROM table_name WHERE condition`.

3. Join ($\bowtie_c$): The join operation between two tables T and S, denote $T \bowtie_c S$ returns the combination of T and S on some condition c. The condition c is some comparison or equality on attributes of both table T and S – the attribute on which c acts on is called the *join attribute*. On the schema portrayed in figure 2.1, the operation $\texttt{COURSE} \bowtie_{\texttt{professor\_id}} \texttt{PROFESSOR}$ returns the data for all courses, along with the data for all professors teaching those courses. The result table of this query would have attributes spanning both tables, each record matching the records of `COURSE` and `PROFESSOR` where the `professor_id` attribute is the same. This operation in SQL is `SELECT * FROM table_1 JOIN table_2 ON table_1.column=table_2.column`.

4. Aggregation operations, such as finding the maximum, minimum, count, or average of a certain column in the table, return a single value to produce summary statistics of data stored in a table. An example of an aggregate operation in SQL could be expressed as `SELECT AVG(grade) FROM REGISTRATION` which computes the average grade across all records in `REGISTRATION`.

5. The `GROUP BY` SQL operator allows a single query to perform multiple aggregates on different groups of data defined by distinct values in the column provided to the `GROUP BY` operator. For example, the statement `SELECT course_code, AVG(grade) FROM REGISTRATION GROUP BY(course_code)` retrieves averages of all courses, along with the course codes, from the registration table. In most instances, `GROUP BY` operations are implemented by sorting the data tuples on the `GROUP BY` attribute, and performing the aggregation on each unique value.

A single difference exists between relational operators and SQL queries – relational operators discard duplicates in their results, while SQL queries do not. For thes rest of this thesis, we treat them as equivalent.

As each of the operators returns a relation, these operations can be performed consecutively, building a complex query containing multiple joins and selections additionally to projections and group bys. From there, query processing is a multi-step process [18, 26] executed by the database engine. First, the engine must parse the query – it will be translated to some internal representation, such as a tree that shows the order in which the operations will be executed, to simplify the later steps. Next, the query will be rewritten. There are many queries that may return the same result set, so the query is rewritten in order to eliminate redundancies and simplify expressions to improve performance. Query optimization is the next step – the database management system uses table metadata and other tools to decide how best to execute a query. This can include reordering the operations in the tree and deciding how exactly to execute each operation. Finally, the query will be executed according to the plan generated by the previous steps.

There has been much research done in the database community for query optimization. The classic method of optimization involves minimizing the disk I/O cost of a given query. For instance, modern day databases provide a variety of indices to make retrieving data faster and easier. As an example, the B-tree [3] is a popular index which speeds up selection queries, as it stores for each unique attribute value the location of all tuples that have this value. A query such as `SELECT student_id FROM REGISTRATION WHERE course_code='101'`, which finds all students registered for the course `'101'`, would require a full table scan to check all tuples in the table to find the matching ones if an index does not exist. However, should an index exist on the attribute `course_code`, then all the database needs to do is look up the index to find the locations of the tuples with `course_code='101'`, and load those into memory instead of the entire table. Furthermore, there exists several join implementations whose performance depends on the cardinality of the input relations.

2.2 Big Data and Distributed Data

With the ever-growing popularity of machine learning, the need for large amounts of data to facilitate the training of models has increased dramatically within the last decade. To train models that are used for increasingly complex tasks, large amounts of training data are required to ensure adequate performance. To this end, many corporations depending on large training sets make use of distributed data — that is, data that are not contained within a single machine. Often, the amount of data is large enough as to not be able to fit in the disk space of a single machine, and it is likely cheaper to expand storage space by adding more machines, rather than augmenting a single monolithic system.

Additionally, data can become distributed through natural use, even without considering the costs of improving a single server. Applications might choose to host their data across servers for many reasons – to avoid a single point of failure, so that data and transactions would not be lost should a single server shutdown, or to give clients in different locations a data server located geographically closer to them to speed up response times. Database servers hosting the same kind of data may have grown independently, each managed by a different corporation in a competing sector before being consolidated through a merger.

However, distributing data is not without its own obstacles. Algorithms that use such data must now consider the cost of transferring and locating data. Data processing pipelines and algorithms must adapt to the fact that not all the data resides on a singular node, and may not be visible from every part of the system. Yet it also allows for further parallelization of tasks — since the data is spread across multiple nodes, so too may the computation. As each separate node has its own storage, so too does each have its own computational power. In recent years, much effort has been devoted creating and improving algorithms designed for deployment in such distributed systems.

There are even multiple ways of distributing data. Vertical partitioning splits a data table into smaller tables based on columns, where each database node only holds a subset of the columns in the entire table. In horizontal partitioning, or sharding, data is partitioned based on rows. Each database server contains a subset of the rows belonging to the global schema. Often, the partitioning can be based on a specific attribute of the table – for example, all records belonging to a certain geographic region existing in one database node. For this paper, we will assume the latter case

when discussing distributed data. That is, each database node holds a subset of rows of a larger table, which we will refer to as a partition.

2.3 Distributed Database Operations

Distributed relational operations refer to relational operations executed on data that is distributed across several nodes. While some queries are easy to distribute, such as projections and selections, others are more difficult.

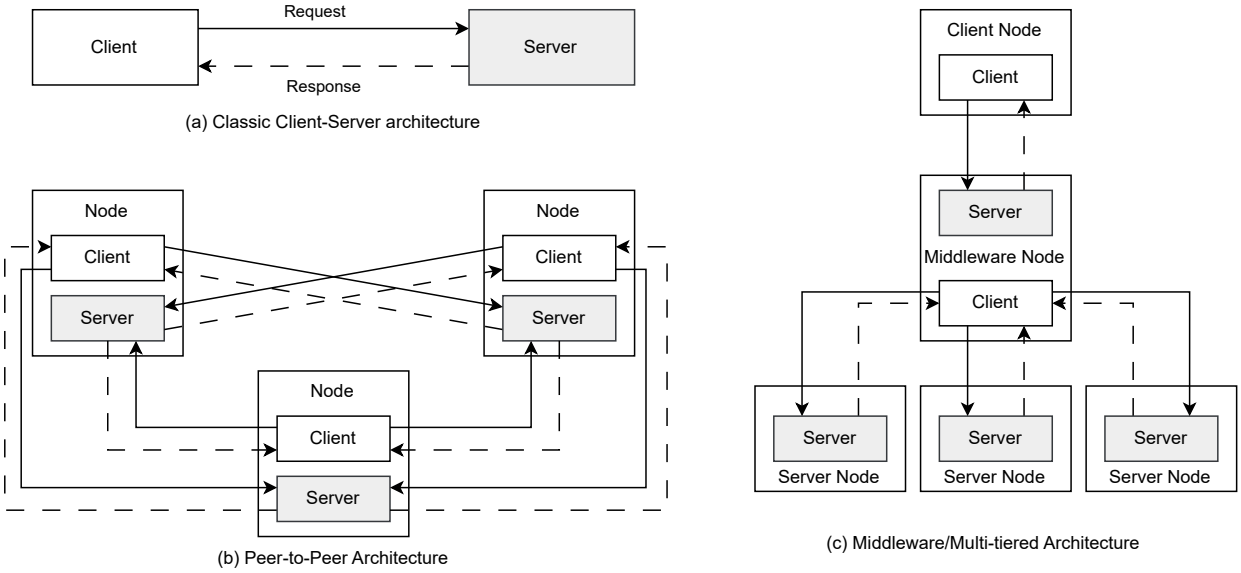

Figure 2.2: Distributed server architectures

In a survey on distributed query processing, Kossman [26] presents various architectures and techniques used in such systems. To begin, we must first examine the architecture available to distributed database systems. In particular, we focus on client-server systems – systems wherein a client node sends a request to a server, which then returns some response. In peer-to-peer systems, each node can act as both a server that stores a partition of the database and a client which requests data from another. In middleware or multi-tiered systems, nodes are layered hierarchically in such a way that they act as servers for nodes in the level higher than them, and as clients for the nodes the level lower. Diagrams of these architectures can be found in figure 2.2. In all of these systems, the client node is the one to initiate the query, and the server nodes the ones that contain the data.

With these architectures in mind, the question of executing distributed queries remains. Koss-

man [26] further presents three techniques: query shipping, data shipping and hybrid shipping.

1. Query shipping has the client ship the SQL code of the query to the server, which then executes the query and returns the result to the client. If there are multiple database servers, some middleware server must exist in order to perform joins on tables that are stored in different nodes. This technique is performant when the server machines are much more powerful than the client machines, but could be bottlenecked by server performance if there were many clients.

2. In data shipping, queries are executed at the client side instead. Data is requested by, then cached at, the client, which can then execute the query locally. Subsequent queries on the same data can then use these cached copies. Data shipping scales well to many clients as the execution for all of the clients is not performed all on the same servers, but can incur extremely high communication costs if the data tables are big. Additionally, client machines must be able to support relational operators, which may not always be the case.

3. Hybrid shipping is a combination of the previous two methods. For instance, in a join between two tables, one table could be cached at the client while the other is at the server. The server will scan its stored table and send the results back to the client, which will perform the join. The table sent by the server will not be cached by the client. However, the optimization of hybrid shipping is far more difficult than either of the prior shipping options and may perform poorly if done incorrectly.

For this thesis, we focus on query shipping, as we assume an environment where only the server has a database execution engine, thus only the server has access to the query optimizations provided by the database management system. Due to this, we also focus on the most natural architecture for query shipping: the middleware architecture which provides a middleware server to which the client can request queries, and which can perform joins and aggregates for data spread across several database servers.

A selection operation on a horizontally partitioned database table involves sending the query to each node containing a partition of the data. The client will communicate the query with the middleware, which will then broadcast the query to each of the database servers holding a partition.

Each server evaluates the selection condition on its local data, filtering out the records that do not meet it. Then, each servers sends the result of their own local query to be aggregated at the middleware, which will then forward the aggregation to the client. Naturally, the records in the resultant table represent all records in the distributed table matching the selection criteria. By performing a distributed selection in this manner, each node works only on their local partition, and less data is sent over the network compared to if the entire dataset is aggregated on a single node, and the selection performed afterwards. Performing a distributed projection is very much the same.

Most aggregate queries exhibit simple distributed algorithms. Operations like count or sum merely perform that operation on each database node, and sum the results at the middleware. Finding the minimum and maximum values are similar, only requiring the operation to be performed again on the results. Calculating the average of data is merely finding the sum, then dividing by the count of records.

Distributed joins require some degree of data transfer between databases to ensure all records are matched. There are many algorithms for distributed joins, but this paper explores two: broadcast join and hash join.

Broadcast join involves transmitting the entirety of one table to all the database servers holding a partition of the second table. Each server conducts a local join between the entirety of the first table and the local partition of the second table, and the results are sent to the middleware to aggregate. This technique is easy to understand and useful when one table is much smaller than the other. In that case, the smaller table will be replicated to all the other nodes to minimize the amount of data being sent over the network. Naturally, this technique fails if the table is too large.

Instead of broadcasting a single table, distributed hash join seeks to have each database server be responsible for a subset of the tuples of both tables. These subsets are determined through hashing. Given n database servers, n hash buckets are created at each server and each server hashes the join attribute of their data partition of each table. The data records are each put into a hash bucket corresponding to the hashed join attribute – thus, data records with the same join attribute are put in the same bucket. Each hash bucket is assigned a server, and the data records assigned to that bucket are transmitted to that server. This ensures that all records with the same join attribute are located at the same node. Now housing a new partition of the data according to the join attribute,

each database server performs a local join in parallel on their partitions, and the results are sent to the middleware to be aggregated.

Query optimization is another concern for distributed query processing. In a centralized system, disk I/O cost estimation correlates well with query throughput. Distributed systems, on the other hand, introduce intra-query parallelism, the cost incurred by transporting data through the network, as well as load balancing and servers with different capabilities. Kossman's survey [26] describes many such optimizations, but those are out of scope for this thesis.

2.4 Machine Learning

Machine learning is a subset of Artificial Intelligence (AI) which focuses on using data to create a statistical model that is able to provide insights on the data or predict the behaviour of future data points. Hernán et al. [19] identify several data science tasks – description, prediction, and causal inference – that machine learning can be used for [24]. In this thesis, we will focus on the former two tasks: description and prediction.

Description is a task dedicated to examining data and providing a quantitative summary of certain features contained within. It is a task suited to unsupervised learning algorithms [24]. Unsupervised learning algorithms work on unlabelled data (thus, 'unsupervised'), to find structure and patterns inherent with the data [10]. Some examples of unsupervised learning include clustering – grouping together and separating data based on their similarities and differences – or principal component analysis – finding which features in the data are most important and descriptive in order to reduce the dimensionality of data with many features. For example, unsupervised learning can be used in recommendation systems where users with similar features can be grouped together by the algorithm, and recommended the same product. The model learned in descriptive tasks refers to the outcome of the task – for instance, the group labels assigned by the learning algorithm.

Prediction is a task that analyzes data in order to predict a certain outcome. For this, supervised learning algorithms are used [24]. Supervised learning requires labelled datasets for learning – datasets that include both features of the data as well as the desired outcome that is being studied. Using the labels of the dataset, supervised learning algorithms are able to learn when their predictions are correct or not, and change the predictions accordingly based on the result. Supervised

learning tasks involve classification – assigning data to certain categories – and regression – predicting the numerical outcome of a given data point. These algorithms could have applications in health care, such as predicting the likelihood of a particular disease given the patient's symptoms. The model in prediction tasks is often a mathematical function, which takes data as input and performs some transformation on it in order to output the predicted labels. The process of finding a function that outputs the correct labels – by changing the parameters of the function – is referred to as "training".

For this thesis, we will focus mainly on algorithms that are popular on relational data. The data points will be the rows of a table, written as $[x_{i1}, x_{i2}, ..., x_{in}, y_i] \in \mathbb{R}^n$, where i represents the ith record in the database table, and the indices of the vector represent the columns of the table. Each x represents an attribute of the training data, whereas y represents the true label, in the case of supervised learning. In the following sections, we explain the algorithms behind linear regression, matrix factorization, and clustering without considering implementation details for the time being. Out of these, linear regression and matrix factorization are supervised learning algorithms, whereas clustering is unsupervised.

2.4.1 Gradient Descent Methods

Iterative methods are defined by the iterations undergone by the algorithm as it moves gradually towards the optimal solution – the optimal solution being values for model parameters that result in the model's predictions being the most accurate. Both supervised and unsupervised learning use them. Gradient descent is a well-known and extremely popular iterative method for supervised learning. At the start of the algorithm, model parameters are initialized with random variables. In each iteration, the algorithm goes through the entire labelled training data, uses the current model to predict the labels for the data and calculates an error (or loss) value as a measure of the difference between the predicted value for each training record and its true value. The goal of gradient descent is to move towards the minima of the loss value function through iterative updates to the model parameters of the prediction function. Thus, an iteration of gradient descent is described as having two phases: the forward pass, which calculates the predicted values, and the backward pass, which updates parameters based on the loss.

Stochastic Gradient Descent (SGD) is a variant of gradient descent that, instead of calculating

the loss and updates for all of the training data in each iteration, calculates it instead for a randomly selected data point. Because updates occur more frequently, SGD moves faster towards the local minima, but exhibits more variance due to its stochastic nature. A technique which takes advantage of both variations is minibatch stochastic gradient descent – a version which calculates in each iteration the update according to a randomly selected subset of the data – known as a "minibatch".

Algorithm 1 A generalized minibatch stochastic gradient descent algorithm

Input: Training data D

1 $f \leftarrow$ Randomly initialized model
2 $\alpha \leftarrow$ learning rate
3 **while** $loss >$ some threshold **do**
4 $\quad batch\_features, batch\_labels \leftarrow$ get_batch(D)
5 $\quad preds \leftarrow f(batch\_features)$
6 $\quad loss \leftarrow$ loss_function($preds$, $batch\_labels$)
7 $\quad \nabla \leftarrow \frac{\delta}{\delta\omega}$ loss_function
8 $\quad f \leftarrow f - \alpha * \nabla$

The algorithm 1 presents a general overview of minibatch stochastic gradient descent. In line 1, the prediction function (or model) f is initialized with random values as parameters – call these ω. Line 2 sets the learning rate, or step size, α as a constant hyperparameter. The iterative process of the algorithm starts at line 3. Every iteration retrieves a minibatch (hereon referred to as a 'batch' for simplicity) of the overall data. Predicted labels for the data are made by inputting the data through the prediction function, and the loss is calculated between the predicted values and the true values through some loss function. Since the predicted values are the output of the f function, the loss function can then be differentiated with respect to f's parameters, ω. This differentiation ∇ is calculated at line 7. In order to minimize the loss function, the weights of the model should step in the opposite direction of the gradient; the magnitude of the step is defined in the beginning as α, the learning rate. This occurs at line 8, as is commonly referred to as the gradient update. While this example stops the iterations after the loss passes some arbitrary threshold, other stopping points exist. Common metrics for stopping the algorithm involve stopping it after a certain number of iterations or after a certain amount of time has passed.

Optimizing gradient descent is a well studied field in machine learning, due to its ubiquity of use in supervised learning algorithms [41]. The choice of learning rate α is non-trivial – setting it too low causes convergence to occur very slowly, whereas setting it too high may prevent it from

converging at all. Many algorithms choose to introduce adaptive learning rates – learning rates that change based on the number of iterations passed, the direction of previous gradients [37], or even the parameter being updated [13].

It is important to note that each iteration of minibatch stochastic gradient descent builds on the previous iterations; updates applied to the model in one iteration will reflect on the predictions of the model for the next. Every update samples a minibatch from the training dataset without replacement. An epoch passes when the entire dataset has been sampled. Often, training a model involves several epochs over the training data. The data is often shuffled once at the beginning of each epoch, so the iterations comprising the epoch can select minibatches without replacement, and once the iterations have gone through the entire dataset, the epoch ends, and the dataset is reshuffled for the next epoch.

2.4.2 Linear Regression

Linear regression is a well-known algorithm used for deriving a real-valued outcome variable y_i from a data point x_i with one or more features $x_{i1}, x_{i2}, ..., x_{in}$. To do so, linear regression models the relationship using the equation

$$y_i = w_0 + w_1 x_{i1} + w_2 x_{i2} + ... + w_n x_{in}$$

where the weights, or parameters, of the model $w_0, w_1, ..., w_n$ must be learned. The w_0 term is also known as the bias term, as the data features do not interact with it. Note that if the data has d features, then the model has $d + 1$ parameters, including the bias term.

While linear regression can be solved analytically for small datasets, it becomes impossible and extremely hard when the datasets grow large. Thus, scientists use gradient descent. Gradient descent, as described in Section 2.4.1 focuses on minimizing the error function (also known as the loss or cost function). While there exists many different error functions, we will focus on Mean Squared Error (MSE) for linear regression.

Let us first consider the algorithm for SGD, which uses a singular data point each iteration. Let $\hat{y}_i = f(x_i)$ be the output predicted by the model f on the output x_i and y_i be the true label. Then, the MSE can be written as:

$$(\hat{y}_i - y_i)^2 = (f(x_i) - y_i)^2 = ((w_0 + \sum_{j=1}^{n} w_j x_{ij}) - y_i)^2$$

To minimize this error function, we find the partial derivative of the loss with respect to each of the parameters. This becomes a direction that the value of the parameter must travel in order to approach the true value. For example, the partial derivative of MSE with respect to w_j is:

$$\Delta_j = 2 * ((w_0 + \sum_{k=1}^{n} w_k x_{ik}) - y_i)(x_{ij})$$

This gradient Δ_j must then be applied to the specific parameters after being multiplied by the learning rate α. Thus, the weight going forward from this iteration will be:

$$w'_j = w_j - \alpha * \Delta_j$$

Many iterations of this algorithm will be run until the error function converges, or until the update is near zero. Other stopping rules exist, as briefly explained in section 2.4.1. The variation in performance of the function can be reduced by using multiple data points per iteration instead of a single one, as in the case of minibatch SGD. In that case, the gradient Δ must be divided by the number of data points in the batch. Thus, in a case where there are n data points in a batch, with each data point providing a gradient of Δ_i, the update equation will be as follows:

$$w'_j = w_j - \frac{\alpha}{n} * \sum_{i=1}^{n} \Delta_{ij}$$

2.4.3 Matrix Factorization

Matrix Factorization [25] is an algorithm used for recommendation systems and determines the matrix factors of a larger dimensional and often sparse matrix. Given a large $m \times n$ matrix M, the goal of matrix factorization is to find two smaller $m \times p$ and $p \times n$ matrices, U and V respectively, that multiply to the larger matrix, where p is a user-defined hyperparameter. In essence, this creates a vector representation of each row and column of M, also known as an embedding.

Consider the use case for movie recommendation systems. The matrix M consists of all the ratings for every film given by every user on the platform. Thus, the element $M_{i,j}$ represents the

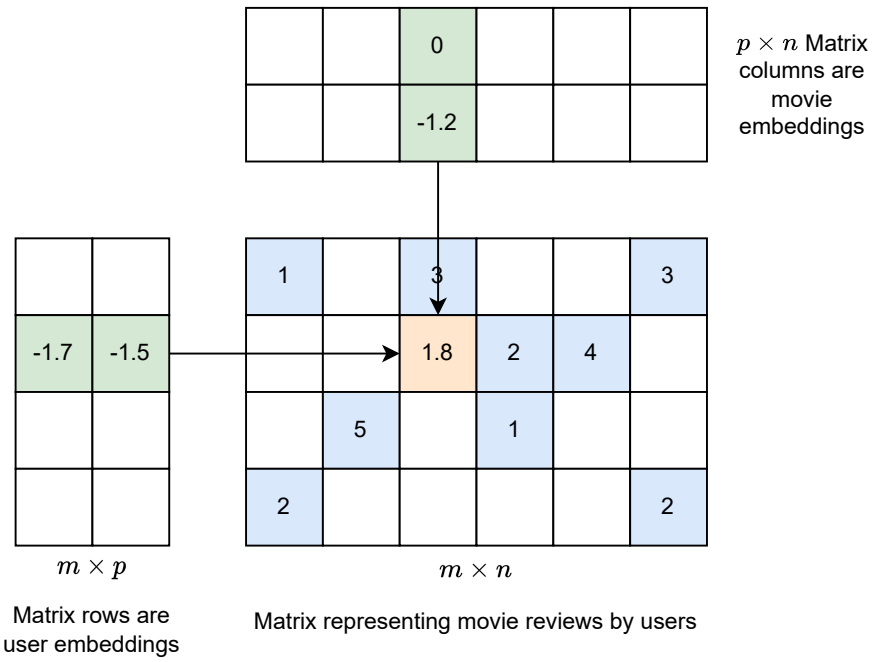

Figure 2.3: Example demonstrating matrix factorization

rating given by user i on the film j. Naturally, this matrix is very sparse and can be stored in an RDBMS as tuples of the form $(i, j, M_{i,j})$. The matrix U represents embeddings for all the users – each row being the vector respresentation of one user, while the matrix V represents embeddings for all the movies – each column being the vector representation of a movie. In this manner, the prediction for user i's affinity to a movie j can be found by multiplying the ith row of U with the jth column of V. The figure 2.3 presents this example. The blue indices in the matrix are the given samples: movies which users have rated. The green columns and rows in the two factor matrices represent the learned embeddings for the user and movie we are interested in – in this case, the second user and the third movie. The orange value represents the predicted rating given to the third movie by the second user.

Thus, the model for matrix factorization consists of both U and V. The factorization process performs gradient descent in much the same way as linear regression by calculating the gradient on the error between the prediction and the true values. It must be noted that for each iteration, that not all model parameters are used. Instead, only the specific rows and columns of U and V are used by the learning for that iteration are needed and updated. For example, if an iteration uses a batch of ratings made by the same user on different movies, then the iteration would only require

and update a single row from matrix U representing that user, and only the columns of matrix V representing the movies that user rated.

2.4.4 Clustering

Clustering data involves grouping the data points such that each group consists of data points that are deemed similar to each other by some metric, and such that data points belonging to different groups deemed relatively dissimilar. There are many such metrics of similarity, and thus many different clustering algorithms [15]. For this paper, we will focus on two of the simplest and most common algorithms, Density-Based Spatial Clustering of Applications with Noise (DBSCAN) and K-means. Stored in a database, cluster data is often just a table with an arbitrary number of features.

K-means [30] is an algorithm in which every data point is assigned a cluster that it belongs to. It requires the user to choose a hyperparameter k defining the number of clusters. The naïve implementation, introduced by Lloyd [29], starts with selecting k random data points to act as cluster centroids. The algorithm then repeats the following two steps:

1. Assign all data points to a cluster according to which centroid is closest, measured by Euclidean distance.

2. After all the clusters have been assigned, find the centre of each cluster and re-assign the centroids as the centre.

The algorithm terminates when the centroids are stable and no longer need to be reassigned each iteration. Though simple and widely used, there are some disadvantages to this algorithm. The number of clusters, k, must be manually chosen and may not represent the true number of clusters in the data, and the solution found by k-means is not guaranteed to be optimal. Correctness of the algorithm can be improved by restarting it with different initial centroids and keeping the clustering with the minimal distance between every centroid and the data belonging to its cluster, or choosing better initialization centroids [2].

Another algorithm is DBSCAN [14]. The input for this algorithm requires two parameters: a distance ϵ defining an ϵ-neighbourhood around each point, and an integer $minPts$ defining the

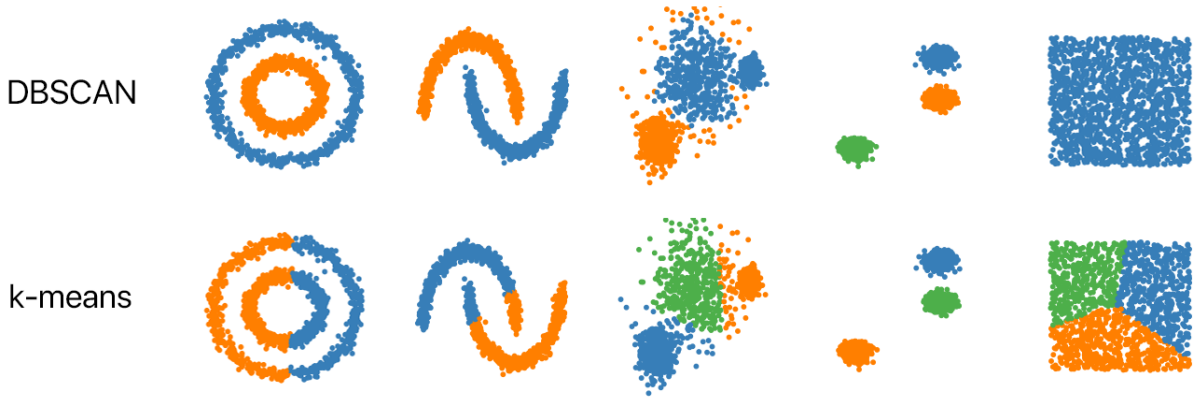

Figure 2.4: Example of differences in clusters found between K-Means and DBSCAN. Taken from [33].

minimum number of points within the ϵ-neighbourhood of a center point required for a cluster to be formed. DBSCAN defines the notion of *core points*. A point p is a *core point* if there are at least $minPts$ points within its ϵ-neighbourhood. A point q is *directly-density reachable* from p if p is a core point and q is within its ϵ-neighbourhood. A point q is *density reachable* from p if there exists a path $p_1, p_2, .., p_n$ such that p_{i+1} is directly density reachable from p_i and $p_1 = p$ and $p_n = q$. If any point p is a core point, then the cluster it belongs to are all the other points that are density reachable from it. The border of the cluster is formed by points that are directly density-reachable to another point in the cluster, but are not core points themselves.

