LAD: A Locality-Aware Dataframe

Xiaohan Wang

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

June 2023

Abstract

Pandas is one of the most popular frameworks for data processing and analysis. To work with Pandas, data scientists typically load data that originally resides in various locations into the local execution environment. If the data is in a relational database system (DBS), some relational operations can be manually pushed to the DBS to retrieve fewer tuples from the database. This can only happen when the data scientists know exactly what they want to achieve with the data, which is usually not the case during the data exploratory phase. In our thesis, we present a client-side data analytic framework, LAD, which stands for Locality-Aware Dataframe, that can handle computation over distributed data more dynamically and in a transparent manner. We use the same API as Pandas, offering a dataframe abstraction for transformation and analytics. It supplies a wide range of relational operations and dynamically decides where to execute the operations depending on how data is distributed across locations. LAD leverages Pandas to execute operations on data that is in the local execution environment while using the DBS engine to optimize and execute the query if the data resides in a remote DBS. We have tested LAD on the TPC-H benchmark and compared it with a pure Pandas approach. Our experiments show LAD outperforms Pandas for most of the scenarios being up to 3.4 times faster.

Abrégé

Pandas est l'un des frameworks les plus populaires pour le traitement et l'analyse de données. Pour travailler avec Pandas, les scientifiques des données chargent généralement des données qui résident initialement à différents endroits dans l'environnement d'exécution local. Si les données se trouvent dans une base de données relationnelle, certaines opérations relationnelles peuvent être manuellement transférées à la base de données pour récupérer moins de tuples de la base de données. Cependant, cela ne peut se produire que lorsque les scientifiques des données savent exactement ce qu'ils veulent accomplir avec les données, ce qui n'est généralement pas le cas pendant la phase d'exploration des données. Dans notre mémoire, nous présentons un framework analytique de données côté client, LAD, qui peut gérer les calculs sur des données distribuées de manière plus dynamique et transparente. Nous utilisons la même interface de programmation que Pandas, offrant une abstraction de dataframe pour la transformation et l'analyse. Il fournit un large éventail d'opérations relationnelles et décide dynamiquement où exécuter les opérations en fonction de la répartition des données entre les différents emplacements. LAD tire parti de Pandas pour exécuter des opérations sur les données qui se trouvent dans l'environnement d'exécution local tout en utilisant le moteur de la base de données pour optimiser et exécuter la requête si les données résident dans une base de données distante. Nous avons testé LAD sur la base de référence TPC-H et l'avons comparé à une approche purement basée sur Pandas. Nos expériences montrent que LAD surpasse Pandas pour la plupart des scénarios, avec une vitesse pouvant aller jusqu'à 3,4 fois plus rapide.

Acknowledgements

I would like to thank my supervisor, Professor Bettina Kemme, for her invaluable guidance, encouragement, and patience throughout my entire study and thesis writing. Her insights and help are essential for the completion of my thesis.

I would also like to acknowledge Doctor Joseph D'silva for his technical support and expertise in database systems. His suggestions are instrumental in the implementation of LAD.

Last, I want to thank Dailun Li for his contribution to this thesis. The query generator written by him was an essential tool for the experiments in this research.

Table of Contents

List of Tables

List of Figures

List of Programs

List of Algorithms

List of Equations

Acronym

AIDA Abstraction for Advanced In-Database Analytics.

API Application Programming Interface.

CSV Comma-Separated Values.

DDL Data definition language.

DML Data manipulation language.

LAD Locality-Aware Dataframe.

ML Machine Learning.

OLTP Online Transaction Processing.

OLAP Online Analytical Processing.

RDD Resilient Distributed Dataset.

RMI Remote Method Invocation.

SF Scale Factor.

SQL Structured Query Language.

SQL/MED SQL Management of External Data.

UDF User-Defined Function.

1 Introduction

1.1 Problem Statement

In the era of big data, data analysis is a crucial part of decision-making processes in various domains, and there are many analytical data frameworks available for data scientists to perform such tasks. Pandas [8] is one of the most popular frameworks. It is a Python-based library that is often used for the exploratory phase of data science. Usually, data in this phase is required to be cleaned and transformed for feature engineering or other downstream tasks. Pandas offers a table-like *dataframe* abstraction and convenient data processing functionalities, including linear algebra and relational transformations. That is, data scientists can use a high-level language and a dataframe API for their data processing. However, Pandas mainly works with local data since it is a client-side library. More precisely, data is typically loaded into the local Pandas execution environment that runs on the local machine of data scientists or on a cloud platform. Therefore, throughout the thesis

and for simplicity, we refer to the machine that runs the data science library (Pandas) as local machine or the client side, and the data that is loaded into this local execution environment as local data. Usually, the data to be loaded resides in local files on the client node but can also reside in distributed file systems, NoSQL databases or relational database systems (DBS). According to [24], more than 60% of the scientists work with the latter. In this case, data scientists might push down some of the relational operations to the DBS using SQL statements to only load a subset of the full dataset to the client side. However, this requires database expertise. A more common approach for data scientists to analyze database data is to move all the data to the client side and then process it.

There exist frameworks such as AIDA [9] and Grizzly [12] that bring the computation to the database by automatically generating SQL queries from high-level language code and executing everything within the database. However, they do not provide a solution to deal with data that is distributed across multiple locations, including local files and data in a remote database. Their current method is sending all data to one location, and executing everything within the database or executing everything on the client side. However, this can be both time-consuming and inefficient.

1.2 Thesis Methodology and Contribution

In this thesis, we introduce LAD (Locality Aware Dataframe), a data analysis tool that runs, just like Pandas, on the client side, but also allows users to work in a unified manner with data that is distributed across the client location and a remote DBS. LAD's interface follows the syntax of the Pandas API to facilitate adoption. LAD dynamically decides where to execute the relational operations depending on data location and hides the detail of data transfer and SQL processing from the users.

LAD uses a dataframe abstraction that wraps Pandas dataframes and database connections. It provides the same API as Pandas, but unlike Pandas, which evaluates operations eagerly and performs only on local data, LAD leverages a lazy evaluation on remote data, where multiple operations can be assembled into a single SQL statement for more efficient execution. Furthermore, LAD optimizes the process such that one query can be partially executed by the local Pandas library and partially by the remote database engine, and the data transfer is only performed when needed.

This is done by integrating a scheduler module into LAD to determine where to execute and what data to transfer using a locality-aware heuristic-based algorithm.

We compare the performance of LAD to pure Pandas where all data is loaded into the client-side Pandas environment and all operations are executed locally, and a heuristic-based approach that manually pushes some relational operations to the remote DBS by using a set of 8 queries from the TPC-H benchmark with varying table configurations. To cover more use cases, we also use a large set of randomly generated queries using a query generation tool and compare the performance of LAD and the pure Pandas approach. Our results show that LAD can outperform the other two approaches in terms of query execution time for most queries and table configurations, often to a large extent.

In summary, LAD provides a tool to work with queries on distributed data residing in both the local file system and a remote database with considerably better performance than Pandas. We hope LAD can simplify the data exploratory phase and allow data scientists to focus on their analysis tasks rather than data transfer and SQL query writing.

1.3 Thesis Overview

The thesis is organized as follows:

- Chapter 1: Introduction - This chapter sets the background and the context for this thesis, introduces the research methodology, and outlines the structure of the thesis.
- Chapter 2: Background - This chapter reviews the relevant literature and provides a theoretical foundation for the implementation of LAD.
- Chapter 3: Related Work - This chapter examines previous research and studies in related fields by discussing three other frameworks, namely, AIDA, Grizzly, and Spark. It also looks at research on distributed query processing.
- Chapter 4: LAD overview - This chapter provides an overview of the proposed LAD system and showcases its uses case and APIs.
- Chapter 5: LAD Scheduler - This chapter discusses the design of the LAD scheduler in detail.

- Chapter 6: Experiments - This chapter presents the experimental setup and methodology used to evaluate the performance of the LAD system compared to Pandas.
- Chapter 7: Conclusion - This chapter summarizes the findings from the experiments and the contribution of the project. It also lists the current limitations of LAD and outlines possible future work.

2

Background

2.1 Overview Relational Database Systems

Relational DBS are the predominant systems to store and manage structured data, commonly used as the backend of online transaction processing (OLTP) and online analytical processing (OLAP). We will illustrate the main characteristics of relational data and relational DBS using the well-known TPC-H database benchmark [4]. The benchmark contains a set of business-oriented tables and queries. It models an international online store where customers can place orders, and each order can contain one or more products registered as lineitems in the order.

2.1.1 Relational Model

Relational DBS requires the data to be specified in a relational schema. That is, all data is represented as relations (tables), where each table has a set of tuples (rows or records) consisting of a

set of attributes (columns). For instance, the TPC-H benchmark database consists of 8 tables, and their schema is shown in Figure 2.1. Taking the *customer* table as an example, each of its rows represents a customer with their unique identifier (primary key), name, address, etc. Each column has a name (e.g., *c_custkey, c_name*, etc.) and a type (e.g., *INTEGER, VARCHAR*, etc.).

Different tables can be connected with each other via primary key / foreign key relationships, where one (or a set of) attribute(s) of one table refers to the primary key of a different table. For instance, the table *orders* has an attribute *o_custkey* as a foreign key such that for a given row in *orders* the value of *o_custkey* is the primary key of the customer in the *customers* table that submitted the order. This allows for relating information from different tables.

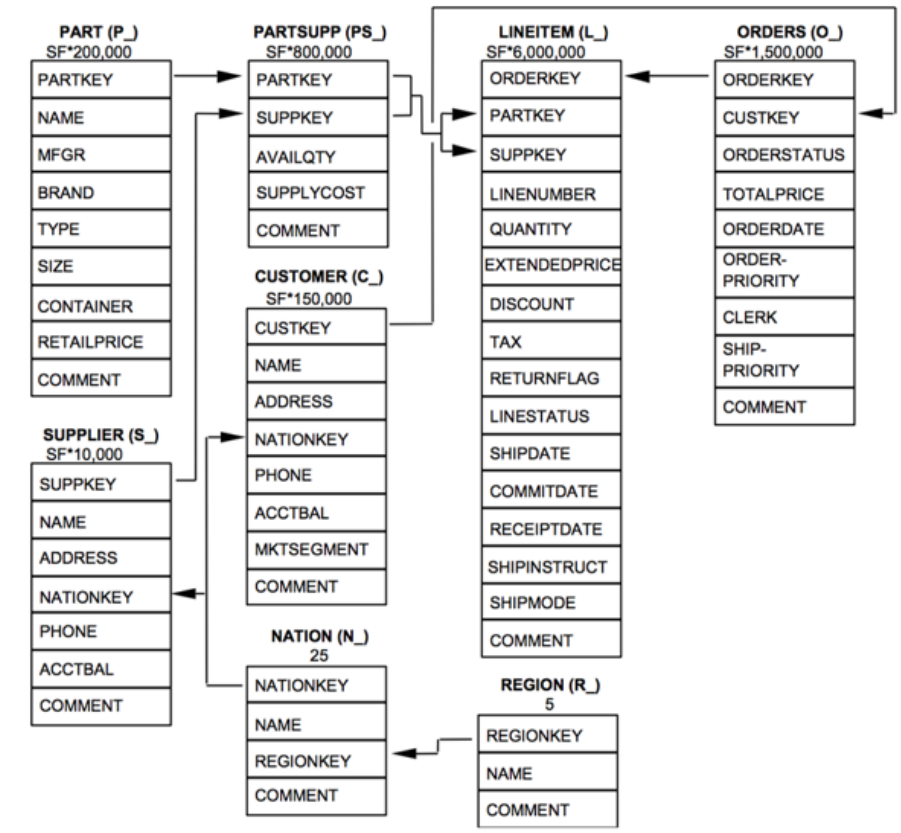
2.1.2 Structured Query Language

Structured query language (SQL) is a declarative query language with which schemas can be defined using data definition language (DDL) commands such as CREATE TABLE and CREATE INDEX, and data can be manipulated and queried by using data manipulation language (DML) commands such as SELECT/INSERT/DELETE. In the context of this thesis, we are interested only in queries that retrieve data from the relations, namely the SELECT command.