This approach has its advantages and disadvantages. Once again, ϵ and $minPts$ are chosen arbitrarily and may not represent the true form of the data. However, unlike K-Means, DBSCAN can detect arbitrarily shaped clusters, can choose to not include outliers or noise in its clusters, and does not require the number of clusters to be known beforehand. The differences in these two algorithms are show in figure 2.4. As this figure shows, K-means takes a defined number of clusters, and must sort the data into these clusters, whereas DBSCAN can determine without input the number of clusters in the data.

2.4.5 PyTorch

PyTorch [34] is a popular Python library that implements several modules allowing for users to write and run deep learning models, but can also support simpler models such as linear regression and

matrix factorization. All mathematical operations use the `torch.tensor` class for data representation, which supports multi-dimensional data. The PyTorch library is known for its use of automatic differentiation through its engine `autograd`. The `autograd` engine keeps track of operations performed during the forward pass and loss calculation in a directed acyclic graph, and the gradient update is automatically calculated through backwards traversal of this graph. The gradient update is stored in the `.grad` attribute of the tensors involved. This library supplies functions for various layers, costs, and optimizers to facilitate all aspects of training.

```
1  import torch
2
3  class LinearRegression(torch.nn.Module):
4      def __init__(self):
5          super().__init__()
6          self.linear = torch.nn.Linear(5, 1)
7
8      def forward(self, x):
9          return self.linear(x)
10
11 model = LinearRegression()
12 loss_func = torch.nn.MSELoss()
13 opt = torch.optim.SGD(model.parameters(), lr=0.02)
14 for i in range(iterations):
15     opt.zero_grad()
16     preds = model(data)
17     loss = loss_func(preds, target)
18     loss.backward()
19     opt.step()
```

Listing 2.1: Example of linear regression written in PyTorch.

The listing 2.1 provides an example of the program flow for training a linear regression model in PyTorch. The model is defined from lines 3-9; this one contains only a single layer, provided by `torch.nn.Linear`. The loss function, in this case mean squared error provided by `torch.nn.MSELoss()`, is defined at line 12. The optimizer is the function that applies the gradient update to the model parameters. In this case, it is defined at line 13, using the default stochastic gradient descent update rule on the parameters provided by `model.parameters()` with a fixed learning rate of 0.02. The iterations start from line 14 onwards – this example uses a fixed number of iterations. First, any existing gradients stored by the model parameters are discarded in

line 15. Next, the forward pass of the iteration occurs, as predictions are made on provided data by the model. At line 17, the loss is calculated between the predicted labels and the true labels. The `loss.backward()` step in line 18 performs the backward pass – gradients are calculated and stored in the `.grad` attribute of the model parameters – in this case, just `model.linear.grad`. Finally, the optimizer applies the stored gradients to the model parameters, and the process repeats itself for the next iteration.

2.5 Distributed Machine Learning

Distributed machine learning is the process of running these algorithms over multiple machines. Recall from section 2.2 that it is preferable to increase computational power and storage space through adding more machines than improving a single one. State of the art machine learning models can consume data in the order of terabytes and models with trillions of parameters [16]. However, such models often learn on datasets consisting typically of images or text that do not typically reside in a RDBMS. Learning on purely relational data is usually conducted on smaller data sets with smaller models, but can still scale to fairly large numbers.

There are two main approaches for distributing machine learning across machines: model parallel and data parallel [44]. In the data parallel approach, data is partitioned across the different worker nodes, while the same algorithm runs on all of them. In most systems, it is assumed that the data is in a shared file system and the worker nodes can load any data they need. In this case, the same model should be used in all the machines, either through the workers accessing a central model located in a server, or the model being replicated in all the workers. The model parallel approach distributes the model instead of the data, which is important when the model is very large, and thus less relevant for learning on data residing in a RDBMS. Each machine has access to the entire dataset, and works on only a part of the model. These two techniques can also be combined for an approach where both the data and the model are distributed across several machines.

2.5.1 Distributing Stochastic Gradient Descent

The most natural way to distribute algorithms using stochastic gradient descent is through data parallelism. Each machine runs an iteration on a minibatch of their own partition of data. The

prediction, loss and gradient calculations can all be done simultaneously and independently from each other. The model updates can then either be applied synchronously or asynchronously.

Synchronous parallelism is when all the machines start each iteration synchronously. That is, every machine will wait until every other machine has finished their iteration and applied the model updates before starting on the next iteration. This ensures that the model being used for prediction at the start of the iteration is the same model being used in all machines. However, this results in the straggler problem, where all machines must wait until the slowest machine has finished its iteration. As the number of machines increase, the problem worsens as the possibility of having a straggler machine increases.

Asynchronous parallelism negates this issue entirely by allowing every machine to start its next iteration as soon as the previous one finishes. Even so, asynchronous parallelism introduces its own suite of problems. A slow machine could be iterating on a stale model and returning updates that are no longer applicable to the current model. A fast machine containing biased data could transform the model into one that is not applicable to the entire dataset. Models may exhibit slower convergence rates. However, this form of parallelism can still be useful in certain circumstances, such as if the model is sparsely updated – that is, if individual iterations performed by machines only access and update a small subset of the model parameters at a time [39].

Stale Synchronous Parallelism (SSP) [21] is an attempt to bridge the gap between these two forms of parallelism. SSP is similar to synchronous parallelism, but allows machines to read models that are stale. The staleness of the model will be bounded by a user-defined constant. This allows machines to work without needing to wait for slower machines, while at the same time providing better correctness guarantees than asynchronous parallelism.

2.5.2 Implementations of Distributed SGD

There are several ways model parameters can be managed across the nodes for SGD. In this section, we present several implementations of SGD across different frameworks.

2.5.2.1 Parameter Server

First introduced by Smola and Narayanamurthy [42] for distributed training of topic models, the parameter server architecture has been generalized for distributed training of many different model

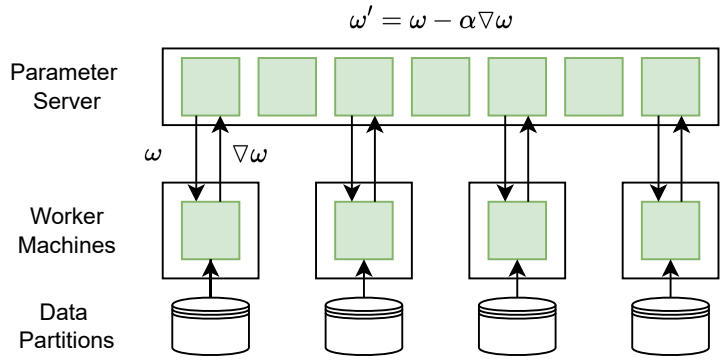

Figure 2.5: Parameter server architecture as described by Dean et al. [8]

architectures. Parameter servers [27] are servers that maintain the global model accessed by the worker machines. While the bulk of the computation is handled by the worker machines, the parameter servers merely aggregate updates sent by the workers and provide synchronization if necessary. To this end, the parameter server holds a global view of the model which all the other worker machines can access.

Communication between the server and the computation machines use two operations: push and pull. Pushing sends locally calculated gradients to the server to be applied to the global model, whereas pulling retrieves the updated model parameters for use in the next iteration – as described in figure 2.5. Here, the model parameters are labelled as ω, the gradient update as $\nabla\omega$ and the learning rate as α.

Additionally, parameter servers are implemented as key-value stores – this allows the push and pull of only a subset of the model. In this manner, parameter servers support both model parallel and data parallel approaches to distributed machine learning and minimizes network load by only communicating necessary data. This is useful in cases where each iteration only works on a small subset of the model, such as in matrix factorization (described in section 2.4.3) where each iteration would only use and update parts of the model according to the data batch it trains on in each iteration. In the matrix factorization example described in 2.4.3, an iteration only requires the embeddings for the specific users and movies present in the minibatch it is working on. Thus, a worker machine running this iteration would only pull the vector embeddings for those specific users and movies, and only send back the updates for those embeddings.

2.5.2.2 AllReduce Machine Learning

The parameter server approach is contrasted with the AllReduce approach. Allreduce is a communication pattern used in distributed systems which expects all participants in the system to produce some partial result, upon which an aggregation is performed to produce a final result, and and each participant receives this final result. While model updates are performed at a single site in the parameter server approach, AllReduce instead distributes the updates as well. Instead of sending all parameter updates to a single server where aggregation is performed, AllReduce instead has worker machines broadcast parameter updates to all fellow workers. Each worker, having a replica of the model, must then apply the updates in sequence. More efficient implementations of AllReduce, such as Ring AllReduce or Tree AllReduce, capitalize on specific network topologies to further optimize this operation [35]. However, AllReduce is generally a synchronous algorithm – every worker must submit their gradient update before the entire system can proceed. Thus, AllReduce works only for synchonous parallelism, and works well for systems were workers exhibit small variance in performance.

2.5.2.3 Distributed PyTorch

PyTorch [34] supports distribution across GPUs and machines. It allows for both model parallel and data parallel distributed machine learning using their Remote Procedure Call (RPC) [7] and DistributedDataParallel [28] libraries. Unlike MapReduce and Spark, these procedures do not support distributed data processing methods, and are solely optimized for distributed machine learning. With these distributed libraries, PyTorch attempts to maintain a similar code structure to its non-distributed variant so as to simplify the process of distribution. Both implementations start with a PyTorch process being run on each worker machine initializing the process group. Each machine is assigned a rank, and then discover each other by accessing the machine with rank 0.

The PyTorch RPC framework can wrap model parameters into remote references which can be accessed by workers. The remote reference (`RRef`) is a wrapper that can be created around any tensor object, and shared to remote workers. Operations done on tensors referenced through `RRef` can be done synchronously, in that the operation blocks until value is returned, asynchronously, which immediately returns a future for the value that will be returned, or remotely, which is done asyn-

chronously and returns a remote reference to the return value. The RPC library uses `dist_autograd` in place of PyTorch's native `autograd`, which extends the backward pass to parameters that exist on different machines and are accessed through `RRefs`. Similarly, a distributed optimizer provided by PyTorch is used in place of the original – it takes `RRefs` in place of model parameters, and each worker applies the optimizer locally for each parameter. The optimizer applies only one set of gradients at a time, with no guaranteed ordering across different machines. Thus, the main differences between a non-distributed PyTorch model and one that uses the RPC framework is that each machine must acknowledge which parameters they contain, and obtain a reference to any remote parameters through `RRef`. The loss and optimizer functions are replaced with distributed versions, and each iteration must be run under the `dist_autograd` context. The RPC framework is ideal for model parallel training as it allows model parameters to exist on different machines and the gradient to be calculated across all of them. This framework can also support the parameter server model of distributing machine learning [43], where all the parameters are on the parameter sever node, and the worker nodes access them via `RRef`s.

On the other hand, the PyTorch DistributedDataParallel library supports data parallel training. This library uses model replication across computation nodes and supports synchronous parallel execution through broadcasting gradient updates and AllReduce. Models are created in each worker, then wrapped in the `torch.nn.parallel.DistributedDataParallel` class, which will synchronize it across workers in the same process group. Instead of giving the base model parameters to the optimizer, it takes instead the parameters provided by the `DistributedDataParallel` wrapper. Gradients are broadcasted and applied across all workers transparently. For that, PyTorch uses the Gloo or NCCL libraries that implement the AllReduce API.

2.5.2.4 Other implementations

As distributed machine learning is a highly studied field, there are many more implementations other than what is described here.

Distributed machine learning can take advantage of pre-existing architectures designed for large-scale distributed data processing. Many of these data processing architectures are built on distributed file stores, such as the Google File System (GFS) [17] or the Hadoop File System (HDFS) [36]. MapReduce [9] is a well-known data processing framework that uses such systems. Each worker

node starts off with applying a map function to its local data. Each data record will be assigned a key during map phase. During the shuffle phase, the workers redistribute the data based on these keys such that all records with the same key are placed in the same node. The reduce phase is next, where each worker performs some aggregation on the new partitions of data. Naturally, the reduce phase depends on completion of the previous two phases, and thus is a strictly synchronous operation. Machine learning algorithms using synchronous parallelism can be expressed using the MapReduce paradigm [44].

Apache Spark [1] is yet another data processing system, optimized for iterative tasks. While MapReduce makes heavy use of its access to a distributed file system to write intermediate results, Spark instead focuses on executing tasks directly in memory. As many linear algebra and machine learning tasks are iterative, this approach saves much time by eliminating disk reads and writes. Furthermore, MLlib [32] is a distributed machine learning library built for Spark which implements scalable versions of several common machine learning algorithms. However, Spark itself still runs on a distributed file system.

Further studies have gone into improving parameter servers, such as NuPS [40], a parameter server designed for non-uniform parameter access. In NuPS, parameters are either replicated across all computation nodes, or pulled from a specific home node depending on how much they are accessed. Parameters that are accessed often by all nodes are replicated and given bounded staleness guarantees, whereas parameters that are accessed by one node but not often by others are relocated to the node where it is most accessed and from where other nodes must request to access it.

2.5.3 Clustering

Clustering algorithms are very different in distributed settings. Centralized clustering algorithms often assume a complete view of the entire dataset, and thus are hard to directly translate to a distributed setting. While there are many distributed clustering algorithms available, we will focus on Density Based Distributed Clustering (DBDC) for this thesis.

DBDC [23] is a multi-step algorithm that assumes all data points to reside in different machines. Unlike the previously explored distributed stochastic gradient descent, DBDC is not an iterative algorithm. It goes through a fixed number of steps – a local clustering on each worker machine, followed by determination of a global clustering, then finally application of the global clustering back

on the local data. Furthermore, DBDC assumes data to be residing in different worker machines and the existence of a global site which can access all worker machines. The algorithm starts off by having every machine do an independent local clustering in parallel. Each machine determines a set of local representatives for each local cluster and transfers these representatives to a central site, where the global model is determined. The global clustering is then communicated back to the other machines to update the cluster labelling of the data points residing on those machines.

The first local clustering is determined by each machine running DBSCAN with a given ϵ and $minPts$. A specific set S of core points are extracted from the core points found by DBSCAN. The set S comprises of points that are not in the ϵ-neighbourhood of any other point in S, and all core points in the cluster can be found in the ϵ-neighbourhood of a point in S. This set S can be found by running k-means with $k = |S|$ on each cluster, where each centroid found is a point in S. Each point r in S is then assigned an ϵ_r value, representing the neighbourhood around the point which it represents. Thus, each local clustering model is a set of pairs (r, ϵ_r) that act as representatives.

The set of representatives are sent to a central system to determine the global model. DBSCAN is run again on the representatives from all the machines with $minPts_{global} = 2$ and $\epsilon_{global} = 2 * max(\epsilon_{local})$. Clusters represented by the local representatives may be merged if they are sufficiently close enough. The global model, consisting of representative points, their clustering labels and ϵ, are sent back to each of the machines to update their local clustering labels.

2.6 Advanced In-Database Analytics (AIDA)

Advanced In-Database Analytics (AIDA) is a system introduced by Joseph D'Silva [11] that runs inside the RDBMS and allows the user to perform data operations without incurring the overhead of moving data out of the database. It runs on a Python interpreter that is embedded in the database so that the user can take advantage of optimizations provided by the RDBMS for database queries while simultaneously making use of well known data science and machine learning libraries that Python provides.

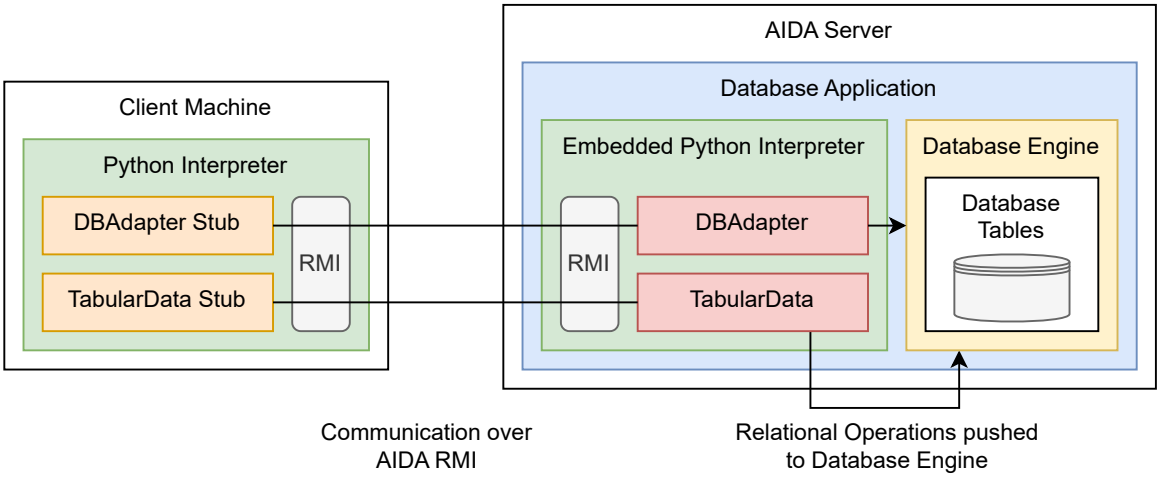

Figure 2.6: Diagram of AIDA architecture

2.6.1 AIDA structure

The AIDA framework uses a client-server architecture. The user uses the client-side API to connect to the AIDA server that is embedded in the database through the use of RMI. This allows the user to send commands and computation to the AIDA process running in the database server.

When the user on the client side connects to AIDA on the server side, AIDA creates a database connection to the database it is currently running in. This database connection object sits on the server side, but a proxy is shipped to the client space so that the client can call functions on it and access the data inside the database remotely. All tasks that the server performs are initialized by the client through calling on server objects using their proxies residing on the client site. This architecture is described in Figure 2.6 – the `DBAdapter` object in the AIDA server and object stub on the client side representing the database connection object.

For instance, in the code segment 2.2, the user first connects to the AIDA server using the `AIDA.connect` function. That function returns the stub of the database connection object to the user, and the user then calls the `_tables` function on it to acquire a list of all tables in the database. The stub gets the function name that the user calls and ships it over to the actual object residing on the server side, where a database query is then executed in the database, and the result is shipped back to the client side.

Indeed, all communication between the client and the AIDA server must be achieved through the RMI communication paradigm supported by AIDA. RMI uses a blocking request-reply structure,

```
from aida.aida import *

hostname = "localhost"; database = "db"; username="user";
password = "password"; jobname = "job";
dw = AIDA.connect(hostname, database, username, password, jobname)
print(dw._tables())
```

Listing 2.2: Example of AIDA database connection usage.

meaning that execution on the AIDA server must be initiated by the client through a request, and the client blocks until a reply from the server is received. We will use this RMI primitive in our distributed architecture to exchange data, be it part of the database tables or model parameters, between the components of AIDA.

2.6.2 TabularData

TabularData objects in AIDA are abstractions of data in the database system. They can represent data that exists in several different formats: as database tables, as numpy matrices or in form of dictionaries. Conceptually, they are similar to other tabular data abstractions such as Pandas Dataframes [31] or Spark Dataframes [1], both of which in turn are inspired by R Dataframes [38]. AIDA TabularData objects differ from those by residing on the server-side AIDA system, and can be accessed through a stub on the client side. Any operations done on the client-side stub will be shipped to the server-side object, where they will be executed within the database environment, as described in Figure 2.6. Relational operations on data represented by the TabularData object will be translated into its equivalent SQL commands, and executed by the database engine. This ensures execution of any relational operations can take advantage of the many optimizations the database engine provides for executing such operations.

The TabularData abstraction exposes an Object Relational Mapping (ORM) API so that the user can perform relational queries on the object directly. The TabularData object also supports linear algebra operations. Relational queries and linear algebra operations can be interwoven at the user's will. Every operation done on a TabularData object returns a new TabularData object containing the result of the operation. The data contained in a TabularData object may not be materialized yet; relational operations are performed lazily and only when the user explicitly requests

for the results of the operations will they be performed. In particular, if a user submits several relational operations and then asks for matrix data, AIDA will combine the relational operations into a single SQL statement to allow the database to perform more sophisticated query optimization. AIDA also supports combining TabularData objects – the `.vstack` method on a TabularData object allows rows of two or more TabularData objects to be concatenated, and the `.hstack` call concatenates the columns of multiple TabularData objects.

Additionally, TabularData supports NumPy matrix representation of the data contained within. This is useful when performing linear algebraic operations. To transform the data into a NumPy matrix format, the client merely needs to call the `.matrix` attribute of the TabularData object. The attribute will first materialize the data, if it has not been materialized already, and then transform it into the NumPy matrix format.

Thus, there are in the end three states a TabularData object can exist in. The first occurs when a client accesses a database table – the AIDA server will create a TabularData representation of this table, but the data itself remains in the database and will not be materialized yet. Indeed, the first and all subsequent relational operations done on this data will not trigger the materialization of the data until the user explicitly requests for the data to be materialized through either the `.cdata`, the `.matrix` call, or any linear algebra operation. While the data has not been materialized, every operation done on the TabularData object will return a new TabularData object containing a reference to the original object, and the operation being performed. The operation will not actually be performed until it is time for materialization, in which case the TabularData object can trace its lineage through the references and construct a complete query from all the operations.

The second state a TabularData object can be in is after the data has been materialized. Then, the client can access the result of all relational operations performed on the original database table. It is stored in the form of an Python ordered dictionary, where the keys of the dictionary are the column names and the values of the dictionary the column data in NumPy vector form. This is known as columnar storage – the data is stored in columns as opposed to rows. This makes it easy for data to transfer in and out of the database engine, as AIDA uses the MonetDB database [22], which also stores its tables in columnar format. Indeed, MonetDB supports *zero-copy* data transfer – queries done by the MonetDB database engine returns results to the embedded Python application without creating a copy of it. Since MonetDB stores objects in the same format as NumPy arrays,

the ordered dictionary format merely references the underlying data, instead of making a copy of it.

The final state in which the TabularData can store its data is the NumPy matrix form. When requested, the TabularData object will materialize its data in the Ordered Dictionary form as described earlier, then transform that into a NumPy matrix by stacking its columns horizontally. In this fashion, linear algebraic operations such as matrix multiplications and transposes can be done on TabularData objects. Certain linear algebra operations will cause a TabularData object to automatically materialize its data in matrix form.

```
1 td = dw.lr_data
2 targets = td.project(('y'))
3
4 x = td.project(('x1', 'x2', 'x3'))
5 biases = db._ones(x.shape[0])
6 data = biases.hstack(x)
7 print(data.cdata)
```

Listing 2.3: Example of TabularData object usage

In code segment 2.3, the user accesses the relational table called `lr_data` located in the database through the database connection object stub at line 1. This invites the server to create a TabularData object referencing the `lr_data` table in the database. Then, the user projects only columns they wish to retrieve from the data – in this case, the column named `'y'` and the columns named `'x1'`, `'x2'`, `'x3'`. The result of these two projections are returned as TabularData objects, the stubs of which are sent back to the client side and stored in the variables `targets` and `data` respectively. The database connection object offers a method called `_ones`, which returns a TabularData object consisting of ones in the shape provided. The shape provided is the number of rows in the `x` table, resulting in a single-column TabularData object with the same number of rows as `x`, containing all ones. Since calling `shape` is a linear algebra operation, and not a relational one, the data for `x` is materialized. At this time point, the data resides in a dictionary stored by the `x` TabularData object – it has not yet been sent to the user. This column is then appended to the `x` table and the result stored in the variable `data`. At the end, the user calls the `cdata` property on `data`, which causes the execution of all the previous operations since the last materialization – in this case, just `hstack`. The `cdata` call also retrieves the newly-materialized data to the client side, where it can

be printed.

2.6.3 Remote Execution Operator

AIDA also provides the user with functionality to ship their own functions to be executed on the server side through use of the `_X` operator, provided by the database connection object. This allows the user to implement their own operations on the remote data that may not be supported by the AIDA API. It also allows the user to perform many operations using only one RMI call by wrapping the operations in a single function to be shipped and executed on the server side. Libraries can also be imported and used in the remote function, as long as the library is installed on the server side.

The function defined by the user must take the database connection object as its first argument. The user can define any other arguments, and the function can return any object. The database connection object also acts as a workspace in which the user can add their own variables, possibly containing complex objects, that are persistent and can be accessed from either the remote function or the client side workspace. Thus, the user can set persistent variables in the database workspace, and access them from within the remote execution function. It also allows the remote execution function to access database tables and other functionalities of the database connection object.

```
1  def linear_regression(db, iterations, batch_size):
2      import numpy as np
3      x = db.data.matrix.T
4      y = db.targets.matrix
5      learning_rate = 0.002
6      for i in range(iterations):
7          batch = np.random.choice(x.shape[0], batch_size, replace=False)
8          batch_x = x[batch, :]
9          batch_y = y[batch]
10         preds = batch_x @ weights.T
11         loss = 2 * (((preds - batch_y).T @ batch_x) / preds.shape[0])
12         db.model = db.model - (learning_rate * loss)
13
14 dw.model = np.ones((1, dw.data.shape[1]))
15 dw._X(linear_regression, 5000, 32)
```

Listing 2.4: Example of custom remote function

The code segment 2.4 provides an example of a user-defined function that is shipped and executed on the server side. Here, the user creates a function `linear_regression` to perform some

linear regression iterations on the data provided by the TabularData object `db.data` and the labels provided by `db.targets`. The database connection object is storing a linear regression model in `db.model`, which is updated every iteration in the function. The function is executed at the end using the `db._X` operator, where the function itself, along with its arguments, are shipped from the client space to the server space and executed.

2.6.4 Data Transferring in AIDA

While AIDA aims to push computation to the data side, it also supports operations for pulling and pushing data to and from the database server. This is useful for cases where data that wishes to interact reside on different nodes – they can then be transferred into the same AIDA environment. These operations are also executed using the AIDA RMI message passing system – the client can request data to be pulled to the client side by calling `.cdata` on the TabularData stub, and can push data to the server side using the `._L` method provided by the `DBAdapter` object.