Listing 2.1 shows a query that includes all the query components we are interested in in the context of this thesis, and its result is shown in Listing 2.2. We would like to note that SQL provides a richer set of components, but those are merely touched on or not the focus of the thesis. The query retrieves the identifier of customers and the average amount of money they saved from discounts over all orders they placed after 1995, March 15. The result is sorted in descending savings order. Such a query is helpful for finding the customers who are more prone to buy discounted goods. The query involves operations like projection, selection, join, grouping, aggregation, ordering, and a nested query. We will go through the operators/clauses used in the query one by one.

The SELECT clause (in algebraic terms referred to as *projection* and also notated as π) determines which columns and possibly aggregations will be returned in the result. It is also possible to apply simple computations or functions on the columns.

The FROM clause indicates the tables which are queried. A table listed in the FROM clause can also be another SQL query. Note how we obtain the *orderkey* and *savings* from the inner query inside the parenthesis. We give the result table an alias name *os* and continue the query on



Legends:

- The parenthesis following each table name contain the prefix of the column names for that table;
- The arrows indicate a foreign key constraint with the head pointing to the primary key of the reference table;
- The number/formula below each table name represents the cardinality (number of tuples) of the table. some tables can be scaled up using a scale factor (SF) to obtain larger tables with an increased number of tuples.

Figure 2.1: The relational schema of the TPC-H benchmark (figure taken from TPC-H standard specification [4])

```
1
2  SELECT c_custkey, AVG(savings) as savings
3  FROM(
4      SELECT o_custkey, SUM(l_extendedprice) - MIN(o_totalprice) as savings
5      FROM orders INNER JOIN lineitem
6      ON orders.o_orderkey = lineitem.l_orderkey
7      WHERE o_orderdate > date '1995-3-15'
8      GROUP BY o_custkey
9   ) os
10 INNER JOIN customer
11 ON os.o_custkey = customer.c_custkey
12 GROUP BY c_custkey
13 ORDER BY savings DESC
```

Listing 2.1: SQL query example

```
1 Output:
2 c_custkey   savings
3 69538       4282618.710000000000
4 95257       4279153.950000000000
5 132817      4272776.870000000000
6 ...
7 (total rows: 99723)
```

Listing 2.2: SQL query example output (SQL)

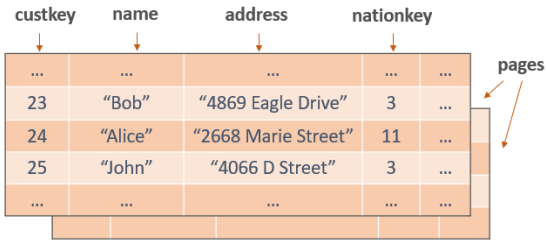
the *os* table. This is also known as nested query or subquery.

The WHERE clause (in algebraic terms referred to as *selection* and also notated as σ) defines conditions that must hold true for rows that should be returned.

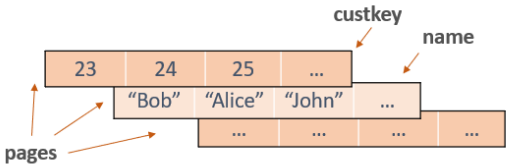
The query contains a JOIN operation that combines information from two tables where rows are linked via the primary key / foreign key relationship. The ON clause specifies on what condition the records should be joined together.

A GROUP BY groups rows that have the same value in a certain attribute and produces a single output row for each of the groups. The SUM and AVG are two examples of aggregation functions where the value of several rows (in the example the rows of a given group) are aggregated. The aggregation functions are often used together with the grouping.

The ORDER BY clause orders the column or columns specified in an ascending or descending order. The default is ascending.



(a) Record alignment in a row-based database



(b) Record alignment in a column-based database

Figure 2.2: Disk storage of row-based and column-based databases

2.1.3 Data Storage

Tables are typically stored on stable storage (i.e., disks) for persistence. They are split into pages and individual pages are loaded to the main memory whenever the corresponding data is queried or modified. How the table data is stored within those tables depends on the type of DBS. There are two categories, row-based and column-based DBS. In a row-based DBS all the attributes of a record are stored together, i.e, each page contains a horizontal partition of all the records. For a column-based DBS, all the values of a single attribute are stored together. Thus, each page stores a vertical partition of the data.

As seen in Figure 2.2, records of the customer table are stored adjacent to each other in the row-based format while all the values of a column are stored together for the column-based database. Note that extra meta information is also stored along the data to help the database track or navigate the data.

2.1.4 Data Flow

Over the past few decades, the database community has put a great effort into optimizing the data querying process, including the IO management, execution plan generation, and table cardinality estimation.

In this section, we go through some of these concepts as our project relies on the DBS engine, and certain aspects of our approach are enlightened by the DBS' algorithms and implementations.

To query data, the client needs to establish a connection with the database server (e.g via JDBC protocol). The DBS typically assigns a thread or a process per client dedicated to the requests submitted by the client. When the client submits a query, the following steps are executed.

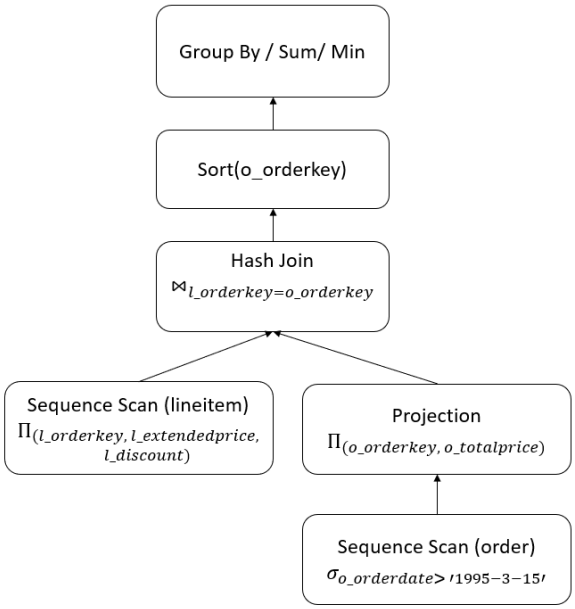


Figure 2.3: Sample query plan output by the optimizer for the query in the FROM clause of Listing 2.1

- **Query Parsing**

  The query parser validates and compiles the query. It resolves the names, references and data types in a query and converts the SQL query to a database internal representation.

- **Query Rewrite** The query is then rewritten into a logic execution tree with the tables as leaves and the operators such as selection, projection and join as inner nodes where the order of execution moves from leaves to root.

  Additionally, some logical operation rules and semantic optimization are applied to simplify or even evaluate the predicates and to reduce redundant information.

- **Query Optimization**

  The query optimizer converts the output from the query rewriter to a query plan that can be used by the low-level modules of the DBS to operate on the real data. It can reorder the operators in the tree to reduce intermediate data size. That is, it tries to push down the operators that will lead to the most reduced intermediate result sets so that they are executed as soon as possible. To do that, it has to estimate the output size of each operator and also the entire query. Furthermore, it decides for each operator how it is exactly executed. This

decision depends on how the data is stored and if indexes are available.

The optimizer generates many possible plans in its search space and searches for the optimal plan. Figure 2.3 shows a possible query plan output by the query optimizer for the inner query shown in Listing 2.1. This is also a strategy often used by many DBS: optimize nested queries separately.

- **Query Execution**

  The query execution incorporates routines to access the data, iterate over it and execute the operations on the fly. How the query is executed depends on how the data is stored. For instance, an iterator object is typically generated for each of the operators in a row-based database. These iterators form a pipeline, consuming the tuples from the previous iterator, and generating tuples for the next one.

2.2 Database Cardinality Estimation

In this section, we specifically focus on one critical aspect in the query optimization stage: the cardinality estimation, i.e., estimating the number of tuples that a query returns. To perform such estimation, the DBS maintains meta and schema information about each column of a table (e.g., data types of each column), and statistics about the rows that make up each table.

We will use the following meta information that is maintained in the database system catalog to make estimations:

- $|R|$: the number of rows of a relation (table) R
- $V(A, R)$: the number of distinct values for an attribute (column) A of relation R,
- $min(A, R)/max(A, R)$: the minimum/maximum values for attribute A of relation R
- $null(A, R)$ the fraction of NULL values for attribute A of a relation R
- $MCV(A, R)$ stands for most common values; it is the set of values of attribute A that appear most commonly throughout all the records in relation R.

- $MCF(A, R)$ are the most common frequencies corresponding to the values in MCV. Here we define the frequency of a given value as the fraction of rows that have this value (number of rows with this value divided by the number of rows of the entire relation R).
- a histogram that stores the distribution of the data

Most DBS maintain such metrics for each column.

Keeping tack of the number of rows and distinct values and the fraction of null values is fairly straightforward. We use a simple example to elaborate the rest of the definitions. Consider column A of relation R with the following current values:

$$(22, 22, 31, 50, 69, 69, 69, 80, 95, 100)$$

Then we have $MCV(A, R) = (69, 22)$ and the corresponding $MCF(A, R) = (0.3, 0.2)$ as 3 of the 10 values are '69' and 2 of the 10 values are '22'.

A histogram of a column divides the values of a column into buckets of a certain range and stores the frequency of appearance of values on a per-bucket basis. There are two types of histograms, equi-width histograms and equi-depth histograms [22]. While the former creates buckets with each having the same attribute value range, the latter creates buckets with different value ranges, but the number of tuples that fall into that range is the same [23].

PostgreSQL's implementation uses the equi-depth histogram. For instance, an equi-depth histogram generated by PostgreSQL from the above example might look like this:

$$(22, 31, 69, 80, 100)$$

In this histogram, 22 and 100 are the lower and upper bounds respectively. There are 4 buckets and each bucket has 1/4 of the tuples falling into it. That means 1/4 of the records fall between 22 and 31, 1/4 of the records fall between 31 and 69, and so on. Note that a histogram might not be accurate in the real database as random sampling is used when computing the histogram.

Other than the notations listed above, we will also use the following notations throughout the thesis.

- $\sigma_{A=a}(R)$ to denote selection on attribute A with the value a of relation R

- $R \bowtie S$ to denote the join between the relation R and S
- $|\cdot|$ to denote the size of a set

2.2.1 Row Estimation of Selection σ

We define the selectivity as the fraction of rows selected by a condition. The estimated number of rows is thus

$$selectivity * |R|$$

When computing the selectivity of a selection operation, we consider two scenarios. If no histogram or MCV are available, the database assumes a uniform distribution of the data. Assuming the selected value is in the normal range (i.e. the value is between the upper and lower bound of all possible values), the selectivity for $\sigma_{A=a}(R)$ is

$$selectivity(\sigma_{A=a}(R)) = \frac{1}{V(A,R)} \tag{2.1}$$

The selectivity for $\sigma_{A \leq a}$ is

$$selectivity(\sigma_{A \leq a}(R)) = \frac{a - min(A,R) + 1}{max(A,R) - min(A,R) + 1} \tag{2.2}$$

Depending on the actual value, this can be quite inaccurate. So ideally, we compute the selectivity based on the recorded most common values or histograms.

For the equality operator $\sigma_{A=a}$, if a is in MCV, then selectivity is simply the corresponding frequency captured in MCF.

If a is not in MCV, we assume a uniform distribution of values that are not in MCV. Thus, selectivity is obtained by dividing the frequency of the values that are not in MCV by the number of distinct values that are not in MCV.

$$selectivity(\sigma_{A=a}(R)) = \frac{1 - sum(MCF(A,R))}{V(A,R) - |MCV(A,R)|} \tag{2.3}$$

where $|MCV(A,R)|$ is the size of the most common values set.

For range predicates like $\sigma_{A\leq a}(R)$, we use the histogram to establish selectivity. For value a that falls in the i-th bucket, we count all the buckets before it and a part of the current bucket [11]. Thus, we have

$$selectivity(\sigma_{A\leq a}(R)) = (i - 1 + \frac{a - min(bucket[i]) + 1}{max(bucket[i]) - min(bucket[i]) + 1})/|buckets| \tag{2.4}$$

2.2.2 Row Estimation of Join $\bowtie$

When estimating the join cardinality, we assume the join to be a equi-join, i.e, the values of the join columns of the two tables should have the same values.

If the join column is a foreign key of R referencing S, it is trivial that the result set is $|R|$ (each record in R points to exactly one record in S). Similarly, if the join column is a foreign key of S referencing R, the result set would have the size of S.

If there is no primary key/foreign key relationship, for simplicity, we assume uniform distribution on the join column value. For a tuple r in R, assuming the join column is A, the number of tuples of S it can match is bounded by

$$|r \bowtie S| \leq \frac{|S|}{V(A,S)}$$

Similarly, for a tuple s in S, assuming the join column is B, the number of tuples of R it can match is

$$|r \bowtie S| \leq \frac{|R|}{V(R,B)}$$

Thus, for the entire table, and excluding the null portion, the estimated result set will be be

$$|R \bowtie S| = \frac{(1 - null(A,R))(1 - null(A,S))|R|*|S|}{min(V(A,S), V(R,B)} \tag{2.5}$$

2.2.3 Row Estimation of Other Operations

The projection and order operation do not affect the row number. Therefore, the row number of the output is the same as the row number of the input.

For operations like group by, the number of the output rows is exactly the same as the number of groups, which is essentially the number of distinct values of the groupby attribute. Then the aggregation operation outputs 1 tuple for each group.

2.3 User Defined Functions

User-Defined Function (UDF) is a feature of many modern DBS that allows users to define their own functions in SQL or other high-level programming languages. Similar to functions in programming languages, UDF in PostgreSQL can take in some inputs and output a single value or a set of tuples in the form of a database table. By using UDFs, complex operations on the data can be performed easily by calling the functions within SQL queries.

2.4 Data Science Frameworks

With the growth of big data and machine learning and the importance of complex data analysis, more real-world applications are not satisfied with the current OLAP capabilities of DBS, as standard SQL is not sufficient for the analysis. A richer set of operations is required to manipulate and transform data. Though DBS have introduced concepts like UDF to grant the user more freedom to manipulate the data within the DBS, many constraints and inconveniences still apply.

Therefore, complex data analysis is often performed outside the DBS. Data science frameworks take the data out of the database and perform the data analysis tasks on the client side (or on nodes dedicated to the framework). These data frameworks usually use a table-like data abstraction, often referred to as a dataframe, and API to manipulate and transform data.

Dataframes are implemented as lightweight libraries in many high-level languages such as R and Python. A dataframe is essentially an in-memory table, a structured data object [21].

In this section, we discuss how dataframes work along with Pandas, because we have implemented our solution using Pandas. We will cover other related frameworks (e.g. Spark) in the related work section.

2.4.1 Pandas Dataframe

Pandas is one of the most popular data analytical libraries for Python. It supplies a rich API to perform a wide range of analytical transformations on table-like data including linear algebra and relational operations.

Pandas' core functionalities are all built on its internal dataframes. A Pandas dataframe represents a table-like structure with a set of columns. Each column is an array of a basic data type. All columns have the same length. Pandas offers dataframes beyond the 2D table format but we ignore them in the context of this thesis. Internally, Pandas uses homogeneous 1-dimensional Numpy [19, 25] arrays for each column if the columns have different data types, but if they have the same data types, Pandas might group them into a 2-dimensional Numpy array [26]. Thus, various functions can be applied using the rich functionalities that are provided by the Numpy library [1].

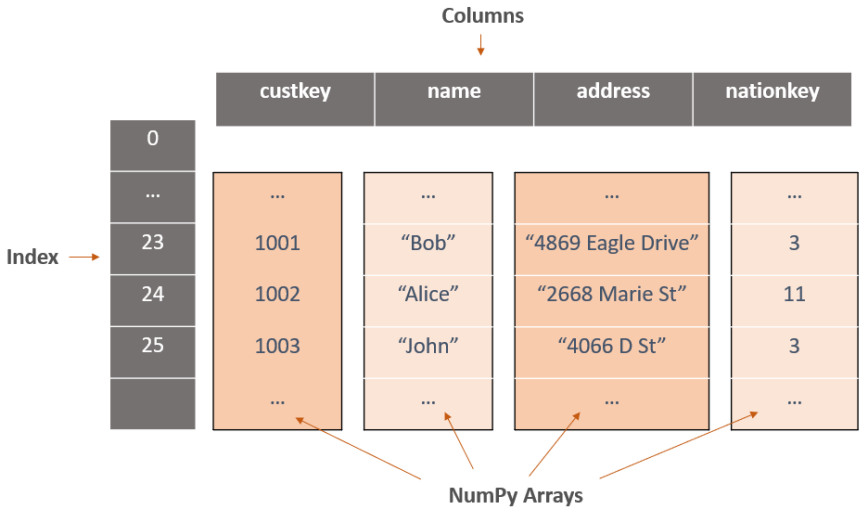


Figure 2.4: An example of Pandas dataframe

On top of Numpy's n-dimensional array representation, Pandas builds a named index for each

[1]Numpy is an open-source Python library that is optimized for operations on n-dimensional arrays and numerical computing. It is widely used among Python programmers and scientists.

axis to provide various functions. In the simple 2D world, these are the row indexes and the column names (the column names can be viewed as a special index), which allow the user to update or retrieve specific columns or rows given the index name. If we load our customer table example into Pandas' dataframe, the structure should look like Figure 2.4. We can see the internal storage is similar to a column-based table. In addition to the tuples' unique identifiers (the primary key), an index is added to indicate the row number.

To load data into a dataframe or persist a dataframe, Pandas provides a lot of IO input and output support, allowing the user to interact with a local file system or a remote database as shown in Listing 2.3. However, the process can be relatively slow depending on the size of datasets and the type of the data (e.g. data set that contains a lot of strings).

```
1 # load data from a local file
2 orders = pd.read_csv('orders.csv', ...)
3
4 # load the lineitem table from a database via database connection
5 lineitem = pd.read_sql_table('lineitem', conn)
```

Listing 2.3: Load data to Pandas dataframe

As a result of its architecture, Pandas can provide a rich set of operators: relational operators, linear algebra operators, and operators like manipulating data at a fixed index or converting the data to a matrix format (Numpy array). The latter two types of operators are usually not supported in a DBS. Listing 2.4 shows an example Pandas program listing a set of relational operators (e.g. projection, selection, join) and other operators.

The code from line 1 to line 21 in Listing 2.4 can be mapped to relational operations while the remaining operators are not available in a DBS. Line 5 projects on the *o_orderkey* from the *orders*; line 8 computes the price after discount and assigns it to the *price* column of *lineitem*; line 12 selects all orders that are placed on 1995, March 15; line 16 joins *orders* and *lineitem*; line 21 updates the values *l_discount* column in *lineitem* if the original value is greater than 0.9; line 24 updates the value at index $(2, 1)$; and line 27 and 28 convert the Pandas dataframe to a Numpy array and transpose the resulted matrix.

For most of the operations, the Pandas dataframe object is immutable. That is, upon calling a function of a dataframe object, the object itself is not changed but a new dataframe object with the required change is created. Operations like retrieving certain columns and certain rows all fall

```
# lineitem and orders are pandas Dataframe

# projection
# acquire the o_orderkey column
o = orders['o_orderkey']

# simple operations on columns
l['price'] = lineitem['l_extendedprice'] * ( 1 - lineitem['l_discount'])

# selection
# acquire a specific row
o = orders[orders['o_orderdate']==Date(1995, 3, 15)]

# join
# merge two tables
t = lineitem.merge(orders, left_on='l_orderkey', right_on='o_orderkey',
    how='inner')

# update
# set data to a value when the condition is met
lineitem.where(lineitem['l_discount']>0.9, 0.9)

# modify data of a single cell at a specified index
lineitem.iloc[2, 1] = '1.23'

# convert to Numpy array to do complex linear algebraic operations
l = lineitem['l_extendedprice', 'l_discount'].to_numpy()
l.transpose()

```

Listing 2.4: Query in Pandas

in this category. These operations do not change the underlying Numpy arrays, they merely make a shallow copy of them. Operations like sorting or applying a lambda function on the data, unless specified otherwise, also create a new dataframe object and do not affect the original ones. Pandas does provide in-place point modification. However, this is rather irrelevant to our project.

2.4.2 Eager and Lazy Evaluation

Pandas uses an eager evaluation for dataframe transformations. That is, the right-hand side of an expression is immediately evaluated and the result is assigned to the left-hand side dataframe, just like how simple expressions are usually evaluated in any programming language. For most object-oriented languages, the objects and expressions are built step by step, therefore the eager evaluation is a reasonable choice. It is straightforward and users can see the results immediately. However, the optimizations that can be applied to accelerate the processing are very limited.

In contrast, if we choose to delay the evaluation of the right-hand side and compute the result

only when the left-hand side reference is actually needed, e.g. for visualization, then this is called lazy evaluation. The left-hand variable only serves as a placeholder when it is initiated. With this, we might be able to avoid some costly computations. This idea is incorporated in the Resilient Distributed Dataset (RDD) concept of Spark and AIDA's TabularData abstraction. They build a lineage tree of the operations and optimize it before the execution.

3

Related Work

There are several projects that aim to save programmers the nuances of writing repetitive SQL queries by providing a unified system to work with a DBS and perform analytical tasks. We will describe three frameworks in this chapter, AIDA, Grizzly and Spark. The first two frameworks provide in-database analytics while Spark is a cluster-based analytics tool that also offers relational and algebra operations. In the end, we will also briefly touch on the concept of distributed query processing as it is conceptually similar to LAD.