As stated previously, the `cdata` call on TabularData objects materialize the data. It returns the columnar dictionary format for the data, which will then be serialized and transmitted to the client site by the RMI functionality. Using this operation, clients can pull data from the AIDA server.

Additionally, the database connection implements the `._L` load function, used to load data at the server. It will take a dictionary or NumPy array, and return a TabularData object stored at the server. This method can thus be used to push data to the server, and later be able to manipulate the data using the TabularData API, and have it interact with other data at the server.

3

Distributed AIDA Architecture

Recall that the goal of D-AIDA is to be an in-database framework supporting both distributed query processing and distributed machine learning in a modular manner. In this chapter, we present the overall architecture of D-AIDA and the data abstraction it implements to provide a global view of partitioned data table to the client and facilitate both these forms of distributed data processing.

3.1 Distributed AIDA Architecture

Extending the AIDA system to be distributed requires consideration of the distributed architecture and what we aim to do with it. So far, we assume all the data is horizontally partitioned across several database nodes, each running their own AIDA system. However, we believe this architecture can also be extended to support vertical partitioning. Recall as well that each AIDA server lives within the Python environment embedded in the RDBMS – in this case, MonetDB. When a user

accesses a table, they should be presented with a global view of the data table, with the precise distribution of the data hidden. Furthermore, the user should be able to perform operations and write algorithms without needing to manually push computation to where the data resides.

A new middleware AIDA system is introduced to manage the communication with the client and with the database nodes. As discussed in section 2.3, middleware architectures allow for the performance of distributed joins in systems where computation – the query – is shipped to the data location. Moreover, the middleware architecture allows a natural implementation of the parameter server architecture described in section 2.5.2.1 for distributed machine learning. A middleware server with knowledge of the data distribution can provide both a global view of the data, access to operations on the data that may require data sharing, as well as a common access point for shared global data, such as models. For the rest of this paper, the sites where the data resides will be called the *database* or *worker* nodes, whereas the middleware site will be known as the *middleware* node.

Instead of connecting to each of the AIDA servers where the data is hosted, the user needs only to connect to the middleware server. The infrastructure supporting this is provided by the RMI functions inherent to AIDA, and is outlined in figure 3.1. This middleware AIDA system acts as a client to the database AIDA servers, propagating client operations to the database servers where they will be executed. Since each AIDA client-server relationship creates a `DBAdapter` object in the AIDA server as well as a `DBAdapter` stub in the client space, the middleware AIDA server contains stubs to the AIDA database servers. Meta-information detailing where partitions of each table in the database lie is stored in the middleware. Like the original AIDA system, the middleware AIDA server creates a database adapter object and returns it as a stub to the client. The database connector also supports frameworks for pushing user-defined algorithms to the database nodes, which will be further expanded upon in Chapter 5.

Meta-information keeps track of the database nodes and the data each of them maintains. This information is stored in a `SQLite` [20] database file. `SQLite` is a light-weight database that stores all its data in ordinary files. It does not run its own server process, but supports data access through the use of libraries in various programming languages. The meta-information contains two tables: `servers` and `tables`. The `servers` table contains only one attribute: the `hostname` of the AIDA servers in the cluster. The `tables` table contains two attributes: each row is a tuple (`table_name`, `hosts`) where `table_name` is the name of the partitioned data table, and `hosts` is a foreign key

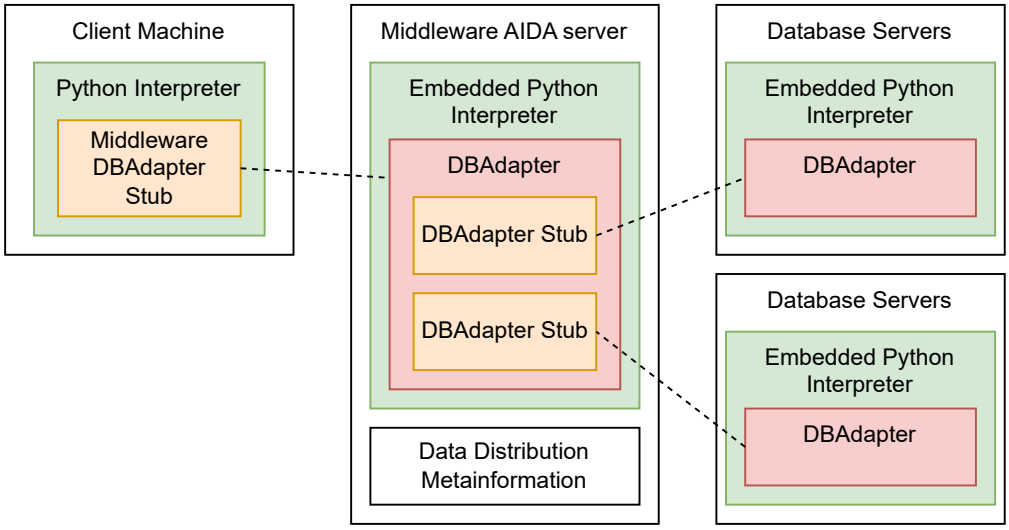

Figure 3.1: Distributed AIDA architecture.

referencing `servers` describing the location of a partition of the table. Because we assume the data to be horizontally partitioned, we expect each partition of the dataset to share the name of the global dataset and have the same features. That is, our current set-up allows a data node to not have all rows of a table, but it should have all the columns.

Upon connecting to the middleware server, the middleware database connection object is initialized. It establishes a connection with all the servers in the cluster, as outlined by the `servers` table in the `SQLite` meta-information database. It also initializes a thread pool for future parallel distributed tasks. The middleware AIDA server is also embedded in an RDBMS, although the database is empty as we assume the data is partitioned across other AIDA servers. This database engine can be used for operations that require data movement to the middleware, such as certain aggregations and the `DISTINCT` SQL operation that are needed for post-processing of distributed relational operators.

3.2 DistTabularData

In AIDA, the TabularData abstraction enables the client to perform both relational and linear algebra operations on relational data. It also allows a wide range of data transformations. In D-AIDA, we aim to support this in a distributed setting. For this, D-AIDA implements the DistTabularData abstraction.

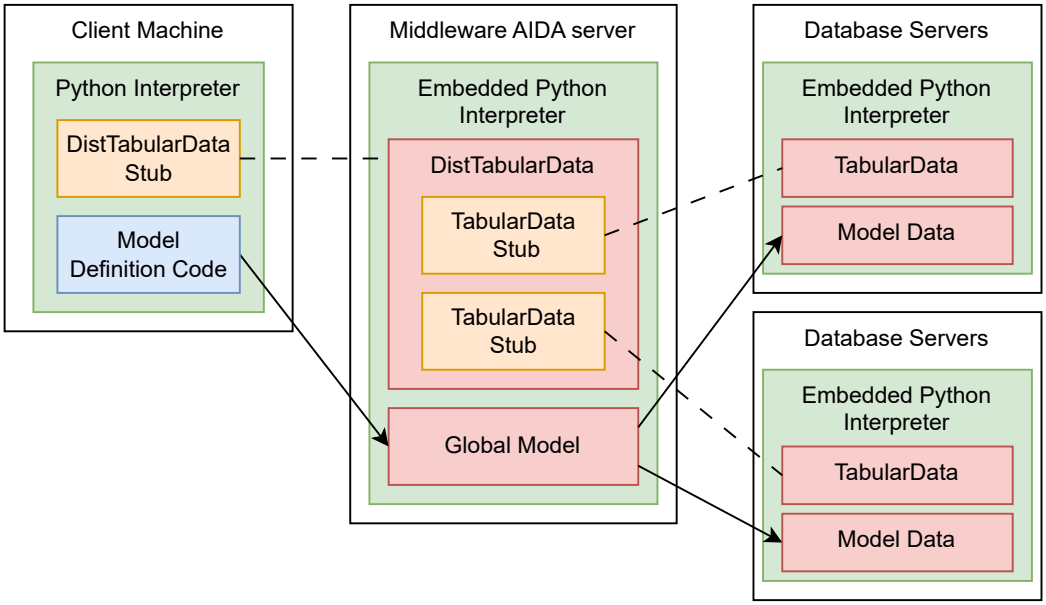

Figure 3.2: Data distribution for D-AIDA.

Like the original AIDA connection, tables in the database can be accessed by referring to the table by name using the database connection object, `DBAdapter`. Instead of consulting the database, the AIDA middleware server consults the meta-information it has on hand to learn where partitions of the table are hosted. For example, if the table `clustering_data` is accessed through the middleware `DBAdapter` stub, the middleware finds all tuples with `table_name='clustering_data'` in the `tables` table in the `SQLite` database file. The `hosts` attribute of these tuples indicate the servers on which a partition of the data resides. Each of these partitions can be accessed normally by the middleware, since the middleware server holds references to the database workspace objects supplied by each server. From here, the middleware offers a holistic view of such a distributed table by implementing the DistTabularData abstraction, which is a wrapper around a list of TabularData objects and their home connections. Like TabularData, the actual DistTabularData object remains at the middleware, and the user only receives a stub when accessing it.

Figure 3.2 presents how the DistTabularData object, along with its component TabularData objects, are distributed in the D-AIDA system. Additionally, it shows how machine learning models, defined at the client, will be held at the middleware and sent to the database servers for work. This will be further expanded upon in Chapter 5.

So far, DistTabularData supports a subset of the operations TabularData objects support. In

most cases, these operations will be pushed to the database nodes that hold the data. To manage operations done in parallel on the database nodes, the DistTabularData uses a thread pool initialized by the middleware database connection object to execute remote operations on the TabularData objects it references. These operations are done transparently to the client; the client will not have access to any of the individual TabularData stubs or direct connections to a database server. Each DistTabularData object holds a dictionary whose keys are the database base connection object stubs to the database nodes, and whose values are the TabularData stubs referencing the database table partition at those nodes. This dictionary can be accessed by the `tabular_datas` property of a DistTabularData object, which is hidden from the client.

4

Distributed Query Processing

Distributed query processing is a well-explored field. In this chapter, we present a proof-of-concept that D-AIDA's architecture allows for a convenient implementation of distributed query operators. We have implemented the most common operators and show in this chapter how their execution is coordinated across the D-AIDA components.

In the majority of these cases, the execution of the relational operators are pushed down to the database nodes where the data resides. Each of these nodes hosts an AIDA database server which the middleware AIDA server will interact with. When pushing the relational operations to the AIDA servers, they will be executed by the RDBMS in which the AIDA servers are embedded. In this fashion, the execution is performed near data and by the optimized relational query engine. For now, the relational operators implemented in D-AIDA serve as a simplified proof of concept. Thus, many are executed immediately instead of the lazy approach followed by the original AIDA implementation.

4.1 Base Operators

DistTabularData objects support most of the distributed relational operations, outlined in section 2.3. Since DistTabularData inherits from the original AIDA TabularData object, many of the method signatures remain identical, with some exceptions. The base operators that execute on a single table remain quite simple and will be listed here:

1. `.project(projcols)`: This directly inherits from the TabularData API. This method performs a projection on the list of columns `projcols`, passed as the parameter. When this is called on a DistTabularData, the object will internally call this same function with the same arguments on all `tabular_datas` it has a reference to in parallel. Each TabularData object then returns a stub to the result of the operation; these stubs are collected and a new DistTabularData object referencing these is returned to the client.

2. `.filter(*selcols)`: This operation similarily inherits from the TabularData API. This is a selection operation with filter conditions `selcols`. Like the projection operation, the DistTabularData object merely forwards the method call along with the parameters to the TabularData objects in the database servers.

3. Simple aggregate functions like `sum`, `count` and `average` go through similar processes, ending with an aggregation at the middleware server. The `sum` and `count` methods are both executed independently at each database server, and the results summed at the middleware and returned to the client. The `average` method performs the distributed `sum` and `count`, then divides the result of the former by the result of the latter at the middleware. In the current implementation, the middleware does not use its database engine for these simple aggregations.

4. Complex `.aggregate(projcols, groupcols)` operations are also supported, although with a non-optimal method. The `projcols` parameter takes the column names and requested aggregations of the output, while the `groupcols` parameter indicates the columns on which the user wishes to perform a `GROUP BY` operation. For instance, `dw.registration.aggregate(('cou⌋ rse_code', {COUNT('student_id'): 'num_students'}, {AVG('grade'): 'average'}), ⌋ ('course_code'))` is the equivalent of the SQL statement `SELECT course_code, COUNT(st⌋`

udent_id) AS num_students, AVG(grade) AS average FROM REGISTRATION GROUP BY co
urse_code. This query returns the number of students and the class average for every course that has been registered. Complicated aggregation queries with group bys such as these are fully executed at the middleware server. The data in the table is pulled from each database server and loaded in the database at the middleware – a new DistTabularData object, containing only a reference to the database connection object at the middleware and the result of the query in TabularData format, is returned to the client. An optimal implementation would perform the aggregations at the database nodes, and perform a final aggregation of the results on a per-value and per-column basis at the middleware.

5. Like group bys, the DISTINCT and ORDER BY operations are queries that require sorting of the global data. Thus, execution for both of these queries also occur at the middleware after pulling the data from the database servers.

6. The cdata performs a special function for TabularData objects – it will initialize transmitting data from a server AIDA process to the client site. DistTabularData also supports this function: calling cdata on DistTabularData objects will also call cdata on the TabularData objects at the database nodes. The middleware concatenates all the tuples of data transmitted, and returns that data to the client.

To conclude, there are three main algorithms at use for these operations. The first algorithm involves the middleware server propagating the query to be independently executed at each of the nodes, and returning the concatenation of the results. The second is much the same, only requiring an additional aggregation at the end. The final algorithm merely pulls all the data towards the middleware, where the query will be executed by the middleware RDBMS engine set aside for that specific purpose. Kossman [26] laid out how operators like group by or order by could be distributed across the data nodes with some post-processing at the middleware. Given that D-AIDA has the middleware embedded in a database already, implementing this optimized version is possible.

4.2 Distributed Joins in AIDA

Joins are highly complex queries involving multiple tables. Distributing joins do not come as naturally as simpler operations – data must be able to move around in the system and interact with other data. Thus, in this section, we will describe the two join algorithms provided by a DistTabularData object. Both joins are accessed through the same `join` call on a DistTabularData object. So far, it is up to the user to decide which join argument to use by providing an optional argument. The default is broadcast join. Currently, only inner joins are supported. The join call of the DistTabularData object is similar to that of the TabularData object, with the exception of one argument.

The DistTabularData `join` method takes the following as arguments:

1. `otherTable`: The other DistTabularData object on which to perform the join.
2. `src1joincols`: The names of the columns of the DistTabularData object on which the join is performed – the join attributes.
3. `src2joincols`: The names of the columns of the other DistTabularData object on which the join is performed.
4. `cols1`: The columns of the first DistTabularData object that will be returned in the output, equivalent to performing a projection on the columns of the first table along with the join.
5. `cols2`: The columns of the second DistTabularData object that will be returned in the output, equivalent to performing a projection on the columns of the second table along with the join.
6. `hash_join`: True or False, depending on whether the client wishes to use broadcast or hash join. This argument is unique to the DistTabularData join and defaults to False.

In contrast to the above, the join call offered by TabularData does not have the sixth argument, but an argument that decides the type of join to be performed – inner joins, cross joins and a variety of outer joins are all options. Since DistTabularData only supports inner joins, this argument has been dropped.

4.2.1 Broadcast Join

Recall from section 2.3 that broadcast joins are the simplest algorithm for distributed joins. One table, optimally the larger one, remains partitioned, while the other table is broadcasted to all database nodes. The join is performed between each partition of the first table and the entirety of the second table at each database node, and the concatenation of the local result sets form the end result. In D-AIDA, it is up to the user to decide which table will be broadcasted. The join should always be performed on the table that remains partitioned, with the broadcasted table forming the first argument.

Conceptually, the implementation of the broadcast join in D-AIDA is a three phase process. Consider a broadcast join between the first table `A` and the second table `B`, which will be broadcasted. First, the TabularData stubs for each partition of `B` are sent to each database node holding a partition of the first table. These nodes then use these stubs to pull all partitions of `B`. The partitions are concatenated together to create a copy of `B` in each database node. In the second step, a join is performed between each copy of `B` and partition of `A`. This join is performed by the database engine on each node where a partition of `A` resides. Finally, the results of the join are sent to the middleware, where they will be aggregated to form the final result.

For a visualization on where the functions are executed, consult figure 4.1 for a sequence diagram of the broadcast join process. For clarity's sake, the `external_join` on the second worker is omitted, but should be noted that it is executed in parallel with the `external_join` on the first worker.

The implementation for this join in the distributed AIDA framework is presented as algorithm 2. The user first calls `join` function. If the `hash_join` parameter is set to false, the algorithm continues on to the function outlined in lines 6-10 of algorithm 2. The function `external_join` is defined at the middleware and shipped to each of the database nodes to be executed in parallel. This function, outlined in lines 1-5, takes as input the TabularData object representing the partition of the first database table that resides at the same site the join is executed on, and all the TabularData stubs belonging to the second table. Using these TabularData references, each database node pulls each database partition of the second table from the other database nodes, and recreates the entire second table. Then, it performs the join between its partition of the first table and the entirety of the second table. The second table is loaded into MonetDB, where the partition of the first

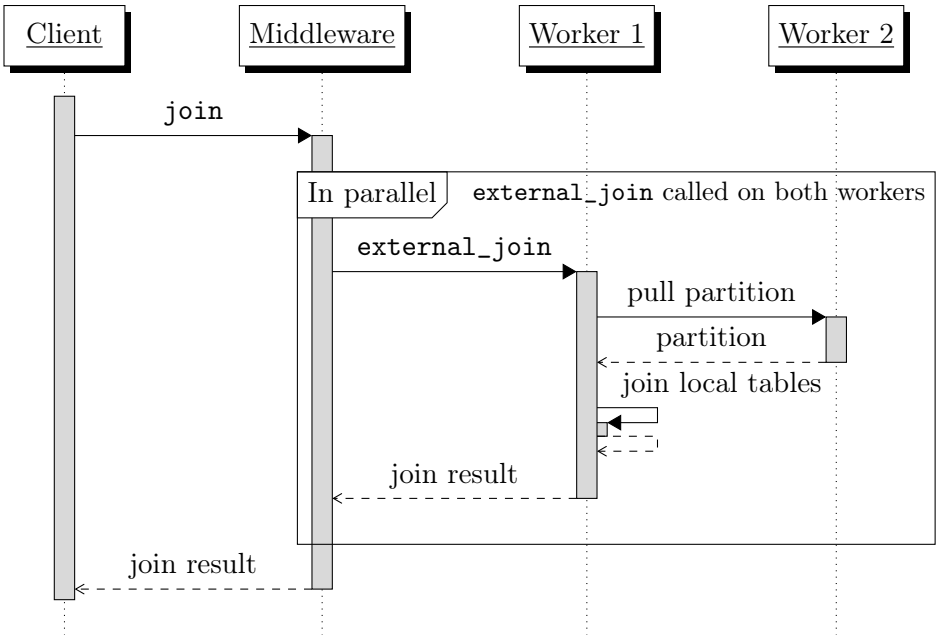

Figure 4.1: Sequence diagram for distributed broadcast join

table already resides, and the join is executed by the MonetDB database engine. The resultant TabularData object is then returned to the middleware, which will aggregate it along with the results from the other sites, and return a new DistTabularData object to the user. Note that when a database node pulls a table partition from a different database node, it becomes a client to the other node without going through the middleware. Indeed, the pull action is a simple execution of the `cdata` call on the TabularData stub pointing to a TabularData object on another database node.

Algorithm 2 Broadcast join algorithm for DistTabularData

```
1  function EXTERNAL_JOIN(db, firstTable, secondTable, src1joincols, src2joincols, cols1, cols2)
2      tempTable ← {}
3      for partition in secondTable in parallel do
4          partition ← partition.cdata
5          concatenate(tempTable, partition)
       return firstTable.join(tempTable, src1joincols, src2joincols, cols1, cols2)

6  function DISTTABULARDATA.JOIN(otherTable, src1joincols, src2joincols)
7      result ← {}
8      for con, table in this.tabular_datas in parallel do
9          r = con._X(external_join, table, otherTable.tabular_datas, src1joincols, src2joincols,
   cols1, cols2)
10         result[con] = table
        return DistTabularData(result)
```

4.2.2 Distributed Hash Join

Described in detail in section 2.3, distributed hash joins involve the re-shuffling of data across database nodes such that all data having the same join attribute in both tables are moved to the same site. The implementation of hash join in distributed AIDA makes use of AIDA's ability to push and pull data between servers.

The hash join algorithm implemented in AIDA is a three-step process. First, each database node hashes and re-partitions the data they are responsible for. These new partitions are pushed, using the AIDA `._L` operator to the other worker nodes. This first step is executed in a function called `hash_partition`, that the middleware pushes and executes on each worker node using the `._X` AIDA function. Next, each worker node now holds new partitions of each table from all the other workers – the partitions belonging to the same table must be consolidated into one. Finally, with the newly consolidated table partitions, the local joins between the new table partitions are execution on each worker node, and the results sent to the middleware to be aggregated.

The distributed hash join algorithm implemented in D-AIDA is visualized in a sequence diagram in figure 4.2. Like figure 4.1, the helper function `hash_partition` is called on both workers simultaneously, but omitted in the sequence diagram on worker 2 for clarity. As can be seen, the `hash_partition` function uses the AIDA `._L` function to push the new data partitions to the other database nodes.

The hash join algorithm will be triggered when the `hash_join` parameter is set to True. It enters the `hash_join` function shown in algorithm 3 starting from line 11. To begin with, the middleware executes the `hash_partition` function on the TabularData objects referred to by both the DistTabularData objects involved in the join. From the point of view of a single database node, this function – outlined in lines 1-10 – hashes the join attribute of all the tuples in the TabularData object that is passed to it. The hashed tuples are sorted into the same number of buckets as there are machines involved in the join. Each bucket is assigned one of these machines, and the data in the bucket pushed to that machine. This function then returns the stubs of the new partitions. After the new partitions are created on each machine, the `hash_join` function continues on line 20 by consolidating the new partitions into a single table on each worker node. For example, if a client were to join table A and B, both distributed on three worker nodes labelled w_1, w_2, w_3, then after

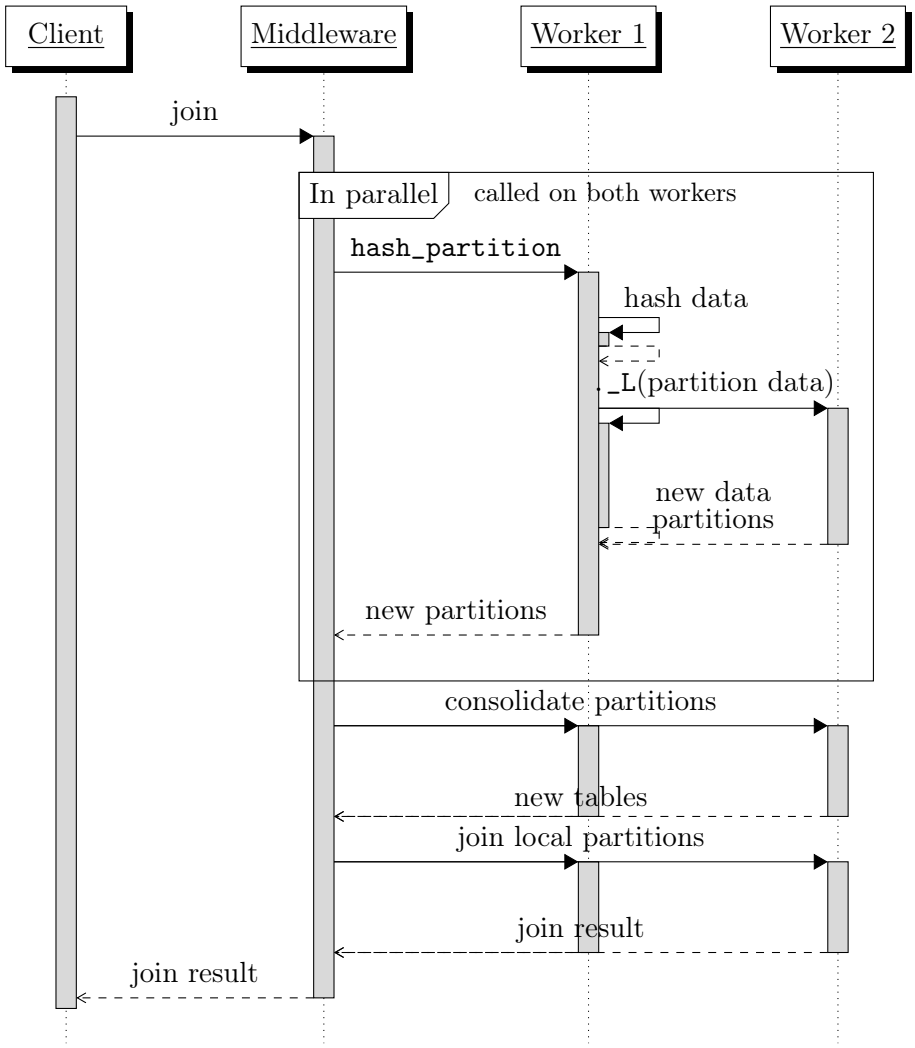

Figure 4.2: Sequence diagram for hash join

Algorithm 3 Hash join algorithm for DistTabularData

```
1  function HASH_PARTITION(db, table, connections, joincols)
2      num_connections ← length of connections
3      partitions ← list of num_connections length
4      for record in table do
5          hashed ← hash(record[joincols])
6          partitions[hashed % num_connections].append(record)
7      table ← {}
8      for i in {0..num_connections-1} in parallel do
9          remote_table ← connections[i]._L(partitions[i])
10         table[connections[i]] ← remote_table
       return table

11 function HASH_JOIN(otherTable, src1joincols, src2joincols, cols1, cols2)
12     ▷ Hashing and redistributing partitions
13     connections ← {this.tabular_datas.keys, otherTable.tabular_datas.keys }
14     redistributed_tables ← {}
15     other_redistributed_tables ← {}
16     for con, table in this.tabular_datas in parallel do
17         redistributed_tables += con._X(hash_partition, connections, src1joincols)
18     for con, table in otherTable.tabular_datas in parallel do
19         other_redistributed_tables += con._X(hash_partition, connections, src2joincols)
20     ▷ Consolidating subpartitions in each node
21     new_table ← {}
22     new_other_table ← {}
23     for c in connections do
24         new_table[c] += concatenate([pt[c] for pt in redistributed_tables])
25         new_other_table[c] += concatenate([pt[c] for pt in other_redistributed_tables])
26     ▷ Joining partitions on the same node
27     result ← {}
28     for c in connections do
29         result[c] = new_table[c].join(new_other_table[c], src1joincols, src2joincols, cols1, cols2)
       return DistTabularData(result)
```

the execution of `hash_partition`, each worker would hold three segments of the new partition. Let A_i be the partition of A on worker w_i. The records in A_i would be hashed and stored into three buckets during `hash_partition` – call these three buckets A_i^1, A_i^2, A_i^3. Each of these three buckets are assigned and pushed to a worker, A_i^1 to w_1, A_i^2 to w_2 and A_i^3 to w_3. Thus, after the execution of `hash_partition`, each worker w_i would hold three new partitions for table A – A_1^i, A_2^i, A_3^i, and the same for table B. These new partitions need to be consolidated into one whole partition of A – call that A_i', which is achieved through concatenating the partitions together. Note that while the concatentation is called at the middleware (lines 24/25), the actual concatenation of the data is done at the database nodes that hold the data. After the consolidation of the local segments for both tables, the two tables are finally joined on their local partitions by the MonetDB database engine, the results of which will be sent to the middleware and forwarded to the client.

4.3 Discussion

Currently, the relational operations implemented by D-AIDA represent only a proof of concept. For a full fledged system, further improvements are required.

For one, better post-processing is required at the middleware for a more optimized implementation of aggregation queries. Currently, simple aggregations are performed outside of the database engine, and complex aggregations require pulling all the data to the middleware. Both of these implementations are unoptimal, and may be even unfeasible in the latter case. Instead, the query needs to be rewritten, to do partial grouping/ordering at the database nodes, then a final grouping/ordering at the middleware.

Additional join implementations could be added as well. Joins are a well-studied and integral part of relational database operations. The two distributed joins implemented by AIDA offer some choice of join optimization to the client, but many other implementations exist. Furthermore, the system could automatically decide which join implementation to use. To optimize this, the middleware would require more meta information regarding the statistics of each database table. Indeed, previous work [45] has shown that the feasibility of systems automatically optimizing distributed joins given table meta information.

Optimized systems might choose to re-order parts of a complex query or otherwise change

portions of a query to return equivalent results in a faster manner, but D-AIDA does not currently support such an optimization due to a lack of lazy evaluation. In the current system, joins and aggregations are immediately executed. The original AIDA implementation generates a linage tree, which it then rewrites into SQL upon evaluation, allowing the database engine to perform optimization. A similar technique could be implemented at the D-AIDA middleware for distributed queries. A database engine capable of optimizing for distributed queries, like those mentioned in Kossman's survey [26], would be useful here.

5

Distributed Machine Learning

D-AIDA provides several frameworks that a user can use to write their machine learning algorithms. The central framework is well-designed for iterative machine learning tasks. The workflow-based framework allows for more freedom in iterative or non-iterative tasks. Finally, the parameter sever framework allows some degree of model-parallelism in designing the tasks.

These frameworks have been chosen because they are best suited for the middleware architecture that distributed AIDA implements. Each of them has the middleware server hold a global model that the client may query after training. Furthermore, these frameworks can be used to implement a variety of distributed machine learning models, particularly those trained on relational data. Indeed, we chose to implement multiple different frameworks instead of a single one in order to increase the level of expressiveness D-AIDA allows for in writing distributed machine learning algorithms. We used these frameworks to evaluate the ease of use in incorporating different algorithmic designs in the D-AIDA ecosystem.

In all of these frameworks, the model and method definitions are written by the user on the client-side python interpreter, and all of it is shipped to the middleware, which will then manage the execution of the methods. All of these frameworks support user-defined models written using whatever library they prefer. Any libraries used by the code, such as NumPy or PyTorch, must be imported at the start of each function. The libraries must also be installed on the server-side embedded Python interpreters. The methods that the client must define are different in each framework, reflecting the differences in execution. Furthermore, users must be cognizant of the format of the data – AIDA natively supports NumPy matrices through the TabularData object, but extra transformations must be performed when dealing with PyTorch tensors.

Additionally, all of these frameworks aim to only transfer data pertaining to the model or gradient updates between the middleware and the worker nodes, not the learning data. Indeed, each of these frameworks require a DistTabularData to act on, which indicates which workers are involved in the execution of the framework. To the functions executed at the database nodes, the middleware passes the TabularData stub belonging to that database node so that the work is done on the proper dataset – however, the actual data will remain in that node. Data stored in non-TabularData format, such as NumPy arrays or PyTorch tensors, will be actually serialized and transferred across nodes. Indeed, users must be of cognizant of the format of the data that is being passed as parameters or return values for remote methods – lest they transfer large amounts of data.

5.1 Central Framework

The central framework is designed for ease of use when training models using iterative methods in the distributed AIDA system. The usage of it is simple: the user designs a model object, incorporating several necessary methods, and sends it to the middleware server, which will then automatically distribute the work defined by the user to the worker nodes where the data resides. In this abstraction, the user knows that the data and the learning is distributed. The user starts training with a DistTabularData stub, but is aware that each node will work on a data partition – the local TabularData object. However, the user is not concerned with the distribution process – the middleware takes care of that. The figure 5.1 illustrates the model calls and data flow for the central framework.

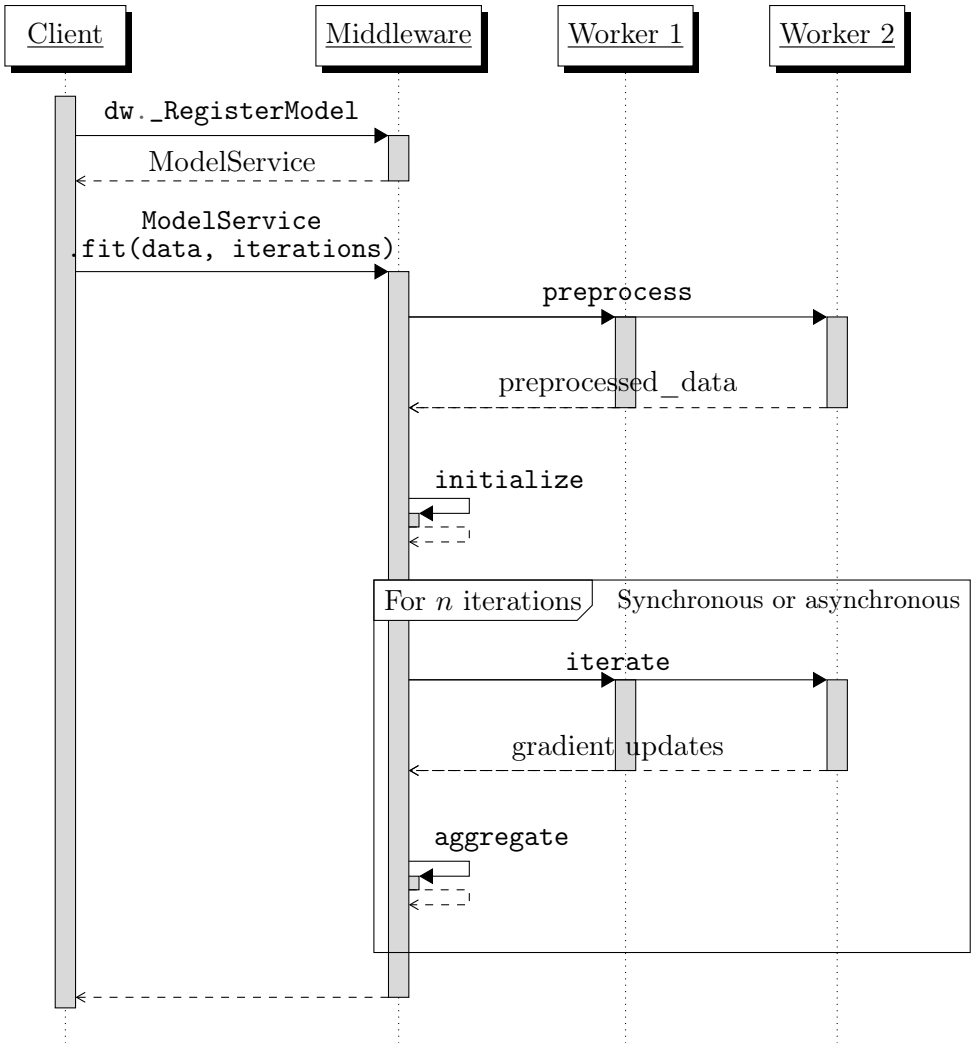

Figure 5.1: Sequence diagram for training on the central framework

This framework is optimized for distributed stochastic gradient descent. The methods designed by the user should follow the standard programming flow for gradient descent – the process starts off with data preprocessing, then model initialization, followed by iterations which calculate the loss and gradient updates, and an aggregation step which applies those gradients to the model. Each of these steps needs to be defined explicitly by the client – the middleware will automatically take care of the execution and distribution of the steps. The data preprocessing step is pushed to each of the database nodes by the middleware, where they will then be executed on their partition of the data. This could, for example, be the selection of rows in the database tables, linear algebraic operations or data transformations that need to be performed. Following that, the model initialization occurs at the middleware, which will keep track of the global model, perform updates to it, and send

it to the workers when necessary. The iterate method, defined by the client, should define the operations necessary for a single iteration. This method is performed by each of the workers for a client-defined number of iterations. Each iterate execution should calculate the loss on a minibatch of data and return the gradient update. The aggregation occurs next, on the middleware. It takes the result from the previous iteration and applies it to the central model. The execution of the iterations can be done synchronously or asynchronously. In the synchronous case, each `iterate` function is called by the middleware in a separate thread; the middleware waits until each `iterate` function has completed and returned before applying the updates received to the model by calling the `aggregate` method. In the asynchronous case, a separate thread is created for each worker which will independently alternate calling `work` in a worker and `aggregate` in the middleware until the number of iterations is satisfied.

Every call to the workers is initiated by the middleware, acting as a client to the database AIDA servers. The calls are performed using the `._X` operator provided by the database adapter connections, which are stored in the DistTabularData. Thus, the work is only performed on the nodes holding a partition of the training data. Each remote call is executed using RMI – the execution must complete and return before the next step can commence. Only the `preprocess` and `iterate` methods are executed at the worker nodes, whereas `aggregate`, the model update, occurs at the middleware.

The methods that need to be defined by the user are as follows:

- **preprocess(con, data):**

  Where `con` is the database connection, and `data` is the data in `TabularData` format. It should return the preprocessed data in another `TabularData` object. This function will be executed in each of the workers. The preprocessing step could vary for different models – many of them would not require a preprocessing step at all. Since the preprocessing step is only performed once at the beginning of training, it could also be used to set up variables that are only initialized once – such as the loss function, or a counter. In the case of linear regression, it could be used to add an additional bias term to the data.

- **initialize(self, data):**

  Where `data` is the distributed tabular data representation of the output of the preprocessing

step. This method will be executed on the middleware, and should be used to initialize the model weights. The weights should be initialized as a property of the model object, and will be passed to the workers in each iteration. Like the preprocess method, it is only executed once, but this time at the middleware. In addition to model initialization, it could also be used to initialize objects that are used to update the model during the aggregate steps, such as the PyTorch optimizer.

- **iterate(con, data, weights):**

  Where `con` is the database connection, `data` is the data in tabular data format and `weights` is the model weights. This function will be run on each of the worker nodes in each iteration, and will require the user to define the prediction, loss, and gradient calculation for a single iteration. It should return the gradient for a single iteration.

- **aggregate(self, results):**

  Where `results` is the output of the `iterate` function. If the model is being trained synchronously, it will be a list of the gradients sent from each worker node to the middleware, and if not, it will be a single result. This will be executed at the middleware.

- **weights:**

  While not a function, the `weights` property of the model should hold the model parameters, in whatever format the user sees fit. The object defined in `weights` will be held as the global model in the middleware. The data held by `weights` will be sent by the middleware to the worker nodes every iteration in place of the entire model, so as to limit the amount of data being sent.

After defining these methods in a user-created model object, the user would then register this object using the `_RegisterModel()` method of the middleware connection. It then returns a `ModelService` object, on which the user can call `fit(data,iterations,sync)` to define and start the training loop. The parameters of the `fit` function are fairly self-explanatory: `data` is the distributed tabular data representation of the data meant to be used in training, `iterations` is the number of iterations to be run, and `sync` is whether or not the iterations should be run synchronously.

Algorithm 4 Middleware algorithm run during `.fit` call on central framework

Input: x: Input data, M: user defined model, n: number of iterations, *sync*: synchronise flag

```
1  data ← {}
2  for connection in x.tabular_datas in parallel do
3      data[connection] ← connection._X(M.preprocess(x.tabular_datas[connection]))
4  preprocessed_data ← DistTabularData(data)
5  M.initialize(preprocessed_data)
6  if sync then
7      for n iterations do
8          for con in preprocessed_data.tabular_datas in parallel do
9              updates ← M.iterate(preprocessed_data.tabular_datas[con], M.weights)
10         M.aggregate(updates)
11 else
12     L ← lock
13     for con in preprocessed_data.tabular_datas in parallel do
14         for n iterations do
15             updates ← M.iterate(preprocessed_data.tabular_datas[con], M.weights)
16             L.lock()
17             M.aggregate(updates)
18             L.unlock()
```

The algorithm 4 outlines the steps taken by the middleware process when `.fit` is called on the `ModelService` object. All of the commands in red are methods or properties that the user must define as part of their model. The preprocess step is executed first in parallel, at lines 2-3. The result of the preprocess step is stored in a new DistTabularData object, and will be passed to the next step, initialize, which will initialize the `weights` of the model. This is so models, like linear regression, that depend on the shape of the data can be initialized properly. The program flow after the model initialization depends on whether or not the client wishes the training to be synchronous or not. If synchronous, the middleware executes the `iterate` method at each worker node, waits until they are all complete, then executes the `aggregate` method with the results at the middleware. This repeats for a user-defined number of iterations. If asynchronous, each worker node performs the `iterate` function, and the middleware immediately executes the `aggregate` function afterwards. The aggregation will occur behind a lock, so model updates are serial. Depending on whether the algorithm is performed synchronously or asynchronously, the `aggregate` function takes either a list of updates – in the case of synchronous execution – or a single update – in the case of asynchronous execution. It is up to the user to implement `aggregate` in such a manner that the updates are applied correctly.

```
1  class LinearRegressionModel:
2      def __init__(self):
3          self.weights = ...
4
5      @staticmethod
6      def preprocess(db, data):
7          ...
8          return (x_bias, y)
9
10     def initialize(self, data):
11         import numpy as np
12         x = data[0]
13         self.weights = ...
```

Listing 5.1: Linear regression preprocess and initialization using central framework.

Listing 5.1 presents a partial implementation of linear regression using the central framework. The full implementation can be found in the appendix at listing A.1. In this example, the algorithm and model are written using NumPy objects, though PyTorch can also be used. At the construction of the linear regression model at line 2, global constants such as the learning rate are defined. The `preprocess` method, as one that is executed on the workers, is wrapped in a `@staticmethod` decorator, and does not have access to any of the instance variables. Since it will be shipped to the workers using the `._X` functionality provided by AIDA, the first argument must be the database adapter. The `data` parameter is a TabularData object residing on the same node as where it is executed. Next, the `initialize` method is defined at line 10. This method will be executed at the middleware, and initializes the model in the `self.weights` property.

```
14     @staticmethod
15     def iterate(db, data, weights):
16         import numpy as np
17
18         batch = np.random.choice...
19         preds = ...
20         grad_desc_weights = ...
21         return grad_desc_weights
22
23     def aggregate(self, results):
24         self.weights = self.weights - ...
```

Listing 5.2: Linear regression iteration and aggregation using central framework

The model definition is continued in listing 5.2 with the `iterate` and `aggregate` methods. Line 18 retrieves the data from the TabularData objects by accessing the internal NumPy matrix representation, and retrieving a batch from it. Then, the predictions are made and the gradient update is calculated. The updates are returned to the middleware through the return statement. Next, the `aggregate` function defined at lines 23-24 show the application of the weights to the model.

```
25 from aida.aida import *
26 dw = AIDA.connect('middleware', 'database', 'username', 'password', 'lr')
27
28 model = LinearRegressionModel
29 data = dw.lr_data
30
31 service = dw._RegisterModel(model)
32
33 # Send start model training, specifying number of iterations
34 service.fit(data, 5000, sync=True)
```

Listing 5.3: Linear regression model training execution using central framework

Finally, the listing 5.3 shows how the linear regression model is executed on the client side. The client first connects to the AIDA middleware server on line 26. Line 31 registers the model at the middleware, returning the `ModelService` object. Finally, the model training is started by calling `.fit` on the `ModelService` object with the data to be used during the training, and the number of iterations to be performed. On calling the `.fit` method, it should be noted that the client only has the references to the AIDA server at the middleware, `ModelService` and the DistTabularData object representing the data to be trained on. These are all located at the middleware server. The client does not have access to the individual partitions or the workers – the distribution is handled by the middleware.

The main limitation with this framework is that it is limited to distributing iterative algorithms – without running several iterations, using this framework makes no sense. Additionally, since the model is passed to the workers as part of the call that starts the iteration on each worker, the worker cannot pre-load the data and identify which parts of the model are more useful: algorithms such as matrix factorization which could be optimized by passing only a small part of the model every iteration could suffer when implemented using the central framework.

5.2 Workflow framework

The workflow framework, inspired by map-reduce, is a more fine-grain framework that allows users more control over independent iterations. Unlike the central framework, which only supports iterative algorithms, the workflow framework supports a wider variety of iterative and non-iterative algorithms, at the cost of requiring more involvement on the part of the client to implement. Following the steps of map-reduce, the workflow framework is comprised of a series of user-defined steps. Each step consists of two phases: the work phase, which is run in parallel in each of the worker nodes, and the aggregate phase, which occurs in the middleware. In essence, it can be thought of as a MapReduce job where the map phase is the work phase and the reduce phase is the aggregate phase. While data transfer occurs between the phases, in the form of models or gradients being sent between the middleware and the worker nodes, the actual training data does not leave the worker nodes.

Each work phase is an RMI call by the middleware to each of the workers, executing the user defined `work` function. Each aggregate phase is executed on the middleware, using the middleware `dbAdapter` object as a context manager to store persistent variables between steps. The workflow framework works on a DistTabularData object – each `work` function executes on a node containing a TabularData object referenced by the DistTabularData object. The workflow model also passes a context object – in the form of a Python dictionary – to each execution of the `work` function. This context object can be modified at the start and at every `aggregate` execution at the middleware in order to pass additional variables in between the middleware and the workers at every setp.

The sequence diagram for the workflow framework is found in figure 5.2. Simply, each `work` step is executed at each worker in parallel, whereas the `aggregate` step is executed at the middleware using the results from the `work` step. In synchronous execution, the middleware will wait until each worker has completed the `work` step, before starting the `aggregate` step locally with the results from the workers. In the asynchronous version, a thread for each worker will be started at the middleware. Each thread will start the `work` method on one worker, wait until that method has returned, then perform the `aggregate` step at the middleware. The `aggregate` step is surrounded by a thread lock, so no two threads can perform that step at the same time. However, the `aggregate` step will not wait for any other worker to complete their `work` step before executing and moving on

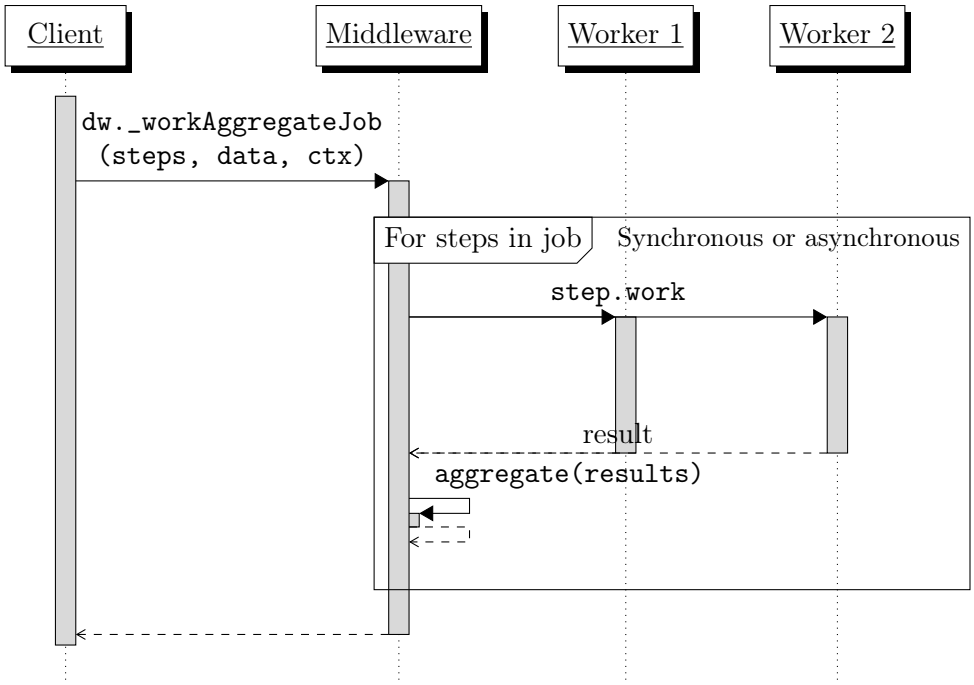

Figure 5.2: Sequence diagram for workflow framework

to the `work` function of the next step.

To start a workflow job, the user must first create a list of `Step` objects, which must have the following methods:

1. **work(con, data, context)**

   Where `con` is the database connection object, `data` is the data being worked on in `TabularData` format, and `context` is a dictionary consisting of whatever else the user may need. In steps following the first step, `context['previous']` consists of the output of the `aggregate` method of the previous step.

2. **aggregate(con, results, context)**

   Where `con` is the database connection, and `results` is a list of what is returned from the worker nodes in the previous `phase` in an arbitrary order. The `context` object is the same object that is given to the `work` function – if the user wishes to change it for the next step, it can be changed here.

The user can then submit the list of steps using the `_workAggregateJob(list, data, context, sync)` method, where `data` is the `DistTabularData` representation of the data being used in the job,

`context` is the context dictionary to be passed to the `work` phases and `sync` the parameter to be set if the steps should run synchronously or not (defaults to `True`). The framework will then alternate between executing the `work` and `aggregate` phases of each step synchronously or asynchronously until completion. In addition, the list can contain a tuple `(step, iterations)` if a step is meant to be run multiple times in sequence during the training.

Algorithm 5 Middleware algorithm run during `_workAggregateJob` call

Input: *li* list of steps, *x* data, *sync* synchronize flag, *ctx* context dictionary

```
1  if sync then
2      for step in li do
3          for con in x.tabular_datas in parallel do
4              result ← step.work(x.tabular_datas[con], ctx)
5          agg_result ← step.agg(result, ctx)
6          ctx["previous"] ← agg_result
7  else
8      L ← lock
9      for con in x.tabular_datas in parallel do
10         ct ← copy of ctx
11         for step in li do
12             result ← step.work(x.tabular_datas[con], ct)
13             L.lock()
14             agg_result ← step.agg(result, ct)
15             L.unlock()
16             ct["previous"] ← agg_result
```

The general program flow the workflow framework executes is presented in algorithm 5. Like before, the red methods are user-defined methods. While not presented in the above algorithm, if the user wishes to have one step repeat several times in a row, they merely need to encase it in a tuple.

As an example of the workflow framework, we present linear regression, this time written in PyTorch. The model and first step of the job is presented in Listing 5.4. The full code for this example can be found in the appendix at Listing A.2. The linear regression is written as normal in PyTorch. The `FirstStep` class contains both the `work` and `aggregate` functions for the first step of the program. Unlike in the central framework, both of these functions are wrapped in the `@staticmethod` decorator, even though `aggregate` is run on the middleware. This is because the workflow framework does not create any object instances in the middleware, so any objects that need to be maintained through iterations, such as the model, must be stored in the AIDA database

```
1  import torch
2  from aida.aida import *
3
4  class LinearRegression(torch.nn.Module):
5      def __init__(self, input_size, output_size): ...
6
7      def forward(self, input):
8          return self.linear(input)
9
10 dw = AIDA.connect('middleware', 'database', 'username','password', 'lr')
11 dw.lr_model = LinearRegression(5, 1)
12
13 class FirstStep():
14     @staticmethod
15     def work(dw, data, context=None):
16         data.makeLoader([('x1', 'x2', 'x3', 'x4', 'x5'), 'y'], 1000)
17         ...
18         return
19
20     @staticmethod
21     def aggregate(dw, results, context):
22         dw.optimizer = ...
23         ...
24         return dw.lr_model
```

Listing 5.4: First step for a linear regression model implemented in the workflow framework

adapter workspace. This holds true for the workers and the middleware. Thus, the model is initialized and stored at the middleware beforehand in line 11. Additionally, the `FirstStep.work` method, implemented from lines 14-18, initializes objects that should be persistent. At line 16, this function makes use of a new in-built method for transforming TabularData data into a PyTorch tensor format – the `makeLoader` function. This creates a loader that will be stored in the TabularData. The first argument is a tuple x, y, where x is a list of columns forming the input variables, while y is the label. The second parameter is the size of the batch retrieved each iteration from the loader. The client may choose to load data in a different format – in that case, they will have to implement it themselves. In this example, the `work` method for the `FirstStep` class is equivalent to the `preprocess` method in the previous framework. Next, the `aggregate` method simply initializes the PyTorch optimizer with the parameters from the model stored at the middleware. This function returns the `dw.lr_model` model object because objects that the `aggregate` function returns will be

passed on to the `work` functions executed on the next step.

```
25 class Iterate():
26     @staticmethod
27     def work(dw, data, context):
28         import torch
29 
30         model = context['previous']
31         batch, target = ...
32 
33         preds = ...
34         loss = ...
35         loss.backward()
36         grads = [param.grad for param in model.parameters()]
37         return grads
38 
39     @staticmethod
40     def aggregate(dw, results, context):
41         dw.optimizer.zero_grad()
42         ...
43         dw.optimizer.step()
44         return dw.lr_model
45 
46 job = [FirstStep(), (Iterate(), 50000)]
47 
48 dw._workAggregateJob(job, dw.lr_data)
```

Listing 5.5: Iterate step for a linear regression model implemented in the workflow framework

The next step is written in the listing 5.5. The `Iterate.work` function executes on the worker nodes and is equivalent to the `iterate` function in the previous central framework. At line 30, the return value from the previous `aggregate` function is found in the `'previous'` key of the `context` dictionary that will be passed from workers to middleware and back again at every step. Line 31 retrieves data from the dataloader one iteration at a time, then the predictions, loss and gradients are calculated from lines 33-35. The gradients are then extracted from the model at line 36, and returned from the `work` function to be shipped back to the middleware. The next `aggregate` function will update the model with the gradients by applying the gradients to the model at the middleware and running the optimizer initialized in the `FirstStep`. Like before, the model will be returned so as to be shipped for the next iteration.

The workflow job is created at line 46. Since this is an iterative algorithm, this example program

puts the `Iterate` class in a tuple, with the second element indicating how many times to run the step. Then, the job is submitted along with the data being worked on at line 48. This program will run synchronously by default, but if the client wishes to change that, they may pass `sync=False` as the last argument to `_workAggregateJob`.