3.1 AIDA

AIDA [9], which stands for Advanced In-database Analytics, is a tool embedded in the DBS that provides both relational operations and algebra operations to users.

AIDA uses an abstraction called TabularData to maintain data. The TabularData objects are

similar to Pandas dataframes, but with a Django-like API. On the client side, users call transformations on TabularData objects to perform analytical tasks. As the data resides in a remote DBS, the actual computations are shifted transparently to the DBS using Remote Method Invocation (RMI) so that the computations are performed near-data.

Internally, AIDA leverages the DBS engine to perform relational operations, and the result sets are stored in Numpy array form in TabularData objects. Therefore, all the linear algebra operations can be executed naturally by the Numpy library. AIDA also maintains a lineage tree that tracks all relational transformations performed on a TabularData so that relational queries can be evaluated lazily. For relational operations, AIDA pushes the SQL query generated from the lineage to the underlying database query engine only when the client explicitly requests the result. For linear algebra operations, the executions are directly performed on the TabularData objects in an eager fashion though.

One of AIDA's drawback is that it adapts Django-like APIs. This is quite cumbersome and not so intuitive for data analysis compared to other more popular Python frameworks like Pandas. Also, its RMI approach requires the modification and support of the server side and all executions are executed within the DBS, which might not be viable for many cases. Finally, AIDA is not designed for handling data that does not reside in the database. There is the option to ship the data to the DBS, however, this has to be done in a manual fashion.

3.2 Grizzly

Grizzly [12] is a client-side dataframe library aiming to push large-scale data processing and analysis tasks to the database engine. It uses Pandas-like APIs and serves as SQL transpiler which translates the Pandas-like APIs on the client side to SQL statements for the database to execute.

They define their own dataframe class, and for each operation, they build a subclass of dataframe. Also leveraging lazy evaluation, they have SQL templates and a dedicated SQL generator that generates SQL statements and sends them to the database for execution whenever the data is needed.

Grizzly also exploits database UDFs to port client-side Python code to the database such that machine learning algorithms can be performed in the database with user-defined PyTorch functions.

In case of performing transformations on data that does not reside in the database, Grizzly ships the data to the database as all executions happen in the DBS. This might not always be the best solution as the transfer time can be large and local execution might be faster.

3.3 Spark

Apache Spark is a popular open-source unified large data framework for distributed computing. It is able to access data from the Hadoop distributed file system and transform the data and do calculations using dedicated cluster machines. It is designed for distributed computing, e.g., Map-Reduce functions.

Spark evolves from the Resilient Distributed Dataset (RDD) [29]. An RDD is an immutable distributed collection of data that can be processed in parallel. Users can perform Map-Reduce tasks on RDD, such as *map* or $filter$. In Spark, these are called transformations. Each transformation creates a new RDD from an existing one, therefore a lineage tree of transformations is built and it is also lazily evaluated.

On top of the functionalities of RDD, Spark allows the user to use a dataframe abstraction to load data and perform operations [2]. The underlying executions are independent of the high-level languages used by users. For Python, Spark provides APIs similar to Pandas' for some relational operations, such as filter and aggregation, while offering other APIs used in distributed computing tasks [10]. When loading the data from a DBS, Spark can also push down some operations to be executed within the DBS. Nevertheless, once a table is taken out from the DBS, all the remaining operations will always be executed within Spark space on a cluster of machines even when this table can potentially interact with other tables in the database via join operations or sub-queries later on. Spark scales well for it splits the task for parallelization and utilizes the computing power of multiple machines. However, this can be an overkill and unsuitable for exploratory works as shown by the experimental results of [9].

3.4 Distributed Query Processing

For large applications, the separation of functionalities is essential for easier development and maintenance. This modularization of an application is usually accompanied by the requirement of mod-

ularization of data storage. For example, a web service may use a relational DBS to store its users' account information while using a multimedia database to store the content published. To allow query execution across the differed DBS, the concept of heterogenous or federated DBS has emerged, that supports distributed query processing. In the 80s and 90s, many projects and tools [6, 18, 28] were developed to construct a heterogeneous database system. Most of them include a hierarchical architecture where at the top level, a mediator maintains a catalog to store a global schema across the data sources and accomplishes actions from query parsing to query optimization while the database adaptors and actual DBS are at the second and third level, respectively.

In the mediator, one approach to generate a query execution plan is to build it in a bottom-to-top iterative way. For every table involved, all the possible ways to access the table are enumerated. On top of this, when encountering a join operation, for every combination of ways to access the tables, all possible ways to perform a join are also enumerated [14]. This can be implemented using dynamic programming.

Apart from heterogenous DBS, homogenous distributed query processing has also been analyzed [14]. Here, data is distributed across a set of identical relational DBS that collaborate in the execution of distributed queries. Again, distributed query plans are generated to decide which operation to be executed where. Some of the strategies can now be found in the Spark distributed query engine.

However, all this work on distributed query processing was developed before modern analytics tools were developed. In this thesis, we also develop distributed query execution strategies and our solutions are inspired by this previous work, but our focus is on embedding distributed query processing within an analytical framework that uses a dataframe abstraction and API-based transformation and not traditional tables and SQL, and where relational operations are only part of the functionality.

4

An Overview of LAD

4.1 Motivation

As described so far, using existing platforms, data scientists often need to move all data to their local machines or to a remote machine cluster to analyze. Even with the frameworks that push computation to the DBS, when local data is also involved, data scientists always need to manually move some of the data. This incurs programming overhead and can be cumbersome, and can lead to bad performance.

To show the inconvenience of Pandas when the user wants to handle data residing at multiple locations, consider a scenario of the TPC-H data where the *orders* table resides in the database and the *customer* table is stored in a csv file on the local file system. Assume the user wants to view the orders after March 5, 1995 for all customers.

Most data scientists will likely load the entire *orders* table from the database and continue the

remaining work with Pandas as shown in Listing 4.1, which incurs significant data transfer and data conversion overhead (from the row-based table used by most relational DBS to a column-based dataframe used by Pandas). Not to mention Pandas also handles everything eagerly, so the execution is likely not optimized.

```
o = pd.read_sql_table('orders', conn)
c = pd.read_csv("customer.csv")

o['o_orderdate'] = pd.to_datetime(df['o_orderdate'])
o = o[o['o_orderdate'] < np.datetime64('1995-03-15')]

t = c.merge(o, left_on='c_custkey', right_on='o_custkey', how='inner')
```

Listing 4.1: Execute everything in Pandas

If the user is familiar with the underlying datasets and SQL, they can write something similar to Listing 4.2 where only a relevant subset of the *orders* table is retrieved. However, this entrusts users to know SQL and thus manually split the operations across the data sources. For more complex queries, this process can be tedious and requires a lot of manual work and significant programming experience. Furthermore, it is not easy to interact with other database tables once the result is retrieved from the DBS to the client-side.

```
sql =  """
        SELECT o_orderdate, o_shippriority, o_orderkey, o_custkey
        FROM orders WHERE o_orderdate< date '1995-3-15
       """

o = pd.read_sql_query(sql, conn,
        parse_dates={'o_orderdate': {'format': '%Y-%m-%d'}})
c = pd.read_csv("customer.csv")

t = c.merge(o, left_on='c_custkey', right_on='o_custkey', how='inner')
```

Listing 4.2: Loading data with SQL in Pandas

We hope that this example shows that using Pandas on large database tables or data from different locations, which can be common during the data exploratory phase, is not well supported by Pandas.

On the other hand, there exist projects like AIDA and Grizzly that use a similar API as Pandas or Django, translate the relational operations under the hood to SQL statements, and can send all executions to the DBS. They both adapt lazy evaluation to leverage the query optimization

mechanics of the underlying database engine. However, AIDA requires adaptation of the database server as it is integrated into the server, and Grizzly only supports shipping operations in one direction, i.e., it always moves the data to the database for execution. That means join operations across tables that are distributed between the client-side and the database will send data to the database server for computation but not the other way around.

As such, our goal is to provide a data analytics tool that has the same dataframe-based interface as Pandas and allows for transparent query execution across distributed data sources, with the aim to offer a convenient tool for the data exploratory phase on distributed data. We name it LAD for Locality-Aware Dataframe. We choose Pandas because it is one of the most commonly used analytics tools for Python that offers a table-based dataframe abstraction, and the APIs are easy to work with for data scientists. As its name suggests, LAD is able to execute on heterogeneous data that resides in different data sources with a unified API. It aims to automatically and transparently execute operators close to where the data resides and when an operation requires input data from different data sources, it decides on where to execute based on a heuristic that aims to minimize data transfer and data conversion. In particular, if data resides on the client side or the file system, LAD processes it using the Pandas dataframe. In contrast, data that resides in a DBS can be processed by the database engine, but might also be loaded to the client side for processing. The decision process and data transfer are hidden from the user. We do so by including a scheduler module inside LAD which determines the best location to execute data operations based on heuristics and by moving data automatically between the client side and the database.

4.2 LAD Architecture and Data Abstraction

Conceptually, the LAD architecture consists of three modules: LAD Dataframe, a scheduler, and data source. We illustrate each module and the connections between them in Figure 4.1

4.2.1 LAD Dataframe

Conceptually, the LAD dataframe exposes the same API as Pandas. Similar to Pandas, for each operation, a new LAD dataframe is created.

Though sharing similar behavior from the perspective of the user, local data and remote data

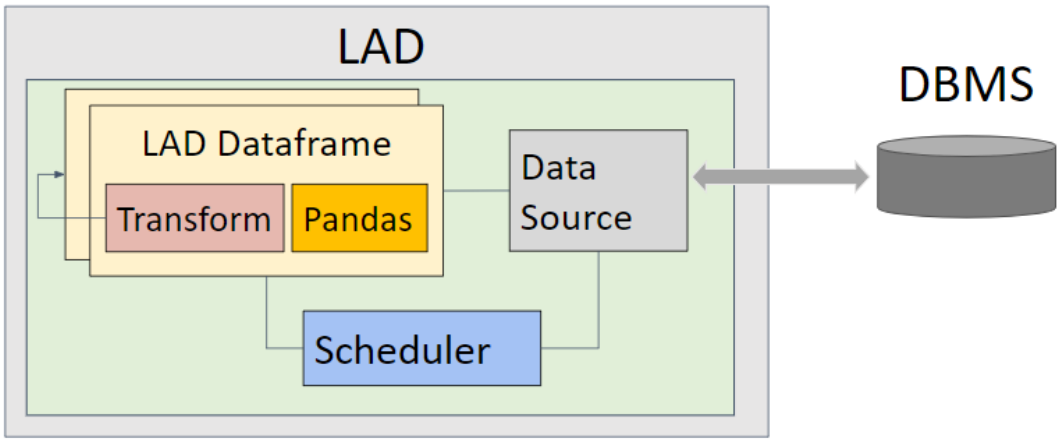


Figure 4.1: An overview of LAD's architecture

are treated differently by LAD.

Unlike Pandas, which eagerly manipulates the actual data, if the dataset is remote, LAD does not immediately execute the transforms on the LAD dataframe. Instead, it creates a new LAD dataframe that only represents a shell without any actual data and a transform object that keeps track of the operation and the source dataframe on which the operation is to be performed. Eventually, we create a lineage tree of LAD dataframes, from which a SQL query can be generated and sent to the database for remote executions.