This framework has a similar shortcoming as the central framework, in that execution on the workers must be started by the middleware, and further communication between the workers and the middleware is blocked until the iteration finishes. Thus, like the central framework, the workflow framework would not be able to send only subsets of the model according the minibatch loaded each iteration by the workers. However, due to its more flexible structure, the workflow framework allows for non-iterative distributed algorithms, such as clustering.

5.3 Parameter Server framework

The parameter server framework is quite different from the previous two frameworks implemented for D-AIDA. Unlike either of the previous frameworks, the parameter server framework does not support synchronous parallelization. Additionally, the previous two frameworks have the middleware drive the work on each of the worker nodes: it indicates when they should start an iteration, and pushes the necessary data to each worker. In the parameter server framework, each worker instead chooses when to start an iteration by themselves, and pulls and pushes model related data as necessary from the middleware.

In this framework, the user must define the model they wish to train and some additional functions that will allow the workers to interact with the model. The model itself will be an object residing in the middleware AIDA server. The model object will be wrapped in a server object, also residing at the middleware, which implements the `pull` and `push` functions required by the workers to interact with the model. The middleware will create and pass the stubs for this server object to the worker nodes when the workers start training. Then, to pull and push parameters and parameter updates to and from the model at the middleware, the workers need only interact with the server object stub they hold. The server object runs a thread for each of the workers, using an RMI method call to initialize the training at each worker, as well as a separate thread in the middleware to continuously update the model when new model updates come in as workers use

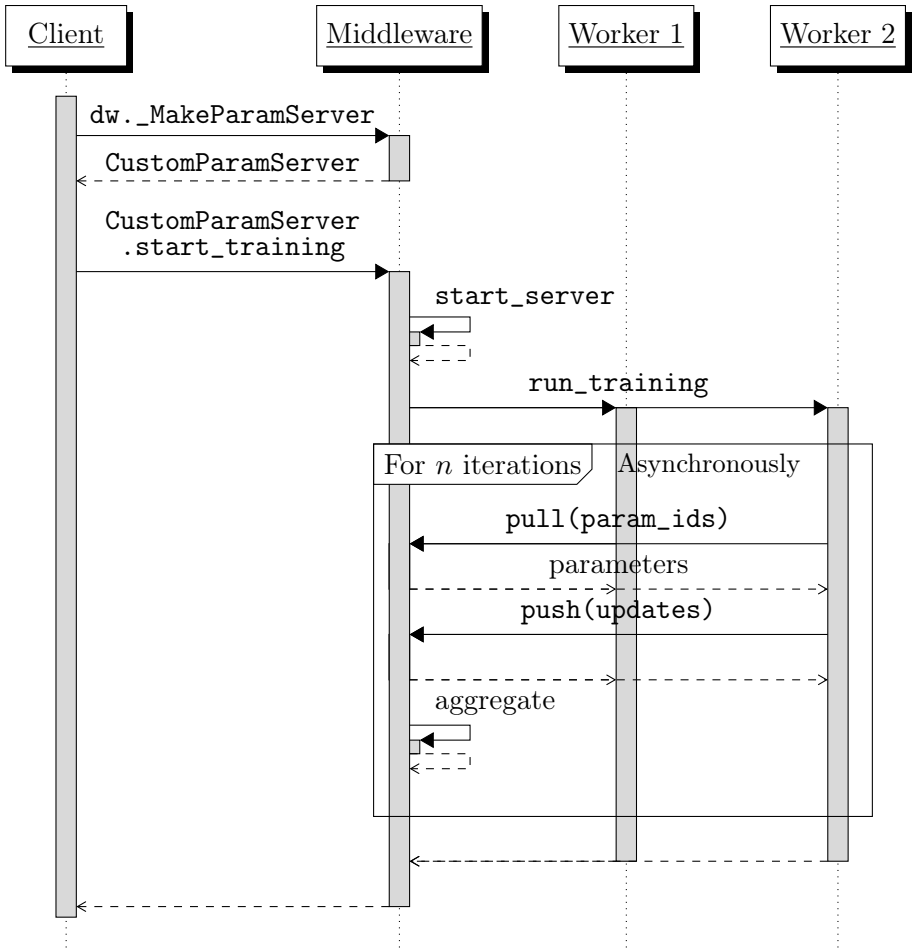

Figure 5.3: Sequence diagram for parameter server framework

the `push` function. Because of this implementation, the parameter server framework only supports asynchronous training. The server object stub is also passed to the client side, so the client can indicate when to start the training.

Thus, each `pull` and `push` function is an RMI method belonging to the stub of the parameter server residing in the middleware. As such, each method is a call to the middleware, an execution by the middleware, and a return from the middleware. The workers block until the call is completed – it is therefore suboptimal for the execution by the user-defined `pull` function to be long. It can, however, be used to grant a primitive form of consistency – each `push` function transfers the model update to a queue in the middleware. The length of the queue is user-defined – if a worker tries to push into a queue that's full, it will block until the queue has room – i.e. when an update has been used by the thread performing the model updates.

The figure 5.3 demonstrates the flow of a process using the parameter server framework. As the diagram demonstrates, this framework, unlike the previous two, has the workers initiate method calls to the middleware instead of only the other way around. Any parameter pulls and update pushes are done asynchronously, and the `aggregate` function is called arbitrarily in the middleware when there are updates available to be applied. An implementation of matrix factorization through the parameter server framework can be found in the following listings.

The parameter server framework allows the user to create the model, then register it as a parameter server in the middleware to allow the processes being run on the data nodes to push and pull parameters during training. In the middleware, the parameter server exists as a key-value storage of parameters from which the workers can request specific parameters through their keys, and update those same parameters by sending the updates. The updates are stored in a queue. A separate thread in the server will update the parameters and empty the queue periodically. This type of framework is useful for when algorithms do not need to access the entire model during the training phase. The user should define the parameters in the server, and the parameter server itself exposes the following functions to the worker nodes:

1. **pull(self, param_ids)**

   This function pulls the parameters indicated by `param_ids` to the worker nodes. The model can be accessed via `self.model`, and the parameter ids as well as how to access parts of the model using them must be defined by the user.

2. **push(self, update)**

   This function pushes parameter updates to the parameter server. Often, this would be gradient updates, along with the `param_ids` of the parameters they're updating.

For this framework, the user must define their model, along with the following functions:

1. **run_training(con, ps, data)**

   Where `con` is the database connection and `ps` is a reference to the parameter server. The user should write this function as if all the training for a single data node is contained within it. Naturally, `data` is the data the training is being run on.

2. **pull(self, param_ids)**

Where `param_ids` is a list of parameter ids to be pulled from the parameter server `ps` by the `run_training` method. This overrides the `pull` method of the parameter server.

3. **update(self, update)**

   Where `update` is a single update sent by the `push` method of the parameter server. This is run in the parameter server intermittently.

The user then initializes the parameter server using the function `_MakeParamServer(model, server, max_queue_length)`. The `model` object can have any form the client wishes – be a NumPy array or PyTorch model. The `server` object should implement the user-defined functions described above. In its constructor, it should take the `model` object featured as the first parameter. The `max_queue_length` parameter is an integer limiting the length the update queue can grow to. If a worker tries to push an update while the queue has hit its max length, the `push` function will block until the queue has been emptied. This parameter can be used to ensure the model is frequently updated, and as long as workers are pushing updates before pulling the updated model, no worker will be working on models that are overly stale. This will return a `CustomParamServer` object, on which the user can call `start_training(data)` to distribute and run the user-defined `run_training` function on the workers.

Algorithm 6 describes the execution of the user-defined functions in the middleware. The `pull` and `push` functions, written in blue, are the functions available to the database nodes as they run the training. The function `start_training`, defined at lines 22-25, merely starts the server update thread, then starts the training on each worker. To ensure workers only pull models that have completed an entire update step, both the `update` function and the `pull` function are behind locks.

Listing 5.6, along with the next two listings, demonstrates an example of matrix factorization, written using PyTorch and the parameter server framework. The full program can be found in Listing A.3 in the Appendix. First, the machine learning model is defined normally as a PyTorch model. In this case, the model holds two embeddings – embeddings for the users and embeddings for the items – which represent the two matrices to be learned. Naturally, this will vary based on the data.

Listing 5.7 demonstrates some of the functions implemented by the user for matrix factorization using a parameter server. The constructor takes the model, and defines an optimizer on it using

Algorithm 6 Process flow for parameter server framework

Input: *model*: user-defined model, *server*: user-defined functions, *schedule*: integer

```
sever ← server(model)
updates_queue ← Queue(max=schedule)
L ← lock

function PULL(param_ids)
    L.lock()
    server.pull(param_ids)
    L.unlock()

function PUSH(update)
    updates_queue.push(update)

function RUN_SERVER(updates_queue)
    while is_running do
        update ← updates_queue.pop()
        if update == "finish" then
            break
        L.lock()
        server.update(update)
        L.unlock()

function START_TRAINING(data)
    Start run_server in another thread
    for con in data.tabular_datas in parallel do
        con._X(server.run_training, data._tabular_datas[con])
```

```python
import torch

class MatrixFactorization(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.user_factors = torch.nn.Embedding(...)
        self.item_factors = torch.nn.Embedding(...)

    def forward(self, data):
        return (self.user_factors(data[0]) * self.item_factors(data[1])).sum(1)
```

Listing 5.6: Matrix Factorization model definition in PyTorch

the PyTorch library. The `pull` function is defined to return the embeddings requested through the `param_ids` parameter. The `update` function takes a list of two gradients – the gradient update for the users, and the gradient update for the items – records them in the `.grad` property of the model

```
11 class CustomMF:
12     def __init__(self, model):
13         import torch
14         self.model = model
15         self.optimizer = ...
16 
17     def pull(self, param_ids):
18         return (self.model.user_factors(param_ids[0]),
           ↪  self.model.item_factors(param_ids[1]))
19 
20     def update(self, update):
21         self.model.grad = update
22         self.optimizer.step()
23         self.optimizer.zero_grad()
```

Listing 5.7: Matrix Factorization server definition in PyTorch

parameters and lets the optimizer perform the model update.

```
24     @staticmethod
25     def run_training(con, ps, data):
26         import torch
27 
28         data.makeLoader(...)
29 
30         loss_fun = torch.nn.MSELoss()
31         for i in range(iterations):
32             users, items = ...
33             factors = ps.pull((users, items))
34             preds = ...
35             loss = ...
36             loss.backward()
37 
38             grads = [f.grad for f in factors]
39 
40             ps.push(grads)
```

Listing 5.8: Matrix Factorization `run_training` user-method

Listing 5.8 shows the training phase of the algorithm. Like in the central framework, the `run_training` is wrapped by a `@staticmethod` decorator since it is deployed at the workers. The dataloader and the loss function, using one of PyTorch's in-built methods, are initialized here. Every iteration starts with retrieving the data from the loader, expressed at line 32. Then, the

parameter ids are retrieved from the data – in this case, the user ids from the first column and the item ids from the second column of the data tensor. Line 33 demonstrates pulling of parameters from the parameter server. Lines 34-36 is the standard method of gradient descent using PyTorch, as demonstrated back in section 2.4.5. Afterwards, the gradients are retrieved from the factors. As the final step of the iteration, the gradients are then pushed back to the parameter server at line 40.

```
41 from aida.aida import *
42
43 dw = AIDA.connect('middleware', 'database', 'username', 'password', 'mf')
44 server = dw._MakeParamServer(MatrixFactorization, CustomMF)
45 data = dw.mf_data
46 server.start_training(data)
```

Listing 5.9: Execution of training using parameter server model.

Finally, listing 5.9 demonstrates how to start and execute the training. The `_MakeParamServer` method takes the class of the model and the user-defined server methods, and returns a `CustomParameterServer` object. Then, `start_training` is called on the object along with the DistTabularData object to be used in the training, which will start the server update thread at the middleware, and execute the previously defined `run_training` method at the worker sites.

Algorithms built using the parameter server framework will always be run asynchronously. The iterations are not driven by the middleware; rather, each worker executes their iterations in a loop, using only the `push` and `pull` functions provided by the parameter server. There is no guaranteed ordering of any updates. Additionally, even the aggregations are performed in a separate thread. In the previous two frameworks, even when executing asynchronously, each worker will wait to start the next iteration until the middleware has applied the previous iteration's updates to the global model. This is not the case in the parameter server framework – the updates are pushed to a queue, and the worker can continue its next iteration without waiting for aggregation to occur at the middleware.

This framework is the only framework in which the workers drive the execution instead of the middleware – as such, they can complete a partial iteration by loading the minibatch of data, then demand the model from the middleware by acting as a client to the middleware AIDA server instead

of the other way around. Thus, unlike the previous two frameworks, the parameter server framework can request for a subset of the model, making it ideal for algorithms like matrix factorization. However, this is not without its downsides. While an iteration in the previous two frameworks would consist of two communication steps – the middleware starting the iteration and the worker returning a value from the iteration – an iteration using the parameter server framework would instead have four communication steps – the worker would make a request to the middleware for specific parameters, the middleware would return those parameters, the worker would push updates to the middleware, and the middleware would acknowledge the push. This can incur a larger communication overhead in instances where optimizations provided by the parameter server are not applicable, such as in linear regression, which requires the entire model for every iteration.

5.4 Discussion

As a proof of concept, the frameworks implemented by D-AIDA are malleable, but could still be improved upon. For now, the central and workflow frameworks support both synchronous and asynchronous execution, and the parameter server framework asynchronous execution. To further expand these frameworks, a tracker in the middleware could be implemented to keep track of iterations in each worker – with this, stale synchronous parallelism (SSP) could be supported. While the parameter server framework can support a weak form of consistency by ensuring that the number of updates in the queue does not exceed a certain number, the queue is global, and could still allow one worker to perform iterations much faster and finish earlier than another. SSP would ensure that no worker proceeds when they are a user-defined number of iterations ahead of the slowest worker.

Currently, all these frameworks support running each worker for a fixed number of iterations. While this is useful, non-distributed iterative machine learning methods often employ other criteria for stopping – such as after the loss decreases past a certain threshold, or when the magnitude of the gradient nears zero. Some of these could potentially be employed in the synchronous frameworks provided by D-AIDA. For instance, a user could choose to throw an exception when the `aggregate` function at the middleware encounters a small gradient – this would prevent the middleware in either the central framework or the workflow-based framework from starting the next iterations. This

could be expanded upon, by introducing other methods the user could define in order to check for certain stopping conditions. However, from review of the literature surrounding distributed machine learning, there does not appear to be a popularly employed condition for stopping asynchronous models aside from stopping after a certain number of iterations. Loss or gradient calculations could differ depending on worker – stopping a worker early because its stopping conditions are met may result in another worker continuing and moving the model parameters towards an undesirable position.

6

Experiments

For experimental results, we focus mainly on testing the machine learning frameworks provided by D-AIDA. As our current implementation of the distributed relational operators serve merely as a proof of concept – in that classical distributed processing can be implemented in D-AIDA, and it does not feature a full-fledged implementation, we focus more on the capability of D-AIDA to support machine learning algorithms and its ability to scale. We compare the different frameworks we have developed against the well-known distributed machine learning libraries provided by PyTorch. For data, we mainly use generated data for simplicity. Our analysis focuses on performance in terms of runtime of learning and how easily the algorithms can be implemented, and not so much on the performance of the model in terms of accuracy, precision, or F-score. Thus, we believe generated data is sufficient for our purposes.

These experiments are performed on multiple machines, each with 32 GiB system memory, with an Intel (R) Xeon(R) CPU E3-1220 v5 @ 3.00GHz. They are run in a non-distributed fashion, in

a system with two worker nodes, and in a system with four worker nodes. In the D-AIDA tests, we assume the data to be pre-distributed and residing in the database at the worker nodes in the form of tables residing in the database. The program is written at the client side, which calls the middleware AIDA server to submit and execute the code. The PyTorch tests access the same data at each worker by reading from a csv file. The code for the PyTorch tests are present at each site, and executed locally everywhere. In our runtime measurements, we do not include the time needed to read the data from the database into TabularData objects, nor the time from csv file to PyTorch tensors.

6.1 Iterative Algorithms

To test these machine learning frameworks, we first ran experiments using linear regression and matrix factorization using all three frameworks. We compared the performance of the D-AIDA system against distributed PyTorch setups using parameters servers made using PyTorch RPC and the PyTorch DistDataParallel modules. We also compared the performance against a central PyTorch model with no distribution – in this central solution, all the data is local to this central node, i.e. no data is transferred. Were we to assume distributed data, a centralized solution would first need to transfer all data to the central node – an additional overhead. For comparison, we run each framework with a set number of gradient descent iterations and time how long it takes for each to finish. For the PyTorch RPC implementation, we follow the tutorial in PyTorch's documentation [43]. However, we find that that implementation is a non-standard parameter server implementation – instead of parameters being moved to the data sites, the data is instead moved to the model sites. As such, this has a negative impact on the performance of the PyTorch RPC parameter server. For DistDataParallel, recall that the worker nodes each run an iteration, then broadcast their gradients and update the local model copy in an AllReduce fashion. The next iteration is started afterwards in a synchronous manner.

Each iterative algorithm is given a set number of iterations to complete – in single, non-distributed systems, all iterations will be completed on one node, whereas in multiple node systems, the iterations will be split equally between every node. While we are not focused too much on the performance of these models in terms of achieved loss after completion, it must be noted

that DistDataParallel automatically scales the gradient step with the number of workers present in the system – for it to achieve a similar loss as the implementations provided by PyTorch RPC or D-AIDA, it must run the full set of iterations on every worker. However, for the purposes of these experiments, we ignore this, and scale the number of iterations run per worker the same for each framework.

In D-AIDA experiments with two workers, three nodes are used: two for each worker node, and one for the middleware and client. The PyTorch RPC setup uses the same configuration, with two nodes acting as workers and one acting as the parameter server, but the PyTorch DistDataParallel approach uses only two workers, as there is no coordinator process. Likewise, setups with four worker nodes are tested on 5 nodes for the D-AIDA frameworks and PyTorch RPC, and 4 nodes for PyTorch DistTabularData.

6.1.1 Linear Regression

Linear regression, described in Section 2.4.2, is a simple gradient descent algorithm. Compared to deep learning algorithms, its model size is quite small, and its computations simple. It is common for use on relational data to detect trends, or to predict an unknown attribute given the others.

6.1.1.1 Dataset

For linear regression, we generate 5 million rows of data following the equation:

$$f(x) = 3x_1 + 5.1x_2 - 6x_3 - 1.5x_4 + 0.2x_5 + 2.33 + N(0, 1)$$

where x_i are the features of the data, and $N(0, 1)$ is the normal distribution with mean of 0 and variance of 1. Each x_i is a random integer between the values of -15 and 15. The coefficients are chosen arbitrarily, and the equation written for ease of use in model implementation. The generated data is stored in a database table with 6 attributes, one for each of the data features and one for the label. In the distributed case, the tuples are randomly and equally divided amongst each partition.

When loading data from the TabularData object to the PyTorch DataLoader, the data is first materialized as the columnar dictionary described in Section 2.6.2. The values of this dictionary, being NumPy arrays, are then transformed into PyTorch tensors. Since they use the same C

language data structure, no copying of the data is necessary. In this case, each of the features, x_1, x_2, x_3, x_4, x_5 are loaded as integers, while the labels, y, are loaded as doubles.

6.1.1.2 Implementation

As we are comparing with PyTorch implementations, we use PyTorch to write algorithms we are running using the frameworks D-AIDA provides.

```
1  import torch
2
3  class LinearRegression(torch.nn.Module):
4      def __init__(self):
5          import torch
6          super().__init__()
7          self.linear = torch.nn.Linear(5, 1)
8
9      def forward(self, data):
10         return self.linear(data)
```

Listing 6.1: Model used in experiments for linear regression

Listing 6.1 shows the model used for all of the implementations. It is a simple model using a single linear layer to represent the linear regression function learned. The linear layer has five parameters, one bias term, and outputs a one-dimensional number.

The loss function used is MSE, provided by the `torch.nn.MSELoss()` function. The model is updated using standard SGD with a learning rate of 0.0003, also provided by the PyTorch optimizer `torch.optim.SGD(model.parameters(), lr=0.0003)`. In each of the D-AIDA setups, the loss function is initialized at each worker, and the optimizer initialized at the middleware, where the model resides. The linear regression algorithm is run with a minibatch of 1 000 records, with a total of 10 000 iterations. On the central node, all 10 000 iterations are run for comparison. On the setup with two worker nodes, each worker node runs 5 000 iterations each on their local dataset. With four worker nodes, each worker runs 2 500 iterations.

The D-AIDA central framework follows the same process as described in Section 5.1, using PyTorch instead. The `preprocess` step on each worker first loads the data in a PyTorch DataLoader and initializes the PyTorch loss function and stores it in the local workspace. The `initialize` function in the middleware initializes the PyTorch model as well as the PyTorch optimizer taking

the model parameters. The `iterate` function loads the next minibatch of data from the dataloader, calculates the gradients from the copy of the model it holds. It returns the gradients to the middleware. The gradients are a list of two PyTorch tensors, representing the gradients for the weights of the single linear layer, and the gradient for the bias of that same layer. The `aggregate` function at the middleware then applies the gradients to the model using the optimizer initialized during the `initialize` function.

The workflow framework creates two steps: the `FirstStep` step and the `Iterate` step. The `work` phase of `FirstStep` does the same operations as the `preprocess` call in the central framework, and the `aggregate` phase does the same operations as the `initialize` call. The next step, `Iterate`, is run multiple times in a row on each worker – 5000 times on each worker if there were two workers. The `work` phase performs the forward and backward pass, retrieving and returning the gradients, and the `aggregate` phase in the middleware applies those gradients.

Since each data record needs the entire model to make a prediction, the parameter server setup lets each worker pull the entire model every time they call `pull` on the parameter server object residing in the middleware. The `run_training` method, implemented by the client and executed on the workers, executes all iterations in a loop. The `run_training` method on each worker first loads in the data in a dataloader and initializes the loss, before proceeding onto the iterations. Each iteration starts with data being retrieved from the dataloader. Then, a `pull` is performed on the parameter server, retrieving the latest model from the middleware server. The loss and gradient updates are calculated, and the gradients are pushed to the middleware as a list of two PyTorch tensors: the gradient for the weights, and the gradient for the bias.

6.1.1.3 Results

To begin, we present the comparisons between linear regression run on several frameworks using two worker nodes. In this instance, each worker node runs 5 000 iterations.

Figure 6.1 presents the difference in execution time between a non-distributed Pytorch (labelled 'Central Pytorch'), PyTorch RPC, PyTorch DistDataParallel, and all D-AIDA frameworks. The execution time takes only the time it takes to execute the iterations – the preprocessing step and data loading step are not included. As these times may differ depending on the way the data is stored and we wanted to focus on the execution time for the machine learning, those steps are not

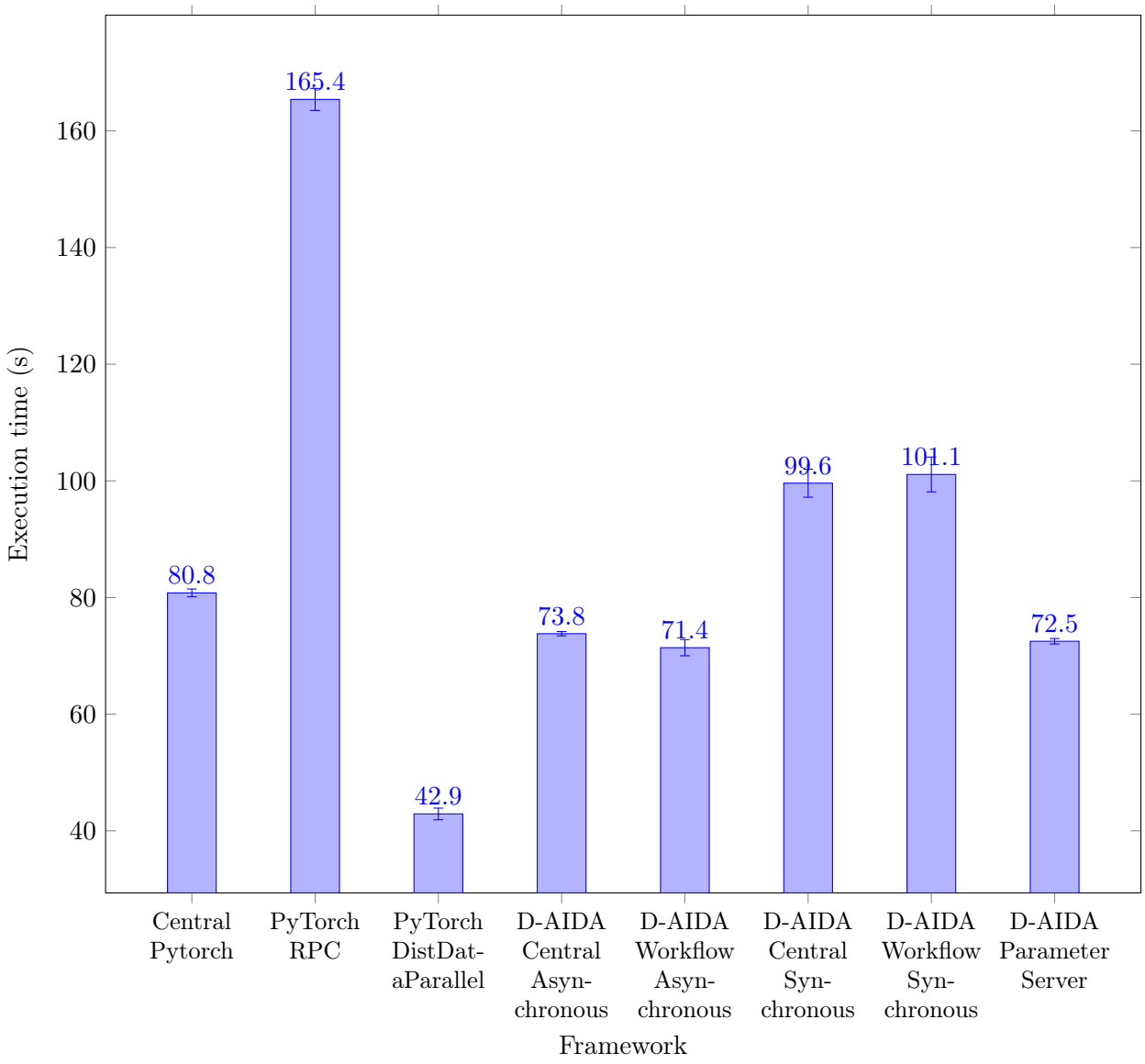

Figure 6.1: Comparison of execution times for linear regression between all frameworks with two worker nodes.

included in the times we measured. For each implementation, the algorithm is run five times, and the average of execution time of all 5 runs is presented in the figure.The error bars represent the standard deviation across all trials for each framework.

For linear regression, the PyTorch RPC framework performs the worse, at nearly twice the amount of time it takes for a non-distributed implementation. The PyTorch DistDataParallel performs the best, reducing the execution time by nearly half compared to the central version. This is natural: this linear regression model is very small, offering only 6 parameters, and thus moving

the data to the model site, like the PyTorch RPC implementation, incurs large amounts of overhead. Otherwise, broadcasting only the model gradients to a small group of two worker nodes, like the PyTorch DistDataParallel implementation, saves in communication costs. Calculation of the new parameter values, which must be executed at all nodes, is not as expensive, comparably. Indeed, for linear regression, the performance improvement is nearly linear over the non-distributed version. This shows that the AllReduce library used by PyTorch offers efficient communication of the gradients. The performance of the D-AIDA frameworks vary: the asynchronous version of the central framework, the asynchronous version of the workflow framework, and the parameter server framework, which is by default asynchronous, perform slightly better than a non-distributed implementation, whereas synchronous versions of both frameworks that support it perform worse. Compared to the magnitude of the mean execution time, the standard deviation of the execution times is relatively small. Of the frameworks that are faster than the non-distributed version, DistDataParallel exhibits a speedup of around 1.88 times, much higher than the speedups from the D-AIDA frameworks, showing a speedup of 1.10 times for the asynchronous central framework, 1.13 times for the asynchronous workflow framework, and 1.11 times for the parameter server framework.

Figure 6.2 presents a more in-depth comparison between the D-AIDA frameworks. We note that we cannot take these microbenchmarks for the distributed PyTorch frameworks, as communication and calculation is linked internally in the library. In this diagram, the calculation time refers to the time it takes to perform the forward pass and gradient calculations on each worker node. The aggregation time refers to the time it takes to aggregate the gradient results and update the model on the middleware node. The batch time refers to the amount of time required to retrieve each minibatch of data from the PyTorch DataLoader at each worker node, and the remaining time refers to the difference between the total execution time and the times previously calculated – any time spent for communication or blocking is recorded here. For times that are individually calculated at each worker node – such as the calculation time or batch time, the average between the two worker nodes is taken.

In Figure 6.2, it can be seen that the calculation time, aggregation time and batch time remains similar across the different frameworks. As batch time is the largest component, this time being roughly half the total execution time of the non-distributed version, it is well distributed across the two workers. Comparably, the communication cost is much larger for the synchronous frameworks.

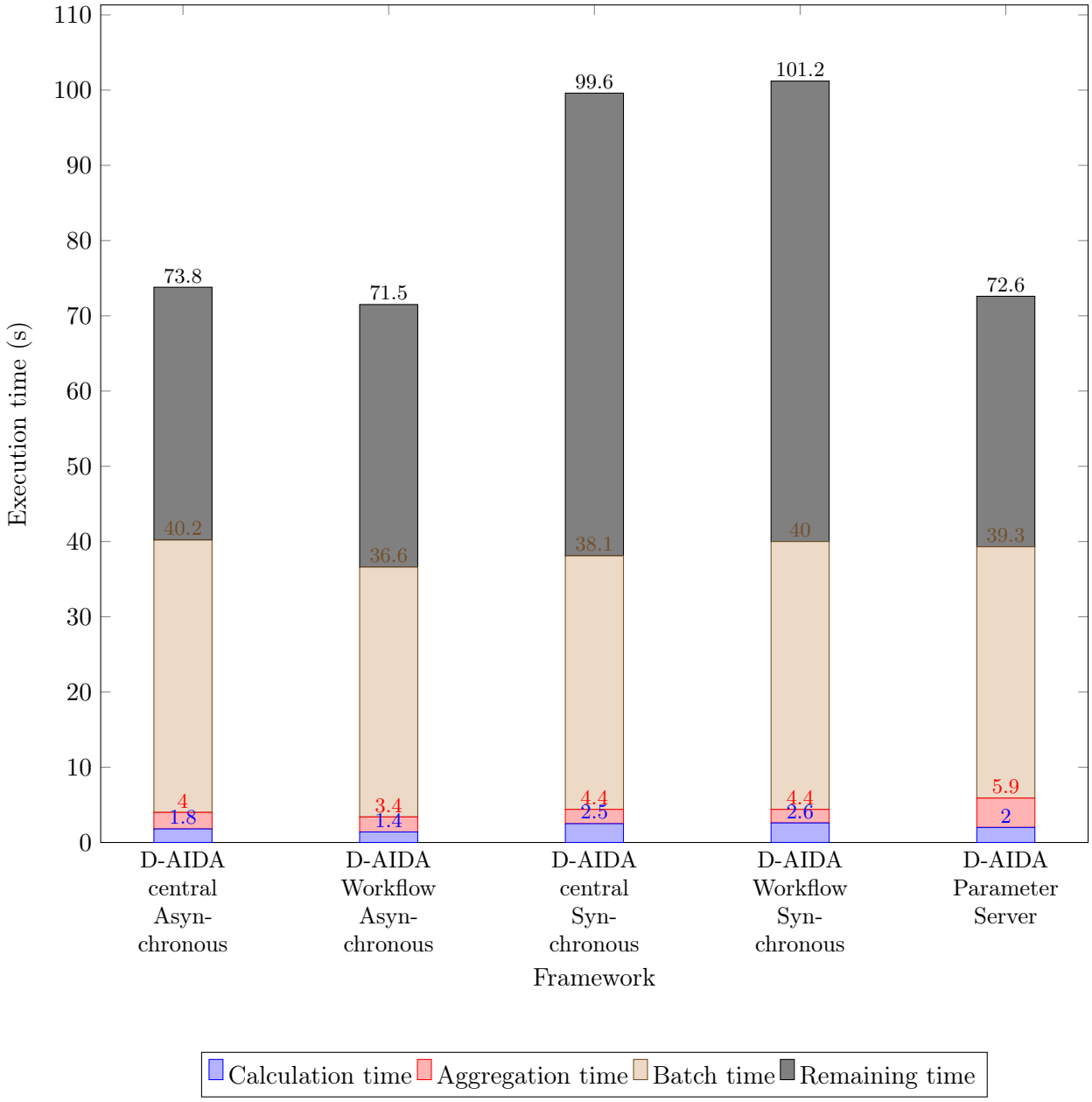

Figure 6.2: Linear regression microbenchmarks for D-AIDA with two worker nodes.

This is to be expected – as synchronous frameworks need to wait for the slowest worker each iteration, the amount of time spent blocked by each worker is much larger than in asynchronous versions. Despite the fact that the parameter server framework takes two more communication steps to pull the updated model and push model updates, its performance remains roughly equivalent to that of the asynchronous central and workflow frameworks – this is likely due to the asynchronous aggregation execution at the middleware.

Compared to DistDataParallel, the time spent in calculation, aggregation, and batching data is similar to the total execution time DistDataParallel exhibits. Indeed, most of the time lost by D-AIDA compared to the DistDataParallel execution time is in the remaining time – the time spent in communication and blocking. This is to be expected – all distributed versions must execute the same calculations, however, PyTorch uses a much faster communication library than AIDA RMI, and likely has more efficient and specialized serialization techniques.

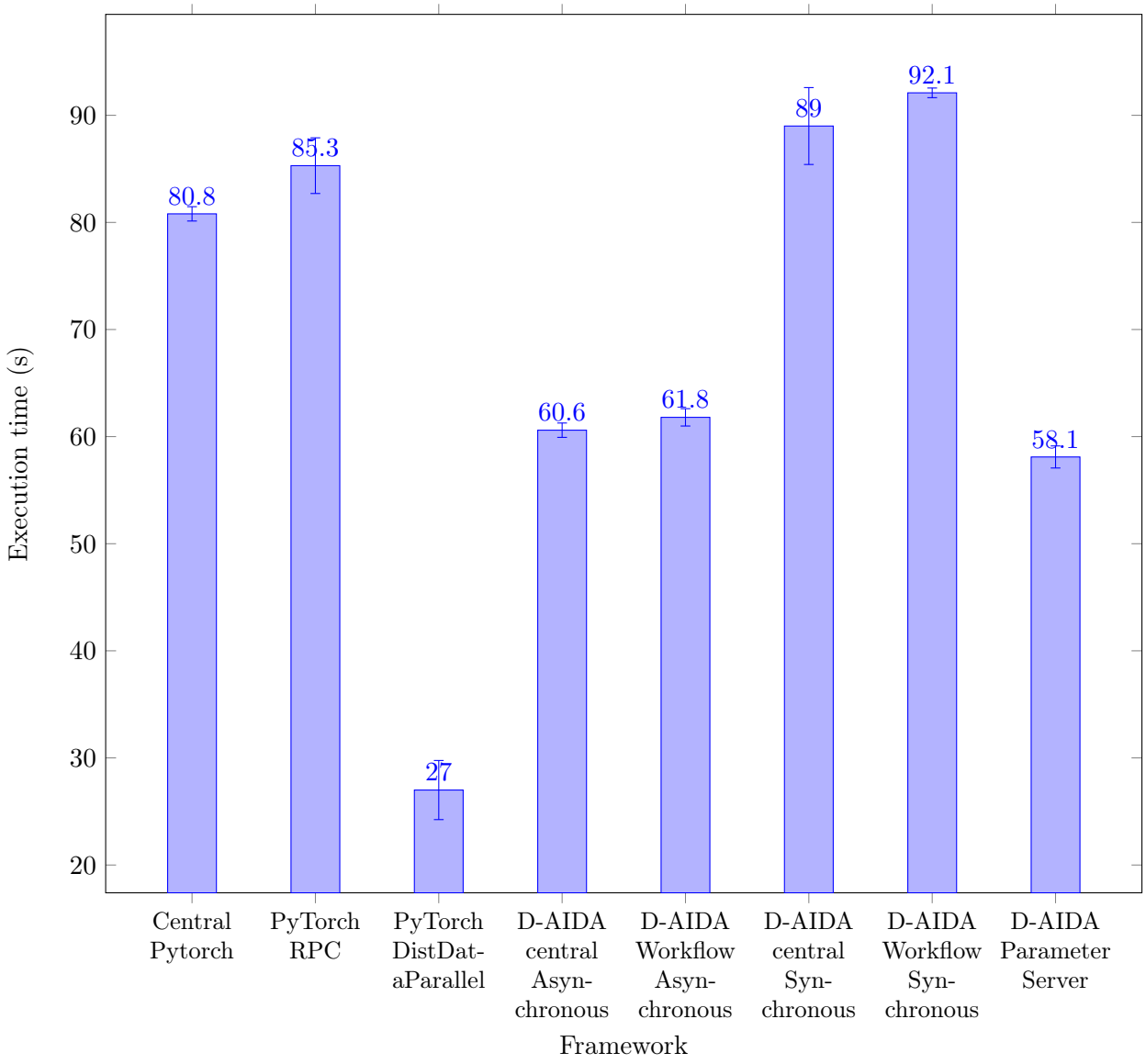

Figure 6.3: Comparison of execution times for linear regression between all frameworks with four worker nodes.

Figure 6.3 presents the same frameworks, running on 4 workers nodes instead. We see large

improvements for the PyTorch RPC framework, cutting its execution time nearly in half – representing a nearly linear improvement. PyTorch DistDataParallel goes from being nearly 2 times faster than non-distributed PyTorch in the two worker environment to being nearly 3 times faster in the four worker environment. D-AIDA also see its execution times decrease across all offered frameworks by around 10 seconds – not nearly as much as either PyTorch frameworks. Now, the asynchronous frameworks are faster than the non-distributed PyTorch version by around 20 seconds, while the synchronous frameworks are around 10 seconds slower. The difference between the synchronous and asynchronous versions of the framework sit at around the difference – 20 seconds – regardless of the number of workers. While we expect synchronous frameworks to be slower than their asynchronous counterparts, this is slower than we expected. It is possible that this difference comes not from increased blocking in the synchronous framework, but from a more static source, such as the overhead incurred from using AIDA's RMI communication in a highly parallel manner.

We present as well the microbenchmarks for the D-AIDA frameworks running with four workings in Figure 6.4. Compared to the two-worker version, the calculation times have gone down by nearly half in all frameworks. This is to be expected, as each worker nodes runs only half as many iterations as in the two worker case. The batch time has also decreased, by half in the synchronous frameworks and by about a third in asynchronous frameworks. As for aggregation time, it has remained the same in synchronous frameworks, whereas the asynchronous frameworks see it doubled compared to the two worker version. This is because asynchronous frameworks perform aggregation once for every iteration from every worker; due to iterations being more likely to occur in parallel, the likelihood of an aggregation needing to occur concurrently with another calculation increases. The aggregation for the synchronous frameworks remain the same – there are less aggregations over all, but the same number of model updates. However, the remaining time remains a bottleneck for all the D-AIDA frameworks, while the working time (i.e. time spent in calculation, aggregation, or batching) roughly matches the total execution time of DistDataParallel. Indeed, across all D-AIDA's frameworks, while the working time has decreased as expected, the communication time remains relatively static across both environments.

Finally, we note that we do not present the loss at the end of the algorithm runtime. The reason for this is simple – the loss is similar across all executions, and thus comparing the loss presents little value. However, we also note that the loss for PyTorch DistDataParallel is slightly higher at

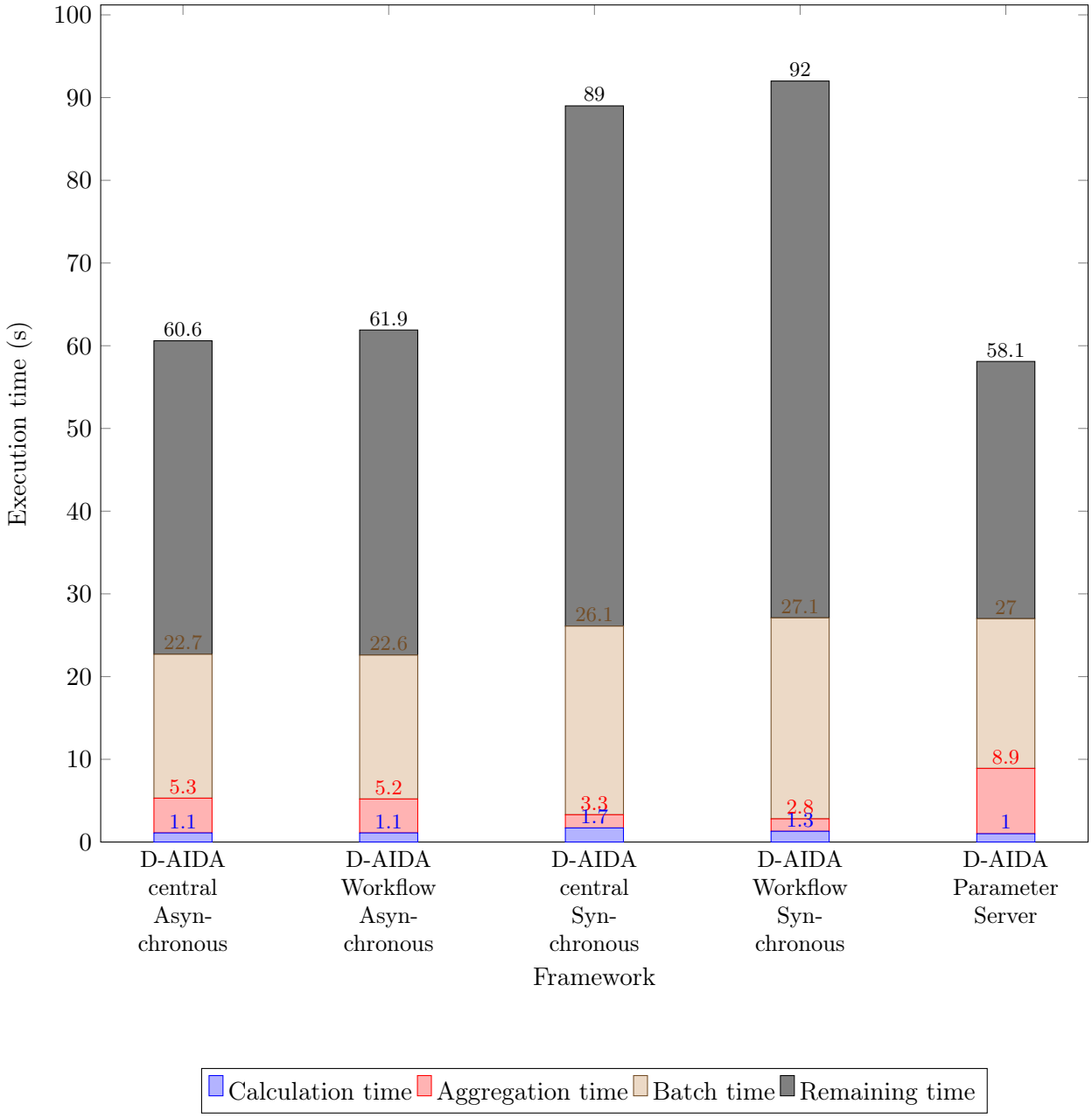

Figure 6.4: Linear regression microbenchmarks for D-AIDA with four worker nodes.

the four worker system – this is because PyTorch DistDataParallel scales its gradient updates with the number of workers in the system. In the four worker environment, only 2 500 iterations are carried out in parallel – thus, PyTorch DistDataParallel treats it as if only 2 500 iterations in total were carried out. In the other frameworks, all 10 000 iterations apply the gradient update at full magnitude, and show only the converged loss at the end of the execution. The loss in this instance converges at around 3 200 iterations, and so the two worker system, having 5 000 iterations in each

worker, does not exhibit this quality. This scaling of the gradient updates is possible in the D-AIDA frameworks as well, but must be implemented by the user.

The results of these performance tests show that the communication overhead using D-AIDA is the largest bottleneck when using the D-AIDA frameworks over distributed PyTorch frameworks. While limited by its use of the AIDA RMI module, asynchronous modules are still faster than a non-distributed implementation. We see expected decreases in both calculation time and batch time, while the time it takes for communication is relatively constant across both environments. Compared to the frameworks implemented by PyTorch, D-AIDA does not scale as well. Unlike PyTorch DistDataParallel, which overlaps communication with computation by performing gradient calculations as soon as a subset of the model parameters have received gradients from all workers, the synchronous nature of AIDA's RMI communication means there is little overlap – all gradients for the entire model must be received before the model updates begin. Additionally, AIDA RMI uses a general serialization technique which is applicable for all objects in Python – it is likely that any serialization performed by PyTorch libraries are optimized for work on PyTorch objects.

6.1.2 Matrix Factorization

Matrix factorization, described in Section 2.4.3, is an algorithm used for recommendation systems. As input, it often takes relational data in the form of reviews given to certain items by users. Compared to linear regression, matrix factorization uses a much larger model. Unlike linear regression, it does not need the entire model in order to perform prediction and loss calculations – this makes parameter servers or other model parallel tactics better suited for this algorithm.

6.1.2.1 Data

The matrix factorization experiments use the anime ratings dataset from Kaggle [6]. This is a set of ratings, ranging from 1-10, by users on anime that they have watched. The full dataset contains information on 73 516 users and 12 294 anime. Originally, the dataset contains information on users who have watched anime, but have not rated it, assigning that tuple a value of -1 in the rating. These tuples have been discarded, leaving 6 337 242 ratings. Each of these features a user id, an anime id and the rating the user gives that particular anime. The data is sorted in increasing order by user id, then by anime id, and partitioned equally when distributed. The data is not shuffled

before being partitioned, so in the case of two workers, one worker would have the first 3 168 621 ratings, and the other worker the last half of the dataset. Thus, much of the data for a particular user will exist on one data node. Due to this, it is unlikely that two nodes will update the same user in an iteration.

Like the linear regression dataset, the matrix factorization dataset is first transformed into PyTorch tensors when it is loaded in the PyTorch DataLoader. In this case, the user ids and the item ids are integers, and the ratings are doubles.

6.1.2.2 Implementation

Like linear regression, since we compare the D-AIDA framework against the PyTorch RPC and DistDataParallel implementations, we write our model in PyTorch.

```
1  import torch
2
3  class MatrixFactorization(torch.nn.Module):
4      def __init__(self):
5          import torch
6          super().__init__()
7          self.user_factors = torch.nn.Embedding(73517, 3, sparse=True)
8          self.item_factors = torch.nn.Embedding(34476, 3, sparse=True)
9
10     def forward(self, data):
11         import torch
12         user = torch.squeeze(data[:,[0]])
13         item = torch.squeeze(data[:,[1]])
14         return (self.user_factors(user) * self.item_factors(item)).sum(1)
```

Listing 6.2: Model used in experiments for matrix factorization

The listing 6.2 describes the implementation for matrix factorization in PyTorch that we use. The user and item matrices are represented by PyTorch embeddings, which act as a dictionary for embeddings. In this case, each embedding is a vector of size 3, and there are 73 517 user embeddings and 34 476 item embeddings. The length of each embedding is a hyperparameter, while the size of the dictionary is determined by the highest id number given to an item or a user in the dataset. The embeddings are stored in a sparse tensor – the gradients will also be sparse in this case. Sparse tensor gradients are stored in a Pytorch `torch.sparse_coo_tensor` object, which internally acts as

a key-value store – the keys representing non-zero indices of the tensor, and the values representing the values at those indices. This makes it so that gradients require less storage when they are sparse, and are thus more optimal for D-AIDA's frameworks which moves gradients between nodes. The forward function takes the user ids and item ids as a tuple of vectors, and returns the predicted ratings.

Like in linear regression, we use MSE loss as our loss function using the PyTorch `torch.nn.MSELoss()` function. The model is likewise updated using SGD with a learning rate of 0.1, initialized as a Pytorch optimizer `torch.optim.SGD(model.parameters(), lr=0.1)`. The loss function is initialized at each of the workers; the optimizer at the middleware. For matrix factorization, we use a minibatch size of 64, and 80 000 total iterations spread evenly across each worker. Thus, in the two worker environment, each worker would run 40 000 iterations, and in a four worker environment, each worker would run 20 000 iterations.

In the central framework, `preprocess` initializes the loss function and loads the data at each worker, while `initialize` initializes the model and optimizer at the middleware. Each iteration on a worker must be started by the middleware – since the model is passed as a parameter to the `iterate` call, the middleware does not yet know which embeddings each iteration would require. Thus, the middleware passes the entire model to each worker with the `iterate` call. The `iterate` call performs the standard loss and gradient calculations, and returns the gradient updates to the middleware at the end of the `iterate` call. The gradient updates are a list of two tensors – the gradient for the user embeddings, and the gradient for the item embeddings. The `aggregate` call on the middleware applies the gradients, and the next iteration starts.

The workflow model is much the same. First, the client initializes and stores the matrix factorization model in the middleware `DBAdapter` workspace. The `FirstStep.work` process on each worker loads the data into the dataloader and initializes the loss function, both of which are stored in the local database workspace. The `FirstStep.aggregate` function initializes the optimizer on the model, and returns the model so that it will be passed to the `work` phase of the next step. The `Iterate` step is performed repeatedly for each worker – 40 000 times each in the case of two workers. Like in the central framework, the loss and gradient calculations performed by the worker are driven by the middleware pushing and executing the `Iterate.work` function each iteration – thus, the middleware must pass the entire model as a parameter to the `Iterate.work` function each

iteration. The `Iterate.work` function returns the gradients – again, in the form of a list of tensors – where it will be applied to the model in the following `Iterate.aggregate` call by the middleware.

The parameter server framework is a natural option for the matrix factorization algorithm. The middleware starts the `run_training` method on all workers. Each worker starts by initializing the loss and dataloader, then continues on to the iterations. Each iteration starts with the client retrieving a batch of data from the loader – consisting of user ids, item ids and ratings. Then, a portion of the model is pulled from the middleware using the user ids and the item ids retrieved in the batch. The client-defined `pull` function takes as input a tuple of two tensors, representing the list of user ids and the list of items ids the worker wishes to pull. The middleware parameter server returns the embeddings indicated by the ids in another tuple. If an id appears multiple times, the embedding for that id will also be returned multiple times. Using these embeddings, the loss and gradient updates are calculated. In particular, the gradients are transformed to a `torch.sparse_coo_tensor` object. After being transformed to these sparse tensors, the gradients for the user ids and the item ids are then pushed to the middleware using the `push` operation on the parameter server. The updates are stored in a queue with max length 3, and applied to the model by another thread running in the middleware.

6.1.2.3 Results

Figure 6.5 presents the execution times, alongside their standard deviations, of the PyTorch frameworks compared with the D-AIDA frameworks in a two worker environment. We note firstly that the non-distributed implementation is faster than any distributed implementation. Due to the relatively small size of the dataset and the large amount of iterations, it is optimal to perform all the computation centrally. However, this may not be the case for larger datasets, or if the datasets cannot leave their database. Here, compared to PyTorch, D-AIDA fares relatively poorly. The central and workflow based frameworks lack the capacity for model parallelization – thus, the entire model must be sent from middleware to workers in every iteration. Comparabed to those, however, the D-AIDA parameter server framework is much faster – the ability to pull only parts of the model to the workers at each iteration speeds up the execution time by nearly 5 times compared to its asynchronous D-AIDA counterparts. The PyTorch DistDataParallel implementation performs the best – en par with the non-distributed version, if error were taken into account. Since DistData-

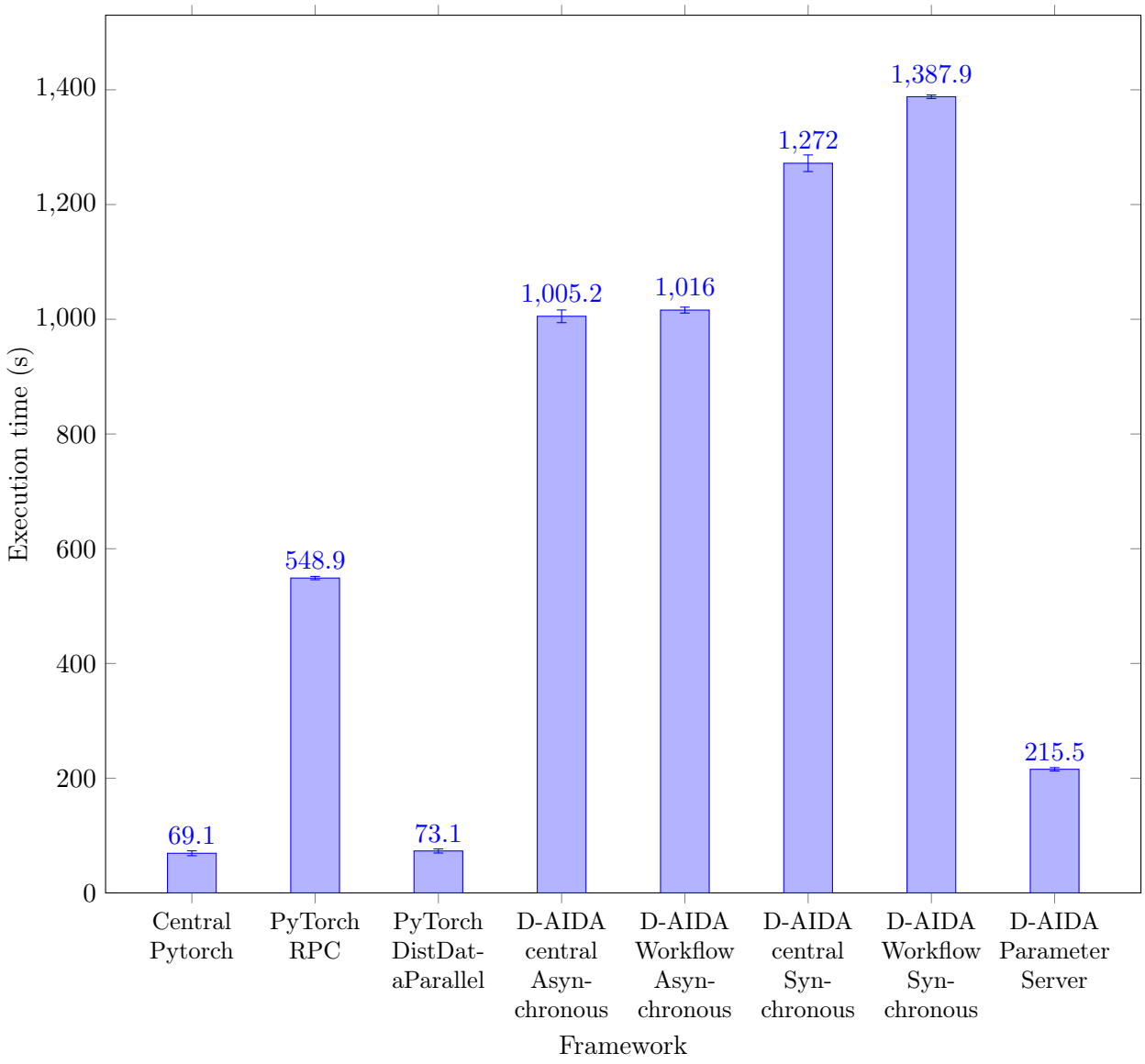

Figure 6.5: Comparison of execution times for matrix factorization between all frameworks with two worker nodes.