On the other hand, we leverage the Pandas dataframe to store data once it resides on the client-side and simply invoke Pandas' functions to process data if all data involved is local.

4.2.2 Data Source

The data source is an abstraction for a database system connection. The details of the data source implementation are shown in Figure 4.2. The data source uses a database adaptor that implements all the functions that are required to communicate with a database. The user only needs to create a data source object by providing the connection information of a database, such as the hostname, username, etc. One data source object thus can be viewed as one database connection. Via the data source, the dataframe can retrieve the required information from the particular database.

There is also a data source manager holding all registered data sources. It provides the scheduler with the specified data source given a unique identifier.

For different kinds of DBS, such as Postgresql, MySQL or MonetDB, different database adaptors need to be implemented. LAD currently provides a PostgreSQL adaptor, which establishes a connection to the database using the Psycopg library[5] and holds it to execute queries later.

Additionally, the scheduler and the dataframe need to be able to retrieve meta-information about tables or potential queries from the DBS, such as cardinality information, histogram, etc. For that purpose, we define a set of utility functions implemented as database-specific UDFs and store them in a configuration file in yaml format. When a data source is initially created, these functions are loaded into the DBS and stored as temporary functions. These functions then become ready to be used by the dataframes and the scheduler to retrieve all necessary meta-data.

Therefore LAD can be easily extended to use other DBS as long as the necessary functions are defined and an adaptor for that particular DBS is provided.

For local data, since we leverage Pandas dataframe to execute operations on the data, we do not need SQL templates or functions for local data. Thus, we simply create a dummy local data source that wraps the Pandas functions to match the interface of our database adaptor to achieve a unified behavior.

Each LAD dataframe holds a reference to a data source. However, whenever a dataframe is the result of a join operation of two dataframes that have different data sources, the data source of the result data frame is left empty as the scheduler will decide the execution location and thus, the data source of the result dataframe.

4.2.3 Scheduler

Whenever a client requests the actual data of a dataframe that has a lineage tree containing a remote DBS table, we have to execute all the operators in the lineage tree. We call this materilization. Upon materialization of a dataframe object, the scheduler is invoked to generate a plan that decides where to execute the operations. The scheduler has access to the dataframe's entire lineage tree, the data source manager and thus, all data sources.

The scheduler will firstly traverse the entire lineage tree to collect necessary information, including the data location, column information, table statistics and the cost estimation of the database engine if necessary. For local data, it relies on Pandas statistics and random sampling. For DBS tables, it uses the UDF of the data source to obtain such information.

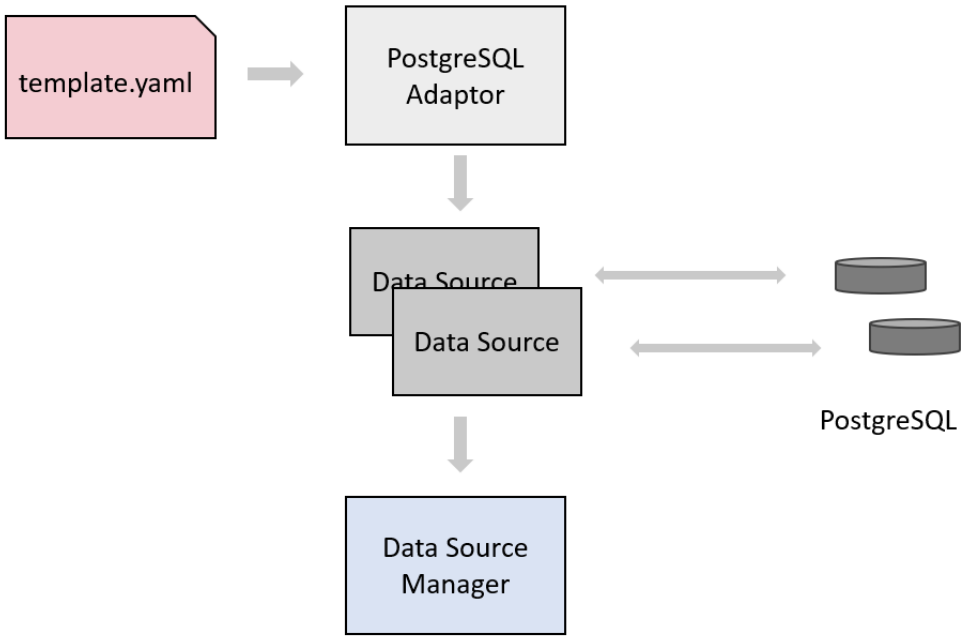


Figure 4.2: Data source module in detail

Essentially, it separates a complex query that involves different data sources into smaller chunks based on cost estimation, decides the data transfer and execution location and requests each system to execute a partial query in order to avoid unnecessary data transfer between the remote database and the local machine. We will cover this in detail in Chapter 5.

4.3 LAD APIs

In this section, we provide an overview of the LAD API and how it can be used leveraging the familiar Pandas APIs while hiding the rewrite to SQL queries, and the data transfer and data type conversion under the hood.

4.3.1 Connecting to a Data Source

To use remote data, the user needs to register the data source by providing the database authentication information to LAD. An example is shown below.

```
lad.add_data_source(db_type, host, user, password, dbname, port, ds_name)
```

The argument *db_type* is a string that specifies the kind of database. Currently, LAD support "postgres" for PostgreSQL.

The argument *ds_name* is the identifier given by the user for each data source. This *ds_name* will be used later to retrieve tables from the data source. LAD will treat a data source associated with two different names as two distinct databases even if they use the exact same database under the hood.

In the example below, the user views all the tables from a data source named *ds*1 and retrieves the *orders* table. Note at this point no data is sent to the client side as LAD uses lazy evaluation. The only information available to LAD right now is the meta information of the orders table such as column names and types. We will cover this lazy evaluation in Section 4.4.

```
lad.list_tables('ds1')
df = lad.read_remote_data('ds1', 'orders')
```

If the user wants to load local datasets, they can call the standard Pandas functions such as *read_csv*.

```
c = lad.read_csv('path to the dataset')
```

4.3.2 Pandas-like API

Once the data source and the tables are registered with LAD, the user can proceed with the remaining operations using only the Pandas API. Take the same customer's orders we mentioned in Section 4.1 as an example, the user can write everything using standard Python and Pandas syntax without the need of any SQL knowledge as shown in Listing 4.3.

```
o = lad.read_remote_table('ds1', "orders")
c = lad.read_csv("customer.csv")

o = o[o['o_orderdate'] < np.datetime64('1995-03-15')]

t = c.merge(o, left_on='c_custkey', right_on='o_custkey', how='inner')
```

Listing 4.3: Execute queries with LAD

```
1
2  # filter on conditions
3  c[(c['c_department'] == 'building')]
4  c.query('c_department == building')
5  c[c['c_department'].str.contains('^.building.*$', regex=True)]
6  c[c['c_department'].isin(['building'])]
7
8  # projection
9  c['c_department']
10 c[['c_department', 'c_custkey']]
11
12 # assign values
13 c['acctabl1'] = c['c_acctbal'] * 100
14 c['acctabl2'] = c['c_acctbal'] + c['c_acctabl']
15
16 # merge with another table
17 c.merge(o, left_on='c_custkey', right_on='o_custkey')
18
19 # aggregation
20 c.groupby('c_department').count()
21 c.groupby('c_department').agg({'c_acctabl': "max"})
22
23 # sort
24 c.sort_values(['c_custkey'], ascending=[False])
25
26 # materialize
27 c.materialize()
28 # the print function calls materialize function internally
29 print(c)
```

Listing 4.4: LAD supported API for relational operations

So far, we support a fairly rich subset of Pandas relational functions with LAD. We illustrate their usage in Listing 4.4, grouping them by functionalities.

Therefore, regardless of the table's actual location, the user can use the same relational functions as if the data were already local in a dataframe without worrying about data transfer or type conversion.

Most other operations are also supported but will always require the input dataframe to be a materialized local dataframe so that we can reuse the Pandas dataframe implementations of those operators.

4.4 Lazy Evaluation

Though sharing the same API, local data and remote data are handled differently.

For local datasets, we eagerly execute each operation with Pandas. However, with relational

operations that involve remote data, we perform lazy evaluation. This section describe how this lazy evaluation is done for relational operations.

As we mentioned before, when creating a new dataframe, we store all the arguments inside the dataframe, and create a transform object based on the type of functions the user calls. At this point, only the meta data, like column information and transform information, is available to the user. Only after the user explicitly calls the materialization function or any other functions that requires the data to be local, will LAD invoke the scheduler to make a plan and execute the queries accordingly. Then the result will be sent to the client side and we say this dataframe is materialized.

From then on, the dataframe's transform and the lineage are discarded. Hence, the remote data will become a local dataset.

4.5 Convert Transforms to SQL Statements

In this section, we briefly talk about how a LAD dataframe lineage tree is translated to a SQL query.

4.5.1 From Transform and Lineage to SQL

Due to Pandas' eager evaluation, it is not trivial to translate some of its syntax to SQL syntax. Thus, we group the Pandas functions with similar usage and define seven general types of transforms to translate Pandas syntax to 6 types of SQL operations: selection, projection, join, group, aggregation, and order. We show how each Pandas function gets mapped to LAD's transforms in Table 4.1.

LAD recursively generates the query text for each transform and its source/sources and incorporates it in the final result as subqueries.

Next, we give a concrete example using the *customer* table and *order* table. We place the *customer* table on the local machine that hosts the Pandas execution environment while placing the orders *table* within the database. The LAD code is shown in Listing 4.5.

For this program, LAD will construct the lineage tree shown in Figure 4.3. Since the customer dataframe is a local dataset, we perform eager evaluation. Thus, c2 is immediately materialized and holds no lineage of c1. For other dataframes that either directly link to a database table or involve mixed data sources, we maintain a transform object to store the transformation information. The

Pandas Function	LAD Transform	SQL Syntax
df[df[...]] df.where(...) df.query(...)	FilterTransform	WHERE ...
df[...] df[new_col] = ...	ProjectionTransform	SELECT ...
df1 + df2	BinaryOperationTransform	SELECT ...
df.merge(...)	JoinTransform	JOIN ...
df.groupby(...)	GroupByTransform	GROUP BY
df.agg(...) df.count() df.sum() df.max() df.min	AggregationTransform	COUNT/SUM/MIN...
df.sort_values()	SortTransform	ORDER BY ...

Table 4.1: Mapping between Pandas and SQL

JoinTransform is a special case, as illustrated in the figure, as it maintains two source dataframes. This is also where the scheduler module will kick in to make decisions.

Upon materialization of the t2 table, the scheduler will decide on a plan to split the tree and transfer some data either to the remote data source or to the local machine. Depending on how the tree is split, partial queries are generated by the transform. Then, the final result will be retrieved from the remote DBS to the local machine if necessary and stored locally in the t2 dataframe as a Pandas dataframe, but the intermediate dataframes will not be materialized.

4.5.2 Implementation Challenges

During the translation from Pandas syntax to SQL statement, we encountered several challenges. We list three difficulties in this section.