Parallel does not send model updates, but each worker broadcasts their parameters to the entire group, it has fewer communication steps than the parameter server framework. It sends only the non-zero gradients every iteration, so the communication overhead for DistDataParallel remains small. At nearly 8 times slower than the central version, PyTorch RPC performs much worse than the DistDataParallel version, though it beats the non-parameter server D-AIDA frameworks. In this case, since the data sent per iteration is much smaller than the model itself, it is natural that PyTorch RPC is faster than the workflow and central frameworks in D-AIDA. Just as well, it is

natural that the D-AIDA parameter server is faster than nearly all the others – it pulls only parts of the model. DistDataParallel is, in this case, still faster than the parameter server approach – every worker only needs to send and receive gradients at every iteration, whereas the parameter server framework forces each worker to request parameters, receive them, push gradients, then return from pushing gradients.

However, we note that the issue regarding the loss in DistDataParallel is prevalent here as well – while the other implementations converge to a loss value of around 5 after the total 80 000 iterations spread across the workers, the PyTorch DistDataParallel only achieves a loss of around 18 after 40 000 iterations across two worker nodes.

Compared to linear regression, the standard deviation of the execution times is smaller relative to the mean (although still larger in absolute magnitude). This is likely due to the increased amount of iterations compared to linear regression, which would reduce the effects of variance in a single iteration on the variance of the overall execution time.

Figure 6.6 presents a breakdown of the execution time for matrix factorization in a two worker environment. We see that for the central framework and the workflow framework, the majority of the time is spent in communication or blocking due to the large size of the model being passed from middleware to worker. The calculation time, aggregation time and batch time is similar across all four measurements. Due to the larger number of iterations, the time difference between synchronous and asynchronous executions are more prevalent here than in the linear regression case – with the synchronous workflows being around 300 seconds slower than their asynchronous counterparts. Compared to the previous two frameworks, the D-AIDA parameter server spends less time at every phase. Evidently, its ability to transmit only parts of the model across the network means its communication cost is much lower than the previous two frameworks. The calculation and aggregation time is much lower as well – it is possible this is due to how the parameter server framework performs the gradient calculations – backwards propagation on a small subset of the model in form of smaller tensors rather than over the entire model. While linear regression featured a bottleneck at the communication phase in D-AIDA, wherein the calculation, aggregation and batch phases of each of D-AIDA's frameworks took about as much time as the entirely of the execution for DistDataParallel, matrix factorization does not show this same characteristic. Compared to the execution time of DistDataParallel, these phases take longer in the central and

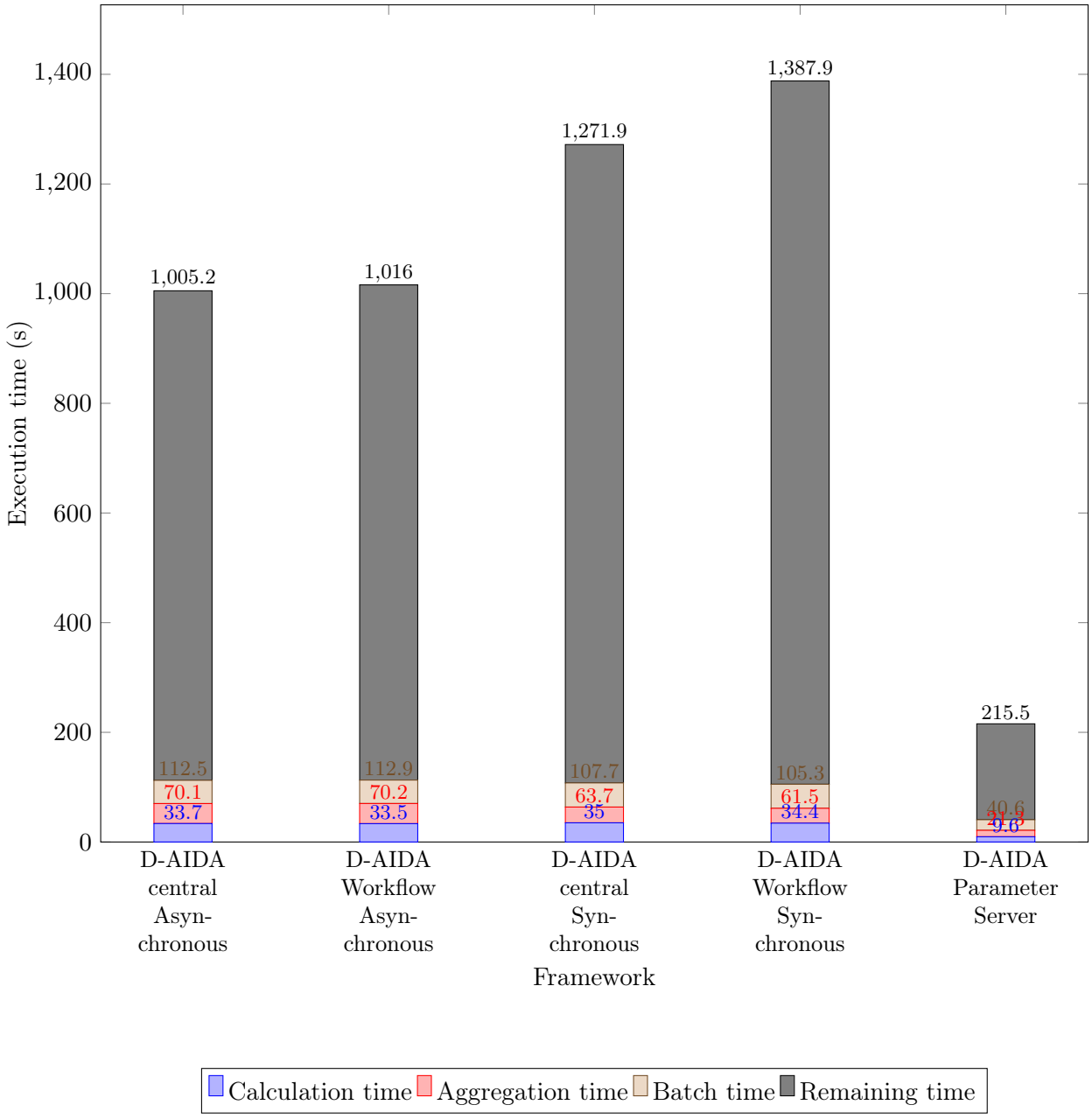

Figure 6.6: Matrix Factorization microbenchmarks for D-AIDA with two worker nodes.

workflow frameworks, and shorter in the parameter server framework. Thus, DistDataParallel displays a larger communication overhead in matrix factorization than in linear regression – normal due to the increased number of iterations and the increased message size.

Figure 6.7 presents a comparison between execution times on 4 worker nodes. Here, we see vast improvements in execution time for each of the frameworks. The most improved is PyTorch RPC, which speeds up by nearly 250 seconds – around a 1.87 times speed up. Most of this is due to the

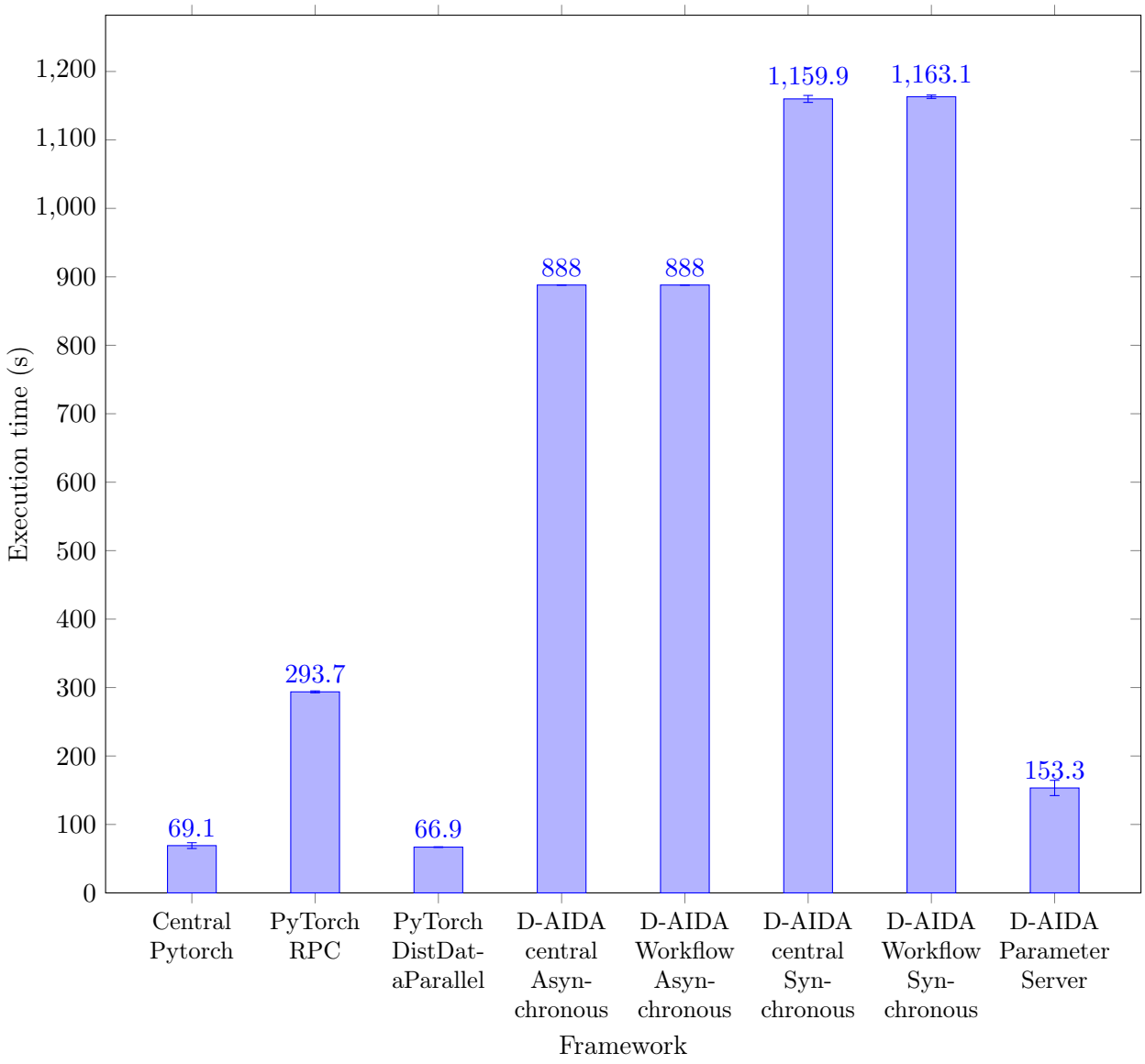

Figure 6.7: Comparison of execution times for matrix factorization between all frameworks with four worker nodes.

increased parallelism that comes with 4 workers – each worker only needs to perform half as many iterations as before. This shows that the majority of the time comes from the communication costs of moving data to the parameter server node and the batch time, since that cost can be parallelized more efficiently than the calculation and aggregation phases, which occur on a single node in this implementation. The D-AIDA implementations also speed up somewhat with 4 workers – by around 100 seconds each, which is only a 1.13 times speedup for asynchronous models and slightly less for synchronous ones. The D-AIDA parameter server framework sees an improvement by around 60

seconds – a 1.4 times speed up. Still, it lags behind DistDataParallel as the fastest implementation.

PyTorch DistDataParallel does not exhibit such a dramatic improvement – only speeding up by around 6 seconds. Notably, since each worker only runs 20 000 iterations, the loss reached by PyTorch DistDataParallel is much higher than the other frameworks; it only manages to achieve a loss of 62.76 on average compared to the other frameworks, which achieve an average loss of 4.61. Even so, PyTorch DistDataParallel shows the potential to match or even be faster than the non-distributed variation given four worker nodes.

In Figure 6.8, the amount of time spent by each D-AIDA framework in each part of the algorithm is presented. Compared to the two-worker variation with double the amount of iterations per worker, the time each worker spends in calculation and retrieving data from the dataloader is cut by almost half for every framework. This is to be expected, as the number of iterations each worker executes is also cut in half. The time spent aggregating and performing model updates in the middleware has increased in the asynchronous versions of the central and workflow frameworks, as well as the parameter server framework, due to the increase in parallelization. For the synchronous frameworks, the aggregation time has decreased, due to the lower number of iterations. In the non-parameter server frameworks, the remaining time has decreased by around 100 seconds each, while the parameter server framework reduces it by nearly 60 seconds on average. While each of the computation times – calculation, aggregation and batch – have decreased by 30% for central and workflow frameworks, and 40% for the parameter server framework, the remaining communication time has not nearly decreased as much. Since PyTorch RPC has large deceases in communication time when moving from two workers to four workers, we can assume D-AIDA has some bottleneck during communication that cannot fully parallelize the communication aspect – likely the serialization and deserialization of objects at the middleware which cannot be fully parallelized in AIDA due to Python's global interpreter lock, but can be in PyTorch due to its implementation in C.

In conclusion, we see that the D-AIDA frameworks perform on average worse than the PyTorch DistDataParallel framework, and only the parameter server framework performs better than PyTorch RPC. From the difference exhibited between the two worker and four worker environments, we see that the D-AIDA frameworks show some bottleneck during the communication phase that causes poorer parallelization than during other phases of execution. The communication overhead is heavy in the D-AIDA frameworks, especially in the central and workflow frameworks which do

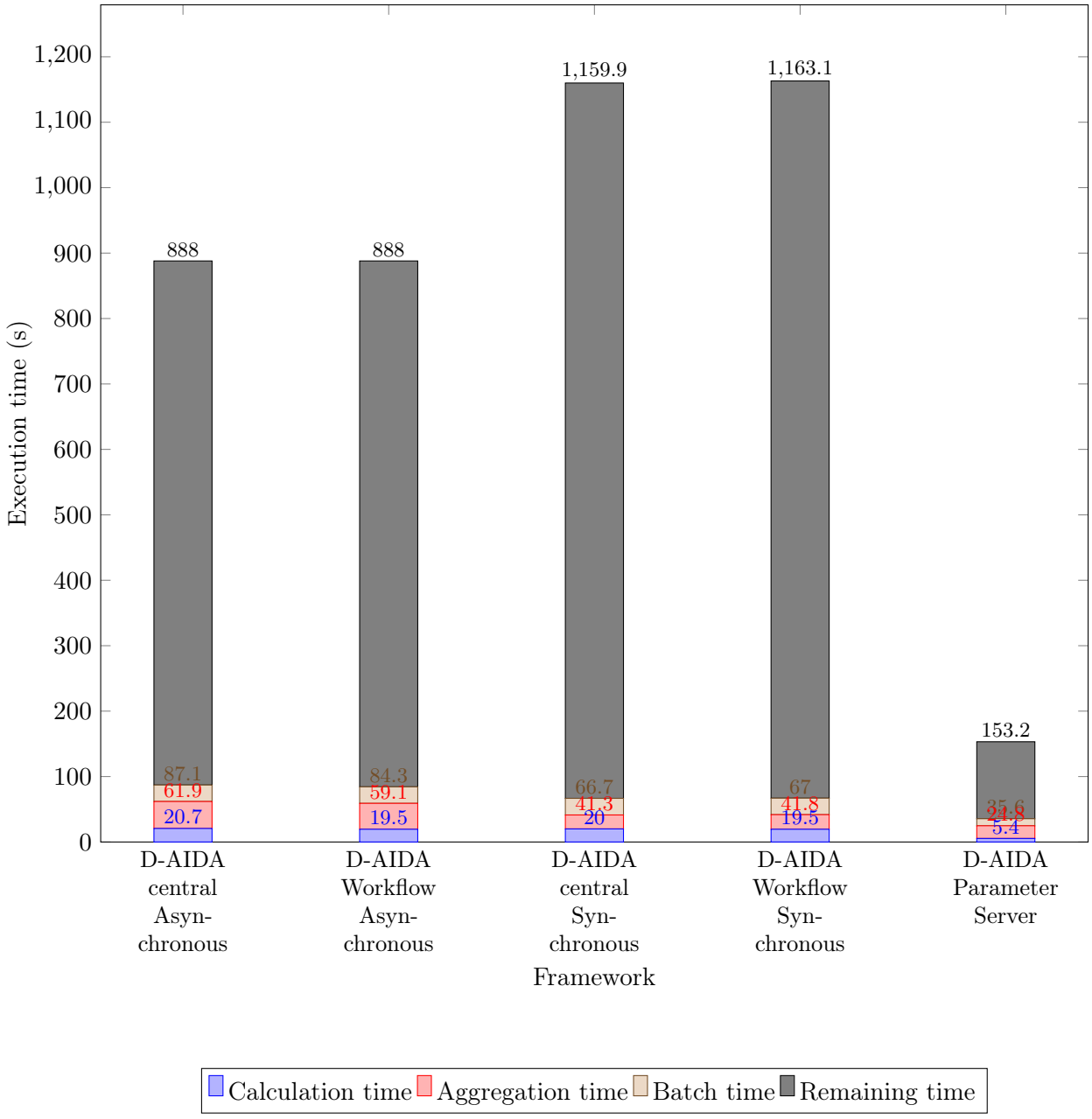

Figure 6.8: Matrix Factorization microbenchmarks for D-AIDA with four worker nodes.

not support model parallelism. In an algorithm like matrix factorization which uses such a large model, the parameter server framework is the only viable approach. It should be noted that, while PyTorch DistDataParallel performs the same number of iterations across each worker much faster than the D-AIDA frameworks, the actual performance of the model remains much worse due to the scaling of the gradient updates. There is no option in PyTorch to prevent such scaling, so models achieving the same level of accuracy as those trained by the D-AIDA frameworks must go through

many more iterations.

6.1.3 Conclusion

Overall, the performance of distributed linear regression and matrix factorization in D-AIDA compared to the PyTorch distributed implementations is not favorable. While the scaling of the calculation phases can match the overall execution time by PyTorch DistDataParallel, the communication overhead is extreme. The D-AIDA RMI was originally meant to optimize the transferal of small amounts of data and facilitate remote execution – using it for large or repetitive data transferals is shown to be suboptimal here. Further evaluation of the runtime in the D-AIDA implementations is required in order to identify other potential bottlenecks, such as those imposed by the Python global interpreter lock, or time spent serializing data. These and other such disadvantages may be hidden by the "remaining time" portion of the runtime evaluations presented in this section.

Additionally, Pytorch implementations – especially DistDataParallel – allow an easier implementation of distributed algorithms if using a PyTorch model. Writing a PyTorch algorithm using DistDataParallel is as simple as creating a wrapper around a PyTorch model and initializing a communication group – whereas for D-AIDA, the user must manually write the code for retrieving gradients from PyTorch tensors and applying them to the model. However, the deployment of D-AIDA frameworks is easier than deployment of PyTorch DistDataParallel or RPC, since the client need only write the code at a local site and interact with the D-AIDA middleware, which will automatically distribute the rest to the workers. In distributed PyTorch, the code must be present at all sites and run at each.

While the PyTorch implementations – especially DistDataParallel – are faster than D-AIDA's, they are more rigid as well. Distributed PyTorch, naturally, works only on PyTorch models and PyTorch models mostly implement iterative algorithms. In the next section, we perform experiments on clustering, a non-iterative algorithm which is not supported by PyTorch.

6.2 Clustering

Clustering is a more traditional, non-iterative unsupervised algorithm common on relational data. Instead of training a global model, it seeks to instead create labels for unlabelled data based on

similarities in the data. In this section, we perform distributed clustering experiments with D-AIDA by implementing the DBDC algorithm described in Section 2.5.3.

6.2.1 Data

The data used for clustering is generated using the `make_blobs` function provided by the SciKit-Learn `sklearn.datasets` library. We generate 4 000 000 samples in total, with four features each, spread across nine clusters. The standard deviation of each cluster is 32, and each feature of the cluster centers generated are bounded by $(-750, 750)$. These values are chosen arbitrarily. In two worker systems, each database holds 2 000 000 data points and in four worker systems, each database holds 1 000 000 data points. The data points are shuffled before being distributed. On partitioning the data, it is possible that more clusters are formed as subsets of each cluster are sent to different nodes. Thus, a global step is required to amalgamate any local clusters found by each worker that may overlap.

Each data point is stored in a database table containing four columns – one for each feature. The features are stored as doubles, and accessed by the SciKit-Learn functions using the `.matrix` property of the TabularData object. Like in the iterative algorithms, the non-distributed test case outside of D-AIDA will read the data from a csv file. The time spent retrieving the data, in either the non-distributed and distributed cases, will not be included in the total execution time.

6.2.2 Implementation

As clustering is a non-iterative algorithm that does not build upon a single model, both the central and parameter server frameworks offered by AIDA do not support this form of algorithm. However, the workflow framework was especially designed to support such algorithms that are not able to be managed by other frameworks. Thus, experiments for distributed clustering are performed using an algorithm written using the D-AIDA workflow framework. Additionally, PyTorch does not natively support any clustering algorithms, so a comparison between PyTorch and D-AIDA frameworks will not be performed here.

For distributed clustering, we implement the DBDC algorithm described in Section 2.5.3. As a reminder, this algorithm consists of three overall steps – a local clustering step at each of the workers, a global clustering step at the middleware which takes the clusters from the first step and

```
1     @staticmethod
2     def work(dw, data, context=None):
3         from sklearn.cluster import DBSCAN, KMeans
4         import numpy as np
5
6         dw.matrix_data = data.matrix.T
7         db = DBSCAN(eps=16, min_samples=3).fit(dw.matrix_data)
8         classes = []
9         for i in range(len(np.unique(db.labels_))-1):
10            classes.append(dw.matrix_data[np.where(db.labels_ == i)])
11        clusts = [KMeans(n_clusters=3).fit(c) for c in classes]
12        rep = []
13        for j in range(len(clusts)):
14            centers = clusts[j].cluster_centers_
15            dists = []
16            for i in range(len(centers)):
17                sub_clust = classes[j][np.where(clusts[j].labels_ == i)]
18                dist = max([np.linalg.norm(centers[i] - point) for point in sub_clust])
19                dists.append(dist)
20            rep += zip(centers, dists)
21        return rep
```

Listing 6.3: Work phase of first step for DBDC implmentation in workflow framework.

finds global clusters, and then a last re-labelling step at each of the workers which adjusts the cluster labels given to the local data according to the global clusters found in the step previous. As each of these steps require waiting for all of the results from the previous steps, this algorithm in run synchronously using the workflow framework.

The `work` phase of the first step implements the local clustering on each worker, as well as the determination of the local representatives of each local cluster, to be sent to the middleware. The code for this first phase can be found in Listing 6.3. First, DBSCAN, implemented by the SciKit-Learn library, is ran on the data to find the local clusterings at line 7. In this instance, we run DBSCAN with $\epsilon = 16$ and $minPts = 3$. Next at line 11, after the clusters have been found in the local data, the representatives for each cluster are determined using K-means, also implemented by SciKit-Learn. K-means is run on each cluster with $k = 3$ to find 3 subcluster centers – these 3 would become representatives for the entire cluster. From here, each subcluster center determined by K-means finds the maximum distance between it and every other point in the subcluster. The results are stored in a list of tuples matching the cluster centers and the distances. This is performed

```
22      @staticmethod
23      def aggregate(dw, results, context):
24          from sklearn.cluster import DBSCAN
25
26          centers = sum(results, [])
27          epsilon = max([r[1] for r in centers])
28
29          db = DBSCAN(eps=epsilon, min_samples=2).fit([r[0] for r in centers])
30
31          return db
```

Listing 6.4: Aggregate phase of first step for DBDC implmentation in workflow framework.

```
32      @staticmethod
33      def work(dw, data, context=None):
34          from scipy import spatial
35          import numpy as np
36
37          db_results = context['previous']
38          tree = spatial.cKDTree(dw.matrix_data)
39          labels = np.asarray([-1] * len(dw.matrix_data))
40          eps = db_results.eps
41          centers = zip(db_results.components_, db_results.labels_)
42          for point, label in centers:
43              cluster = tree.query_ball_point(point, eps)
44              labels[cluster] = label
45          l = dw._L(lambda d: d, {"label": labels})
46          return data.hstack([l])
```

Listing 6.5: Work phase of the second step for DBDC implmentation in workflow framework.

in lines 13-20.

Following the DBDC algorithm, the `aggregate` phase of the first step takes all the cluster representatives and distances from all the workers. This step is shown in Listing 6.4. At line 27, the maximum distance for all representatives is found, and set as the ϵ. DBSCAN is run again, using that ϵ and $minPts = 2$ as arguments. This finds the global clustering, and this global clustering is then passed to the workers in the next step.