1. Database tables are not ordered

   Unlike a Pandas dataframe, the records in a database table do not have a fixed order. This

```
# register dataset to LAD
# orders is inside database
o1 = lad.read_remote_data('ds1', 'orders')

# customer is local dataset
c1 = lad.read_csv('/customer.csv')

# find customer with BUILDING mktsegment
c2 = c1[(c1['c_mktsegment'] == 'BUILDING')]

# find orders placed before 1995.3.15
o2 = o1[o1['o_orderdate'] < np.datetime64('1995-03-15')]

# join the tuples with the same custkey
t1 = c2.merge(o2, left_on='c_custkey', right_on='o_custkey')

# find number of orders placed for each customer since 1995.3.15
t2.groupby('o_orderdate').count()
```

Listing 4.5: LAD query example

means common operations in Pandas like retrieving or updating a value of a specific row using the index are not possible to do in the database unless the row index of the dataframe is also the primary key of the database table. Also, with Pandas, one can update easily any element of any column within a Pandas dataframe as Pandas does not impose any integrity constraints, such as the foreign key or primary key constraints. However, such constraints might exist in a database table, and if we translate such an update to an UPDATE/DELETE statement on the table to which a dataframe refers to, the update can fail. Therefore, we currently do not support single row access via the index or updating a LAD dataframe that refers to a remote database table. The data would first need to be materialized to a local dataframe.

2. Assigning a new column to a dataframe

   Pandas allows the addition of a column to an existing dataframe, that is, this operation changes the existing dataframe. However, this cannot be done directly with the architecture we described above. If we were to directly add a new column directly to a LAD dataframe that is part of a lineage to a database table, it could affect other dataframes that depend on it (come later in the lineage tree), which could result in unexpected behavior.

   To solve the problem, we add a thin wrapper around the LAD dataframe. We did not mention this when describing the dataframe of LAD in previous subsections because the wrapper does not change the behavior of the LAD dataframe. The only difference is that the object we

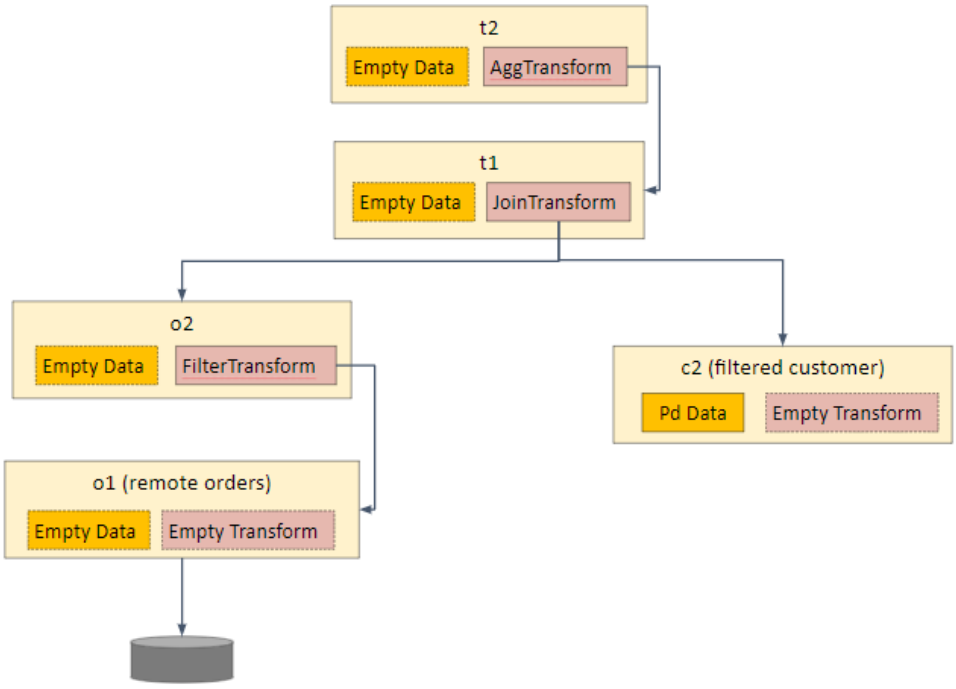


Figure 4.3: Lineage tree stored by LAD

return to the user is in fact the wrapper. The wrapper maintains a list of LAD dataframes. Upon assigning a new column, instead of changing the original dataframe, we create a new dataframe as usual and append it to the end of the list. When we perform further transforms, we perform the transforms on the tail dataframe of the list.

This way, even if we add a column to a dataframe, it does not affect the behavior of other dataframes that depend on the original dataframe.

3. Combine SQL statements for special cases

   For filter operations like

```
o = o[o['o_orderdate'] < np.datetime64('1995-03-15')]
```

   Pandas evaluates the expression in the square bracket first, which returns a boolean vector, like [True, True, False, False ...]. This indicates for each row of the data if the selection criteria are met. The vector is then used to filter the original data. When we use lazy evaluation, the extra

step of calculating the boolean vector is unnecessary. Thus, inside the projection transform, when it detects the previous operation is a logical operation or a comparison operation, (e.g. AND, OR, $\leq$, .etc), it will merge the projection and filter into one SQL statement.

Another special case is groupby and aggregation. In Pandas, a groupby operation creates a special groupby object and not a dataframe. However, in LAD, the groupby object is just treated like other dataframes. If an aggregation transform detects its previous dataframe has a groupby transform, then it will combine the SQL statements as well.

5

LAD Scheduler

LAD incorporates a scheduler module which accesses the dataframe's lineage tree and all data sources in order to determine the location of execution and push the data to that location if necessary. In this chapter, we describe in detail how the scheduler decides on an execution plan and automates the data transfer between different data sources.

5.1 Scheduler Building Blocks

Upon materialization of a dataframe, the dataframe invokes the *materialize* function of a global scheduler object. First, the scheduler needs to get all the previous operations that led to this dataframe up to the source dataframes to make an execution plan. Thus, the scheduler traverses the current dataframe's lineage tree up to the leaf source dataframes. During the traversal, the scheduler constructs an *execution block tree* to handle the planning, data transfer, and execution,

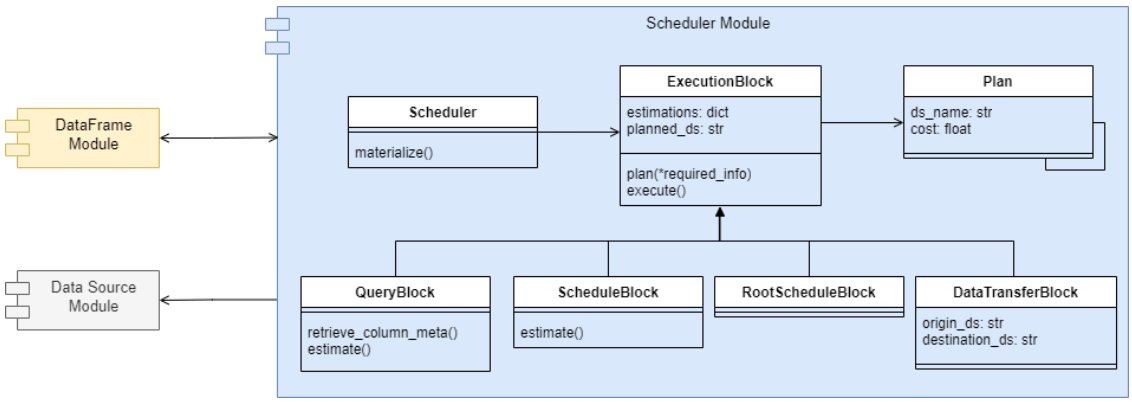


Figure 5.1: Class diagram for the scheduler module

and this tree is composed of a new abstract called execution block.

The execution blocks are created to simplify the problem and are the building blocks of the LAD scheduler. They split the lineage tree logically for heterogeneous executions. The generation of alternative execution plans, the selection of the best plan, and the actual query execution either with the DBS or with Pandas are performed by the execution blocks together. Especially for the selection of the best plan, which largely relies on the cost of executing a plan, the execution blocks start a recursive process to make an estimation of the execution cost. This is done by accessing the information through the dataframes and the data sources. This process is the core of the Scheduler. In Figure 5.1, we show the class definitions of the scheduler module and its interactions with the dataframe module and the data source module.

Depending on their primary task and usage at different query planning or execution phases, we divide the execution blocks into four categories. In the figure, these are the 4 classes that inherit the execution block class. We list the task of each of them below.

1. **Query block**:

   The query block is constructed by grouping a series of dataframes and is analogous to a branch of the lineage tree. All dataframes in the query block have the same data source, and thus can be executed by a single system (Pandas or DBS). For example, if we were to perform a selection and a projection on a database table, then everything can be performed within the database. The original database table, the selection transform and the projection transform naturally form a query block together. In other words, the query block is a wrapper of a part

of the lineage tree whose operations can be performed with one system without data transfer. The query block is then also responsible to execute all the operators that are part of the block.

The query block has access to the dataframe's datasource, and therefore can obtain the meta information of the dataframe and estimate the size of the dataframe. If the dataframe is remote, then the database engine needs to be queried for the statistics or estimation of the dataframe. If the dataframe is local, which means the data is available in Pandas, then the statistics stored in Pandas, and random sampling is used to give an estimation.

2. **Schedule block**:

   When building the execution block tree, a scheduling block is inserted wherever 2 branches merge in the lineage tree due to a join operation. The join transform always has two source dataframes and the data sources of these dataframes are different. This also means the schedule block has two query block children in the execution block tree.

   The schedule block is responsible for obtaining the meta information from the children query blocks and calculating the cardinality or statistics of the result set of the join operation using the heuristic methods we introduced in Chapter 2.2. As the children query blocks have different data sources, data transfer might be required to move data to one location to perform further operations. The cost for all possible data transfer options between two query blocks is then combined and computed recursively at the schedule block.

3. **Root Schedule Block**:

   The root schedule block, as indicated by its name, is always at the root of the execution block tree and oversees all cost information from all the blocks along the tree. By selecting the plan with the lowest data transfer cost, it decides at which data source the actual execution of a query block will take place and whether data transfer is required. Should data transfer be performed before executing some of the query blocks, it inserts a transfer block before the query block so that all data needed for its child query block is transferred to the right location for query execution.

4. **Transfer block**:

The transfer block is inserted in the execution block tree where data transfer is necessary. It contains information about the origin data source and the destination source, and it simply performs data transfer from one data source to another by uploading or downloading the table to or from the database. The download from the database to the local Pandas environment is straightforward. When uploading to a database, the transfer block first requests the data source to create a temporary table inside the database based on the column information of the dataframe that needs to be transferred. Then, it sends the data in batches while converting the non-compatible data types on the fly. We note that the implementation of the transfer block is also, to a certain degree, DBS dependent, as creating temporary tables is not standardized across DBS. Furthermore, there are several ways to transfer data to a DBS. As explained in [3], the temporary table option is an effective option.

5.2 Scheduling Workflow

We use the example from Chapter 4 to illustrate the entire process of planning and execution. We want to perform some operations on the *customer* and the *order* tables and these tables are at two locations. Assume we have already obtained the lineage tree shown in Figure 4.3. We have redrawn the lineage tree in Figure 5.2 (a) with the necessary information for the scheduler. Note that the original *customer* table is not part of the lineage tree as it is local in Pandas, and for operations performed with pure local data, we directly evaluate the result with Pandas. Thus, the right leaf node of the tree is the filtered customer table.