The final step in the DBDC algorithm involves the worker nodes assigning their local data points to the global clusters found in the previous step. This is shown in Listing 6.5. The global clustering is retrieved from the `context['previous']` variable at line 37. From there, a kd-tree, implemented

by the SciPy library, is used to index the local data in order to find nearest neighbors of each data point quickly. The cluster centers, their labels, and the ϵ is retrieved from the global clustering. In lines 42-44, the labels are applied to the data by going through the cluster centers and assigned all data points to the cluster label if they are within ϵ distance of the cluster center. Lines 45-46 merely load the labels as another TabularData object, and append them as a column to the original data points. The final `aggregate` phase of the second step involves taking the TabularData of labelled data returned from the `work` phase and wrapping them in a DistTabularData object for the user to manage.

We compare the execution of this algorithm on two and four worker AIDA nodes. Additionally, we compare it to a non-distributed clustering algorithm – DBSCAN, running on a single node. The non-distributed DBSCAN will use $\epsilon = 16$ and $minPts = 3$, the same hyperparameter values used as in the first step of the D-AIDA workflow implementation. The DBDC algorithm is more complex than DBSCAN, so a comparison between the two is flawed. Nevertheless, this is the best comparison to be had, as DBDC uses a similar metric as DBSCAN for clustering, and indeed uses a local DBSCAN when performing the inital clustering at each worker node.

6.2.3 Results

Figure 6.9 presents a comparison of execution times for a non-distributed clustering using DBSCAN, DBDC run using the D-AIDA workflow framework on two worker nodes, and DBDC run using the D-AIDA workflow framework on four worker nodes. The standard deviation of the entire execution time is presented as an error bar. For each distributed algorithm, the different steps of it has been timed. The "local clustering" step refers to the first step where each worker node performs the local clustering using DBSCAN and finds the cluster representatives using K-Means. The "global clustering" step refers to when the middleware finds the global clusters from the cluster representatives sent by the workers. The "local cluster readjustment" step refers to the final step when each worker assigns each data point to a cluster found by the previous global clustering step. The "remaining time" bar refers to any time not measured – the communication time and the time spent blocking.

As Figure 6.9 shows us, the majority of the execution time is in the first step: local clustering. Compared to the non-distributed version, both the distributed versions are much faster in this

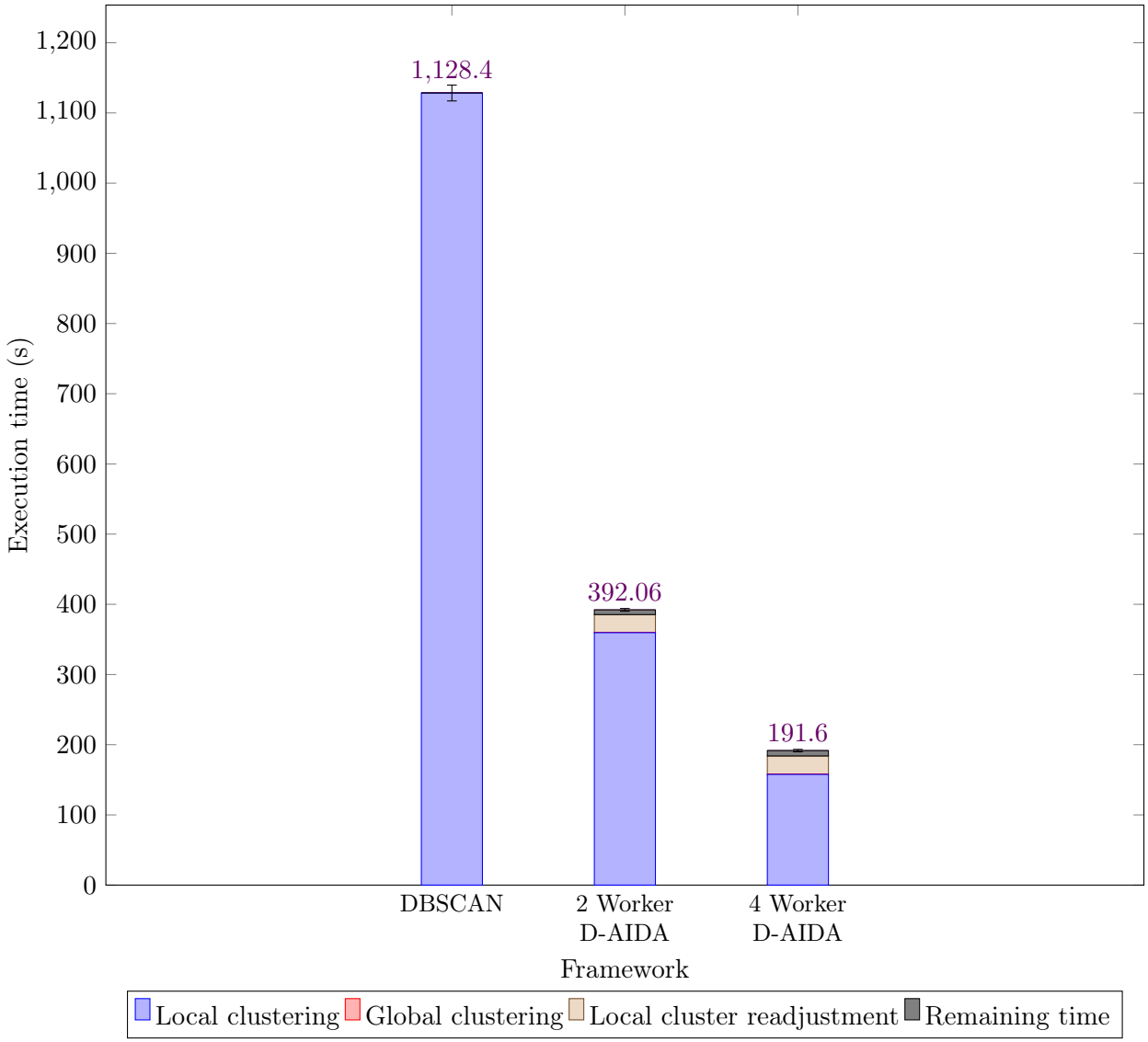

Figure 6.9: Comparison of clustering times

regard – in the distributed version, each worker only needs to work on a subset of the data, and this can be completed in parallel. The non-distributed version must run DBSCAN on 4 million data points – in the 2 worker environment, each worker only needs to run DBSCAN on 2 million data points, and in the 4 worker environment, only 1 million. The next largest time share is the final step, the local cluster readjustment. This only occurs in the distributed versions, and involves finding where every data belongs in the global clusters. Compared to those two steps, the global clustering and communication takes very little time. Taking only the local clustering time into consideration, the two worker environment is around 3 times as fast as the the non-distributed environment, and

the four worker environment 3 times as fast as the two worker environment. This is a super-linear increase – and to be expected, given the average runtime of DBSCAN to be O($nlogn$). As the first local clustering step performs the same DBSCAN algorithm on each of the worker nodes, only on smaller partitions of the dataset, this is a fairly meaningful comparison. The additional time – spent either in communication, the global clustering step, and the local cluster readjustment step – can be seen as overhead from the distribution.

In these experimental times, the non-distributed DBSCAN presents the highest degree of variation in its observed times. Indeed, the majority of the runtime, and therefore the variation in runtime, is cause by the local DBSCAN clustering algorithms run on each node in the distributed versions as well. As the time taken for DBSCAN decreases, so too does the runtime variation.

We note finally that we do not compare the clusters found by each algorithm. For one, DBSCAN and DBDC are different algorithms which are not guaranteed to find the same clusters. For another, there is no definite measure of the goodness of a clustering, thus, it is difficult to adequately compare two separate clusterings.

6.2.4 Conclusion

Distributed clustering works out very well for D-AIDA. Unlike matrix factorization and linear regression, there is no well known library in Python supporting distributed clustering. In our implementation, we prove that it is simple to implement a version of distributed clustering using one of D-AIDA's frameworks, and it performs better than a non-distributed version.

Compared to the iterative algorithms, clustering does not have so much communication – the majority of the time is spent in calculation. Thus, the RMI is not as big of a bottleneck here, and indeed, the performance by D-AIDA here is much better than in the non-distributed case. Furthermore, there is no library offering an implementation of DBDC, but its implementation in the D-AIDA workflow framework was quite simple. A user needs only identify the separate steps to be performed at the local level by the workers, or at the global level by the middleware, and D-AIDA will distribute the execution accordingly.

7

Conclusion

7.1 Contributions and Findings

In this thesis, we implemented a proof of concept for a system that performs distributed relational queries and offers frameworks for writing distributed machine learning algorithms on relational data that is distributed across several database systems. Each database system hosts an embedded Python interpreter that can execute Python code with minimal data transfer out of the database. This distribution of the data across these database nodes will be hidden from the user, and managed by a new middleware server.

We have demonstrated the viability of implementing distributed relational operations by implementing several basic ones using D-AIDA – including selections, projections, aggregations and two distributed join implementations. D-AIDA's middleware architecture allows the client to access a global view of partitioned data through the DistTabularData object, and allows them to perform

simple relational operations on tables that have been partitioned horizontally. Computation of relational operators will be transparently pushed down to the worker nodes and executed by the RDBMS, using its query optimization engine.

Several frameworks designed for distributed machine learning have been implemented in D-AIDA. We have also proven the viability of these frameworks through implementing several algorithms using them. The D-AIDA middleware server offers a site to manage a global model which will be trained by the worker nodes. Common algorithms using relational data, such as linear regression, matrix factorization and clustering, can be implemented using the various frameworks provided by D-AIDA. Additionally, by testing in environments with different number of workers, we have shown that algorithms implemented using these frameworks have the potential to scale well.

Furthermore, both these features will use the same API – the DistTabularData object, which indicates how the data tables are partitioned. Using this, relational queries and user-defined algorithms can be automatically distributed to the nodes where the data resides. This also allows the interweaving of relational queries and machine learning algorithms without unnecessary data transferal – something novel to this framework.

7.2 Future Work

Further experiments could be done using the D-AIDA framework to test its ability to host multiple different algorithms. While we have explored gradient methods and clustering in this thesis, future work could explore ensemble methods where each worker node hosts a local model whose predictions can be aggregated by a global model to produce a singular prediction. Additionally, machine learning methods such as hyperparameter searching, early stopping and regularization have not been thoroughly explored in the D-AIDA system. While this thesis has focused much on the training aspect of machine learning, further research could be performed on prediction phase execution in such a system.

The D-AIDA design still has some flaws that could be fixed in future work. Currently, no optimization exists in the middleware for distributed relational queries. As indicated by Kossman [26], a middleware architecture could have a query optimizer at the middleware that decides how best to optimize a query. Even without a query optimizer, the current implementations of distributed

queries in D-AIDA could be optimized. For instance, complex aggregations and group bys currently pull all data from the database nodes to the middleware to be executed – this could be optimized by having the aggregations be performed at the worker nodes, and their results further aggregated by the middleware.

Additionally, we find that the AIDA RMI communication library may not be sufficient for all machine learning algorithms. Currently, all RMI communication is synchronous – that is, a client performing a request on the server blocks until a response a received. By introducing an asynchronous call – a client making a request on the server, but can continue execution until they require the response – further optimizations can be made to distributed execution by allowing more overlap between the computation and communication phases. Additionally, RMI is a peer-to-peer communication system. By implementing a broadcast message, other paradigms such as AllReduce could become viable. Thus, add an AllReduce communication library into D-AIDA and developing a learning framework based on it would definitely be worthwhile.

Other than those proposed, more research could be pursued in identifying the exact cause of D-AIDA's underwhelming performance in distributed machine learning compared to that of PyTorch. While some effort has been made in this thesis, further improvements to D-AIDA will likely need to study the time difference between the recorded computation time and the total execution time – labelled in this thesis as "remaining time". Identifying exactly what is taking up this time would help steer future development of the D-AIDA framework.

For now, all machine learning frameworks support by D-AIDA offer a global model hosted by the middleware which the worker nodes must update. Allowing for model replication across the workers could open the doors for more sophisticated frameworks, such as the parameter server offered by NuPS [40] or the AllReduce gradient update implemented by PyTorch DistDataParallel [28].

Furthermore, the D-AIDA middleware server can be extended to run not only on a dedicated middleware node, but also the worker nodes. In this case, every worker node can act as a middleware server for the client to connect to, granting the client access to the entire system no matter which node they connect to. Further optimizations for relational queries and machine learning operations could be performed depending on whether the middleware server the client accesses also hosts data.

Finally, another direction that could be explored by D-AIDA is failure handling in the distributed system. Data replication across different nodes is a common form of ensuring data is available

despite any one node failure – however, in the current implementation, we assume that data is non-replicated. The training should occur only on the primary copy of each data point so as to avoid bias from reading multiple copies of the same data point. Many distributed learning frameworks support the use of checkpoints so as to be able to restore trained model parameters in the event of a failure – this too has been neglected in the current implementation. Currently, the middleware serves as a single point of failure for the trained model. Model replication across the system could resolve this issue, in addition to potentially improving the training speed.

Bibliography

[1] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. Spark sql: Relational data processing in spark. In *2015 Association for Computing Machinery Special Interest Group on Management of Data (ACM SIGMOD) International Conference on Management of Data* (2015), pp. 1383–1394.

[2] Arthur, D., and Vassilvitskii, S. K-means++ the advantages of careful seeding. In *Eighteenth Annual Association for Computing Machinery-Society (ACM) for Industrial and Applied Mathematics Symposium on Discrete Algorithms* (2007), pp. 1027–1035.

[3] Bayer, R., and McCreight, E. Organization and maintenance of large ordered indices. In *1970 Association for Computing Machinery Special Interest Group on File Description & Translation (ACM SIGFIDET) (Now Special Interest Group on Management of Data (SIGMOD)) Workshop on Data Description, Access and Control* (1970), Association for Computing Machinery, p. 107–141.

[4] Chamberlin, D. D., and Boyce, R. F. Sequel: A structured english query language. In *1974 Association for Computing Machinery Special Interest Group on File Description & Translation (ACM SIGFIDET) (Now Special Interest Group on Management of Data (SIGMOD)) Workshop on Data Description, Access and Control* (1974), Special Interest Group on File Description & Translation '74, Association for Computing Machinery, p. 249–264.

[5] Codd, E. F. A relational model of data for large shared data banks. *Communications of the Association for Computing Machinery (ACM) 13*, 6 (June 1970), 377–387.

[6] CooperUnion. Anime recommendations database. `https://www.kaggle.com/datasets/CooperUnion/anime-recommendations-database`, Dec 2016.

[7] Damania, P., Li, S., Desmaison, A., Azzolini, A., Vaughan, B., Yang, E., Chanan, G., Chen, G. J., Jia, H., Huang, H., et al. Pytorch rpc: Distributed deep learning built on tensor-optimized remote procedure calls. *Machine Learning and Systems 5* (2023).

[8] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., et al. Large scale distributed deep networks. *Advances in Neural Information Processing Systems 25* (2012), 1223–1231.

[9] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Communications of the Association for Computing Machinery (ACM) 51*, 1 (2008), 107–113.

[10] Delua, J. Supervised vs. unsupervised learning: What's the difference? `https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning`, Mar 2021.

[11] D'silva, J. V. *AIDA: An Agile Abstraction for Advanced In-Database Analytics*. PhD thesis, McGill University, 2020.

[12] D'Silva, J. V., De Moor, F., and Kemme, B. Aida: Abstraction for advanced in-database analytics. *Very Large Data Bases (VLDB) Endowment 11*, 11 (2018), 1400–1413.

[13] Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research 12*, 7 (2011), 2121–2159.

[14] Ester, M., Kriegel, H.-P., Sander, J., and Xu, X. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Second International Conference on Knowledge Discovery and Data Mining* (1996), KDD'96, AAAI Press, p. 226–231.

[15] Estivill-Castro, V. Why so many clustering algorithms: A position paper. *Association for Computing Machinery Special Interest Group on Knowledge Discovery and Data Mining (ACM SIGKDD) Explorations Newsletter 4*, 1 (Jun 2002), 65–75.

[16] Fedus, W., Zoph, B., and Shazeer, N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *The Journal of Machine Learning Research 23*, 1 (2022), 5232–5270.

[17] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *Nineteenth Association for Computing Machinery (ACM) Symposium on Operating Systems Principles* (2003), pp. 29–43.

[18] Haas, L. M., Freytag, J. C., Lohman, G. M., and Pirahesh, H. Extensible query processing in starburst. In *1989 Association for Computing Machinery Special Interest Group on Management of Data International Conference on Management of Data (ACM SIGMOD)* (1989), pp. 377–388.

[19] Hernán, M. A., Hsu, J., and Healy, B. A second chance to get causal inference right: A classification of data science tasks. *CHANCE 32*, 1 (2019), 42–49.

[20] Hipp, R. D. SQLite. `https://www.sqlite.org/index.html`, 2020.

[21] Ho, Q., Cipar, J., Cui, H., Lee, S., Kim, J. K., Gibbons, P. B., Gibson, G. A., Ganger, G., and Xing, E. P. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in Neural Information Processing Systems* (2013), C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, Eds., vol. 26, Curran Associates, Inc., pp. 1223–1231.

[22] Idreos, S., Groffen, F., Nes, N., Manegold, S., Mullender, K. S., and Kersten, M. L. Monetdb: Two decades of research in column-oriented database architectures. *Institute of Electrical and Electronics Engineers Data Eng. Bull. 35*, 1 (2012), 40–45.

[23] Januzaj, E., Kriegel, H.-P., and Pfeifle, M. Dbdc: Density based distributed clustering. In *Advances in Database Technology-EDBT 2004: 9th International Conference on Extending Database Technology, Heraklion* (2004), Springer, pp. 88–105.

[24] Jiang, T., Gradus, J. L., and Rosellini, A. J. Supervised machine learning: a brief primer. *Behavior Therapy 51*, 5 (2020), 675–687.

[25] Koren, Y., Bell, R., and Volinsky, C. Matrix factorization techniques for recommender systems. *Computer 42*, 8 (2009), 30–37.

[26] Kossmann, D. The state of the art in distributed query processing. *Association for Computing Machinery (ACM) Computing Surveys 32*, 4 (Dec 2000), 422–469.

[27] Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 583–598.

[28] Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[29] Lloyd, S. Least squares quantization in pcm. *Institute of Electrical and Electronics Engineers Transactions (IEEE) on Information Theory 28*, 2 (1982), 129–137.

[30] MacQueen, J., et al. Some methods for classification and analysis of multivariate observations. In *Fifth Berkeley Symposium on Mathematical Statistics and Probability* (1967), vol. 1, pp. 281–297.

[31] McKinney, W., et al. pandas: a foundational python library for data analysis and statistics. *Python for High Performance and Scientific Computing 14*, 9 (2011), 1–9.

[32] Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research 17*, 1 (2016), 1235–1241.

[33] NSHipster. Dbscan. `https://github.com/NSHipster/DBSCAN`, 2021.

[34] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

[35] Patarasuk, P., and Yuan, X. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing 69*, 2 (2009), 117–124.

[36] Patel, A. B., Birla, M., and Nair, U. Addressing big data problem using hadoop and map reduce. In *2012 Nirma University International Conference on Engineering (NUiCONE)* (2012), Institute of Electrical and Electronics Engineers, pp. 1–5.

[37] Qian, N. On the momentum term in gradient descent learning algorithms. *Neural Networks 12*, 1 (1999), 145–151.

[38] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2022.

[39] Recht, B., Re, C., Wright, S., and Niu, F. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in Neural Information Processing Systems 24* (2011), 693—-701.

[40] Renz-Wieland, A., Gemulla, R., Kaoudi, Z., and Markl, V. NuPS: A parameter server for machine learning with non-uniform parameter access. In *2022 International Conference on Management of Data* (jun 2022), Association for Computing Machinery, pp. 481–495.

[41] Ruder, S. An overview of gradient descent optimization algorithms. *arXiv e-prints* (Sept. 2016), arXiv:1609.04747.

[42] Smola, A., and Narayanamurthy, S. An architecture for parallel topic models. *Very Large Data Bases (VLDB) Endowment 3*, 1-2 (2010), 703–710.

[43] Varma, R. Implementing a parameter server using distributed rpc framework. `https://pytorch.org/tutorials/intermediate/rpc_param_server_tutorial.html`.

[44] Verbraeken, J., Wolting, M., Katzy, J., Kloppenburg, J., Verbelen, T., and Rellermeyer, J. S. A survey on distributed machine learning. *Association for Computing Machinery (ACM) Computing Surveys 53*, 2 (March 2020).

[45] Wang, X. Lad: A locality-aware dataframe. Master's thesis, McGill University, 2023.

A
Appendix

```
class LinearRegressionModel:
    # Initialize model and learning rate
    def __init__(self):
        self.weights = None
        self.lr = 0.05

    # Separate training data and labels
    @staticmethod
    def preprocess(db, data):
        x = data.project(('x1','x2','x3','x4','x5'))
        y = data.project(('y'))
        x_bias = db._ones(x.shape[0]).hstack(x)
        return (x_bias, y)

    # Initialize model using preprocessed data
    def initialize(self, data):
        import numpy as np
        x = data[0]
        if self.weights is None:
            self.weights = np.ones((1,x.shape[1]))
        else:
            if self.weights.shape[0] + 1 != x.shape[1]:
                raise ValueError("Model weights are not the same dimension as
                ↪  input.")

    @staticmethod
    def iterate(db, data, weights):
        import numpy as np

```

```
        x = data[0].matrix.T
        y = data[1].matrix
        batch_size = 64

        # Retrieve data batches
        batch = np.random.choice(x.shape[0], 64, replace=False)
        batch_x = x[batch, :]
        batch_y = y[batch].reshape(batch_size, 1)

        # Make predictions and perform gradient descent
        preds = batch_x @ weights.T
        grad_desc_weights = (-2/batch_size) * (batch_x.T @ (batch_y - preds))
        return grad_desc_weights

    def aggregate(self, results):
        # Synchronous aggregation -- results is list of gradients from each worker
        if self.sync:
            n = len(results)
            for i in range(n):
                self.weights = self.weights - (self.lr * results[i].T / n)
        else:  # Asynchronous aggregation -- results is single gradient
            self.weights = self.weights - (self.lr * results.T)

from aida.aida import *
dw = AIDA.connect('middleware', 'database', 'username', 'password', 'lr')

model = LinearRegressionModel
data = dw.lr_data

service = dw._RegisterModel(model)

# Send start model training, specifying number of iterations
service.fit(data, 5000, sync=True)
```

Listing A.1: Example of linear regression written in NumPy using the central framework in D-AIDA.

```
import torch
from aida.aida import *

class LinearRegression(torch.nn.Module):
    def __init__(self, input_size, output_size):
        super().__init__()
        self.linear = torch.nn.Linear(input_size, output_size)

    def forward(self, input):
        return self.linear(input)

dw = AIDA.connect('middleware', 'database', 'username','password', 'lr')
dw.lr_model = LinearRegression(5, 1)

class FirstStep():
    # Preprocess data at the workers, initialize batch iterator and loss function
    @staticmethod
    def work(dw, data, context=None):
        import torch

        data.makeLoader([('x1', 'x2', 'x3', 'x4', 'x5'), 'y'], 1000)
        dw.iterator = iter(data.getLoader())
        dw.loss = torch.nn.MSELoss()
        return

    # Initialize optimizer at middleware
    @staticmethod
    def aggregate(dw, results, context):
        import torch
        dw.optimizer = torch.optim.SGD(dw.lr_model.parameters(), lr=1e-3)
        return dw.lr_model

class Iterate():
    # Perform forward and backward pass at workers
    @staticmethod
    def work(dw, data, context):
        import torch

        model = context['previous']
        try:
            batch, target = next(dw.iterator)
        except StopIteration:
            dw.iterator = iter(data.getLoader())
            batch, target = next(dw.iterator)

        preds = model(torch.squeeze(batch).float())
```

```
47          loss = dw.loss(torch.squeeze(preds), target)
48          loss.backward()
49          grads = []
50          for param in model.parameters():
51              grads.append(param.grad)
52          return grads
53
54      # Perform gradient update at middleware
55      @staticmethod
56      def aggregate(dw, results, context):
57          dw.optimizer.zero_grad()
58          for r in results:
59              for grad, param in zip(r, dw.lr_model.parameters()):
60                  param.grad = grad
61          dw.optimizer.step()
62          return dw.lr_model
63
64  job = [FirstStep(), (Iterate(), 50000)]
65
66  dw._workAggregateJob(job, dw.lr_data)
```

Listing A.2: Linear Regression using PyTorch with D-AIDA's workflow framework.

```
import torch
from aida.aida import *

class MatrixFactorization(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.user_factors = torch.nn.Embedding(1500, 3, sparse=True)
        self.item_factors = torch.nn.Embedding(2000, 3, sparse=True)

    def forward(self, data):
        user = data[0]
        item = data[1]
        return (self.user_factors(user) * self.item_factors(item)).sum(1)

class CustomMF:
    def __init__(self, model):
        import torch
        self.model = model
        self.optimizer = torch.optim.SGD(self.model.parameters(), lr=0.1)

    def pull(self, param_ids):
        return (self.model.user_factors(param_ids[0]),
        ↪  self.model.item_factors(param_ids[1]))

    # Model update at middleware
    def update(self, update):
        self.model.user_factors.grad = update[0]
        self.model.item_factors.grad = update[1]
        self.optimizer.step()
        self.optimizer.zero_grad()

    # All iterations at worker
    @staticmethod
    def run_training(con, ps, data):
        import torch

        # Data preprocessing
        data.makeLoader((['user_id', 'movie_id'], 'rating'), 64)
        x = iter(data.getLoader())
        iterations = 40000
        loss_fun = torch.nn.MSELoss()

        for i in range(iterations):
            try:
                data, rating = next(x)
            except StopIteration:
```

```
46                 x = iter(data.getLoader())
47                 data, rating = next(x)
48
49             users = torch.squeeze(batch[:, [0]])
50             items = torch.squeeze(batch[:, [1]])
51
52             # Pull parameters from middleware
53             factors = ps.pull((users, items))
54             preds = (factors[0] * factors[1]).sum(1)
55             loss = loss_fun(preds, torch.squeeze(rating))
56             loss.backward()
57             grads = []
58
59             # Put gradients into sparse tensor
60             grads.append(torch.sparse_coo_tensor(torch.unsqueeze(users, dim=0),
               ↪  factors[0].grad, (1500,3)))
61             grads.append(torch.sparse_coo_tensor(torch.unsqueeze(items, dim=0),
               ↪  factors[1].grad, (2000,3)))
62             ps.push(grads)
63
64 dw = AIDA.connect('middleware', 'database', 'username', 'password', 'mf')
65 server = dw._MakeParamServer(MatrixFactorization, CustomMF)
66 data = dw.mf_data
67 server.start_training(data)
```

Listing A.3: Matrix Factorization written using the D-AIDA parameter server framework.