In the following subsections, we describe four phases the scheduler performs to build and execute a plan. It first creates a basic execution block tree based on the lineage tree. Then it performs cost calculation for all options of where to execute part of a plan. The plan with the lowest estimated cost is then chosen. Next, we build the final execution block tree that involves the transfer blocks. The final phase is then to execute this plan.

5.2.1 Building the Initial Execution Block Tree

When the user invokes the *materilize* function, the global scheduler object starts to construct an execution block tree. At the root of the lineage tree, we have performed the aggregation operation to

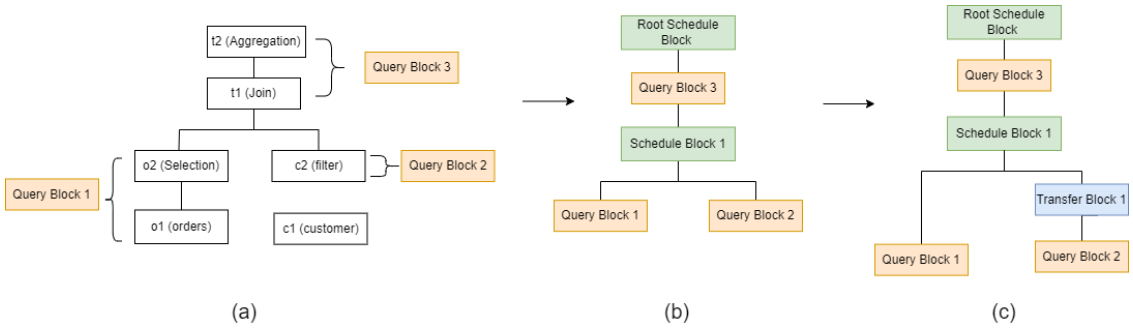


Figure 5.2: LAD scheduler constructs a tree composed of execution blocks from the lineage.

obtain the $t2$ dataframe, and when we trace back the sources of $t2$, we eventually have two branches. We traverse the lineage tree to reach the leaves of the lineage tree and build the execution block tree from bottom to top. First, we wrap $o1$ and $o2$ dataframes in query block 1 and the c2 dataframe in query block 2 as depicted in Figure 5.2 (a). Join is a special operation as it merges two dataframes together, and these two dataframes might be in two different locations. We insert a schedule block that accesses query block 1, query block 2, and the join dataframe to estimate the costs of all possible data transfer options. Here the data transfer option is essentially the decision to bring the data to the database from the local Pandas dataframe or vice-versa. Then, we wrap the remaining dataframes in the third query block.

After completing the traversal of the lineage tree and connecting all query and schedule blocks, we insert the root schedule block at the root. The basic execution block tree built at this phase is shown in 5.2 (b). The root schedule block then starts the remaining planning and execution process.

5.2.2 Generating Execution Plans and Performing Cost Calculations

Traditionally, when computing query execution cost in a distributed DBS, usually the CPU time, the disk IO, the number of messages, and the data transfer rate all need to be taken into consideration [20]. However, in our scenario, as shown by experiments in [3], the most dominant factors are the data transfer and conversion time for large tables between the row-based DBS and the column-based dataframe. Therefore, we simplify the problem by using the size of the result set of a query block for the cost estimation as it reflects the cost of data transfer and conversion. We calculate this by multiplying the estimated number of tuples by the size of all attributes that the result table has.

The planning is done recursively. In our example, the root schedule block calls the plan function

Algorithm 1: Plan function of the query block Q_i

```
1  if has no children then
2      if data source is database then
3          plan ← NIL
4          Generate SQL query
5          plan.QR ← Request the database to analyze and estimate the query result set size
6      else
7          plan.QR ← Get the result size from Pandas meta information
8      plan.ds_name ← data source name
9      plan.cost ← 0
10     plans ← {plan}
11 else
12     plans ← child.plan()
13     Qi.QR ← Calculate the size of the result set using heuristics methods
14     for plan in plans do
15         plan.QR ← QR
16 return plans
```

of query block 3, and query block 3 calls the plan function of the schedule block, and so on. During this process, if some statistics regarding some columns are needed, for example, the scheduler block 1 might need the number of distinct values of the join attributes, the request to retrieve the meta-information about the relevant attributes will then be propagated together along the planning function call. In other words, the request will be sent as the arguments when invoking the plan function of the children execution blocks. This is shown by the plan function of the ExecutionBlock class in the class diagram in Figure 5.1. While the plan functions are called recursively, the creation of the different plans and their cost calculations are done bottom-up. Algorithm 1 shows the actions for a query block, Algorithm 2 for a scheduler block that was created due to a join, and Algorithm 3 for the root scheduler block. For simplicity, the algorithms omit how blocks further up in the tree receive the required meta information that they need to make cardinality and result size estimation through the recursive call.

Recall that query blocks will perform operations on data that is local and we do not consider the CPU or IO cost of execution. Therefore, there is no actual execution cost for the execution within the query block, but at some time point, data might need to be transferred. If the result that is produced by a query block must be transferred then the transfer cost is calculated as the size of the query result. Therefore, each query block Q_i keeps track of the size of its query result

QR and appends it to the plan object (see lines 5, 7, 13 in Algorithm 1). If the query block detects its sets of dataframes are within the database, such as query block 1 in the example, then it gets the SQL query generated by the dataframes and requests the database to estimate the result set size of this query. For query block 1, this would be the cardinality after the selection. If the query block detects that its dataframe has a local data source, which is the case for query block 2, it must be a single dataframe because of the eager evaluation on local data. Then it automatically obtains all the information about the column size and row numbers. If meta-information such as MVC or the number of distinct values is not present, we use random sampling to get them if the data size is large. For simplicity, we do not compute the histogram but assume a uniform distribution instead. Then we store these estimations of the result tuples in this query block. Furthermore, when the query block is a leaf node, it creates a single plan object P with a cost $C_P = 0$ (the result set remains at the execution location) and an execution location referring to a data source, i.e., a DBS or local (see lines 9, 10 in Algorithm 1). This information is then transferred to the parent scheduler block.

An intermediate schedule block is always introduced when the two input tables of a join reside at different locations. The join can thus be executed at any of the two locations and the result set of one of the child query blocks needs to be transferred to the other location. If the children of a schedule block are both leaf query blocks, then each of them has created so far one plan with cost of 0. The schedule block now creates two plans, one for each possibility to perform the join. The cost of each of these plans is the cost of transferring the result set of one of the child query blocks to the other location (see lines 8, 9 in Algorithm 2). That is, in the example, schedule block 1 will create two plans, plan1 transferring the result of query block 1 to the location of query block 2 with the cost $C_{plan1} = plan1.QR$, and plan2 having correspondingly cost $C_{plan2} = plan2.QR$. Each plan also keeps track of the children plans that it extended and the location where the join will execute (lines 10-12). Furthermore, as it has already collected all the meta-information it also estimates the output size of the join, which becomes the query result size of the plan (lines 13). The result tuple size of the join operation is estimated using the formula described in Section 2.2.2. The parent query block (in our example query block 3) takes over these plans with their costs and locations and simply updates the query result size based on the operations that come after the join, which is in our example of query block 3 the aggregation (see lines 12-15 in Algorithm 1). The heuristics to

Algorithm 2: Plan function of intermediate schedule block

```
1  init
2      new_plans ← Empty list
3      child1, child2: two children query blocks
4      plans1 ← child1.plan()
5      plans2 ← child2.plan()
6  for plan1, plan2 in plans1, plans2 do
7      new_plan1 ← new_plan2 ← NIL
8      new_plan1.cost ← plan1.cost + plan2.cost + plan2.QR
9      new_plan2.cost ← plan1.cost + plan2.cost + plan1.QR
10     new_plan1.children ← new_plan2.children ← (plan1, plan2)
11     new_plan1.ds_name ← plan1.ds_name
12     new_plan2.ds_name ← plan2.ds_name
13     new_plan1.QR ← new_plan2.QR ← Calculate the size of the join result set using
         heuristics methods
14     new_plans ← new_plan ⋃ {new_plan1, new_plan2}
15 return new_plans
```

do so are covered in Section 2.2.1.

For other intermediate schedule blocks (not existent in our example), the children might thus have already more than one plan. If child 1 has p_1 plans and child 2 has p_2 plans, then the scheduler will create $2 * p_1 * p_2$ plans, i.e., for each combination of plans of the two children it will create 2 plans; one for each possible transfer direction. The cost of each of these plans is the sum of the costs of the underlying two plans plus the size of the result of the query block that will be transferred for the join. Again, the location will be set accordingly and join result size estimated.

The final root schedule block shown in Algorithm 3 has only one child query block. It will take over the plans of this child query block and perform a final calculation. If the plan indicates the DBS to be the location for the execution of the child query block, then a final transfer of the result to the client side is needed. Thus, it adds to the plan's execution cost the result set size of the child query block. If the location of the query block is local, the costs of the plan remain the same. From all the generated plans, the root schedule block then selects the plan with the lowest cost.

Overall, this planning process makes a complete search through all possible execution plans over all possible data source locations. The growth of the potential plans is exponential, but as the process only happens at limited operations (namely join), we believe the search time should be manageable.

Algorithm 3: Plan function of the root schedule block

```
1  init
2      child: child query block
3  plans ← child.plan()
4  finalplan ← NIL
5  finalplan.cost ← infinity
6  for plan in plans do
7      if plan.ds = remote DBS then
8          plan.cost ← plan.cost + plan.QR
9      if finalplan.cost > plan.cost then
10         finalplan ← plan
11         finalplan.cost ← plan.cost
12     return finalplan
```

At the end, each created plan object is essentially a tree that is built gradually and in the end, it has a structure corresponding to query blocks of the execution block tree and indicates the execution location for each query block. For our example, there will be at the end two alternative plans. Listing 5.1 below shows first the plan where the join is executed locally and then the plan where the join is executed in the DBS. In the example, the customer dataframe after the selection has 30 142 tuples, and the orders table after the selection is estimated to have 727 305 tuples. If the join is executed locally, the transfer costs for the 727 305 order tuples will be 727 305 × 99 (72 003 195), which is the product of the estimated row number and estimated total column size in bytes. If the join is executed at the DBS, the execution cost is estimated to be 30 142 × 157 (4 732 294) for the transfer of the filtered customers to the DBS. The schedule block 1 will also estimate the cardinality after the join, which is then overwritten by query block 3 to consider the group by (estimated to have 2406 resulting tuples and the column size is 12). Finally, the root scheduler block will take these two plans and only adjust the final execution costs for the plan where the join takes place in the database as the result set of 2046 × 12 (24 552) bytes must now be retrieved leading to an overall execution cost of 56 812 080 for this plan, and this is also the plan that gets picked by the root schedule block.

```
local (cost: 72003195, QR3: 24552)
     |-- ds1 (cost: 0, QR1: 72003195)
     |-- local (cost: 0, QR2: 4732294)

ds1 (cost: 56812080, QR3: 24552)
     |-- ds1 (cost: 0, QR1: 72003195)
     |-- local (cost: 0, QR2: 4732294)

```

Listing 5.1: Sample Plan Object

5.2.3 Building the Final Execution Block and Execution

Taking the plan with the lowest execution cost as a basis, transfer blocks are inserted at the appropriate places in the execution block tree. In particular, when a node in the plan created by an intermediate schedule block indicates a location *ds* for the execution of the join, then a transfer block is inserted before the query block that executes in a different location. For our example, transfer block 1 is inserted before query block 2 shown in Figure 5.2 (c).

From there, the *execute*() function is called recursively starting from the root schedule block. Each block will perform its action. The transfer blocks send the data to the specified location. For executions within the DBS, the query blocks generate the SQL code and send it to the database for execution, otherwise they simply execute on the local data. The schedule blocks simply do nothing at this stage.

6

Experiments

The purpose of LAD is to distribute the execution of queries should the underlying tables be distributed. In particular, LAD can automatically push some computation to a database system should a query involve database tables.

In this section, we analyze the performance of LAD for such distributed datasets. To make sure we cover many different scenarios, we include several queries from the TPC-H benchmark (rewritten in Pandas) as well as automatically generated queries that also use the data of TPC-H. These queries all involve multiple tables, and these tables are distributed across the local machine that runs Pandas and a remote database server.

We compare LAD with two other methods. First, we compare it against an unmodified Pandas, where all necessary data is first loaded locally into the Pandas space at runtime and all query execution happens within Pandas. For the TPC-H queries from the benchmark, we also compare against a setup where an advanced developer manually pushes the relational operations on database

tables to the database system whenever possible. That is, the query is manually written as a mix of SQL statements on the tables of the database system, a manual retrieval of the intermediate results to Pandas, and an execution of the remaining query within Pandas. We use this manual approach to show the effort needed at the client side to write optimized code and compare the performance of this manually optimized code with our automated LAD. Briefly speaking, the user needs to be very familiar with the database system and the underlying data to write the Pandas code that achieves the same result as LAD or the optimal result. We will cover how these queries are written and optimized in detail later on.

When looking at the results, our focus is on end-to-end runtime. We compare the execution time of LAD with Pandas and the manually optimized queries, and analyze how much benefit LAD can achieve for these queries compared to Pandas, and whether it can generate as good execution or better plans as manually optimized code.

6.1 Environment

We use a machine with 2 processors of Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz. The memory is 128 GB and the storage is 1.4 TB. We run the Pandas client and the DBS on the same machine each in a different process. This leads to low data transfer times between the DBS and Pandas, as there is no networking cost across machines. Thus, this likely favors the Pandas-only approach.

6.2 Datasets and Benchmark

We run our experiments on the TPC-H benchmark. It contains 8 tables: *customer*, *lineitem*, *nation*, *orders*, *region*, *supplier*, *part*, and *partsupp*. We list the name abbreviations, the base table row numbers, column numbers, and the table sizes in Table 6.1, ordered by table size. The data is for scale factor (SF) 1. We scale up the table size by increasing the scale factor, thus the final row number will be $SF *$ (base row number).

6.2.1 TPC-H Queries

We select 8 queries from TPC-H queries. These are queries that can be written using Pandas operators that are currently supported by LAD and that acess at least two tables as LAD is designed

Table	Abbr	Number of Columns	Number of Rows	Table Size
region	r	5	5	8192 bytes
nation	n	4	25	8192 bytes
supplier	s	7	10000	1704 kB
customer	c	7	160,000	28 MB
part	p	9	200000	29 MB
partsupp	ps	5	800000	136 MB
orders	o	9	1500000	195 MB
lineitem	l	16	6001215	770 MB

Table 6.1: TPC-H table statistics

for distributed queries. The selected queries cover almost all the functionalities we currently support and are representative of fundamental business ad-hoc queries and some standard data analyzing operations that users will perform commonly. The full description of the 8 queries can be viewed in Appendix A.

As the original queries are SQL statements, we rewrite them using Pandas syntax so that we can run them with LAD and Pandas. For example, the SQL text of query 3 (Listing 6.1) of the TPC-H is converted to Listing 6.2. In the Pandas code, the *customer* table, the *orders* table and *lineitem* table might be a Pandas dataframe or a placeholder for a LAD dataframe depending on whether Pandas or LAD is used. In case of Pandas, should the underlying data be a database table it will be first loaded into this local dataframe. In case of LAD, the LAD dataframe abstract is used and the data is loaded automatically when required.

```
1  select
2    l_orderkey, sum(l_extendedprice * (1 - l_discount)) as revenue,
3    o_orderdate, o_shippriority
4  from
5    customer, orders, lineitem
6  where
7    c_mktsegment = 'BUILDING'
8    and c_custkey = o_custkey
9    and l_orderkey = o_orderkey
10   and o_orderdate < date '1995-03-15'
11   and l_shipdate > date '1995-03-15'
12 group by
13   l_orderkey, o_orderdate, o_shippriority
14 order by
15   revenue desc, o_orderdate
```

Listing 6.1: TPC-H query 3

```
c = customer
o = orders
l = lineitem
c = c[c['c_mktsegment'] == 'BUILDING']
c = c[['c_custkey']]
o = o[o['o_orderdate'] < datetime.date(1995, 3, 15)]
o = o[['o_orderdate', 'o_shippriority', 'o_orderkey', 'o_custkey']]
l = l[l['l_shipdate'] > datetime.date(1995, 3, 15)]
l['revenue'] = l['l_extendedprice'] * (1 - l['l_discount'])
l = l[['l_orderkey', 'revenue']]

t = c.merge(o, left_on='c_custkey', right_on='o_custkey')
t = t.merge(l, left_on='o_orderkey', right_on='l_orderkey')
t = t[['l_orderkey', 'revenue', 'o_orderdate', 'o_shippriority']]
t = t.groupby(('l_orderkey', 'o_orderdate', 'o_shippriority'), sort=False)
.sum()
t.reset_index(inplace=True)
t.sort_values(['revenue', 'o_orderdate'], ascending=[False, True],
inplace=True)
```

Listing 6.2: TPC-H query 3 in Pandas syntax

6.2.2 Auto-Generated Queries

To further test the robustness and the performance of LAD, we create a much larger test set with auto-generated queries that use Pandas/LAD syntax.

Following database SQL query generation approaches like [7] and [1], our lab developed a query generator using Pandas' syntax [15]. The generator takes the database schema information such as table names, column names and types, and the foreign key constraints as inputs. Then, it recursively concatenates relational operations through an intermediate representation of the SQL operations and Pandas functions and produces a set of Pandas queries. The process is shown in Figure 6.1. It also stores various required constraints in these intermediate representations so that different tables can be joined together in a meaningful way, e.g. across primary key/foreign key relationships.

Figure 6.1: Query generation flow

Using this Pandas query generator, we randomly generated 500 queries over the TPC-H dataset. A resulting query sample is shown in Listing 6.3:

Some statistics about these queries are listed in Table 6.2. We record the number of projections,

```
df = nation[nation['n_regionkey'] >= 0][['n_nationkey','n_name','n_regionkey','n_comment']]
.merge(partsupp[partsupp['ps_supplycost'] <= 442]
.merge(supplier[(supplier['s_nationkey'] <= 34) & (supplier['s_acctbal'] <=
12577)][['s_suppkey','s_nationkey','s_phone','s_acctbal']],left_on='ps_suppkey', right_on='s_suppkey'),
left_on='n_nationkey', right_on='s_nationkey')
```

Listing 6.3: Auto-generated query sample

number of filter conditions (note each filter operation could have multiple filter conditions, for example, *ps_supplycost <= 422 AND s_acctbal <= 12577* are two filter conditions), number of join/merge operations, number of groupings and number of aggregations. We also include the number of rows and columns of the result set of each query. The average, median, max, and min values of them are given in the table. These numbers show that we were able to generate a rich variety of queries.

	Projections	Filter Conditions	Join	Groupby	Aggregation	Row	Columns
Average	2.57	9.89	1.84	1.69	1.69	714012.7	11.67
Median	2	10	2	2	2	720	11
Max	9	21	5	5	5	4.39E8	33
Min	0	1	1	0	0	0	3

Table 6.2: Statistics of the auto-generated queries

6.3 Setup

We mimic a distributed environment by partitioning the 8 tables of the TPC-H benchmark across two locations randomly, in form of csv files that can be accessed by the Pandas process and within a PostgreSQL database server. To be more specific, for the purpose of testing different configurations, each location actually has all the tables, but only the specified tables in a given table configuration are allowed to be used by the Pandas process or the DBMS.

For local tables for Pandas, we store them in csv files and preload the required ones into the memory as dataframes. In contrast, the remote database tables are to be loaded only when needed. For LAD, this process is handled automatically, while for Pandas, we manually read the table from the database should the table be used in the given query.

Then, we run the various queries for each experiment and record the end-to-end runtime accordingly. While the loading time for local tables into dataframes is not recorded as mentioned before,

the data transfer time from or to the database will be included in the runtime.

6.4 Test With TPC-H Queries

For the manually written TPC-H queries, we test three methods: Pandas, LAD, and a manually optimized and heuristic-based method. We've already shown how a query looks like for Pandas and LAD in Section 6.2.1. The manual approach uses the heuristic to push down all the operations that can be performed on database tables to the database end. Since we assume we will always retrieve the final result into Pandas, we retrieve the intermediate results into the Pandas environment and perform the rest of the operations within Pandas. As such, we write partially SQL query and partially use the Pandas API, thus splitting the queries to perform operations at different locations as needed.

We use the same query 3 example which takes the *customer*, *orders*, and *lineitem* tables as inputs. Assuming the *lineitem* table resides within the database, then our heuristic-based query would look like Listing 6.4. This requires the user to be familiar with the table definitions in the database. Note that this manual approach can push some operations on database tables to the database but it cannot transfer data from a dataframe to the database. LAD might do this, e.g. to allow a join between a database table and dataframe to be executed at the DBS. Given that manually writing these queries for every possible table configuration is time-consuming, we only did this for the 8 TPC-H queries we selected.

6.4.1 Result Discussion

We show the end-to-end runtime of the three approaches in a bar plot for different queries and table configurations from Figure 6.2 to Figure 6.8. In each of these plots, the x-axis and the color distinguish the methods we use, and the y-axis is the runtime in seconds. Each figure displays the results for one of the 8 selected queries. Within each figure, each subplot shows the result for a specific table configuration, and the particular distribution of the tables is listed on top of the subplot. Those letters are the abbreviations of the TPC-H tables shown in Table 6.1

Note that the number of sub-plots is different for different queries as they involve a different number of tables - which influences the number of possible configurations. Note that we only

```
sql = """SELECT l_orderkey, l_extendedprice*(1-l_discount) AS revenue
    FROM lineitem
    WHERE l_shipdate > date '1995-3-15'"""

# here conn is any database connection object
l = pd.read_sql_query(sql, conn)
c = customer
o = orders

c = c[c['c_mktsegment'] == 'BUILDING']
c = c[['c_custkey']]
o = o[o['o_orderdate'] < datetime.date(1995, 3, 15)]
o = o[['o_orderdate', 'o_shippriority', 'o_orderkey', 'o_custkey']]

t = c.merge(o, left_on='c_custkey', right_on='o_custkey')
t = t.merge(l, left_on='o_orderkey', right_on='l_orderkey')

...

```

Listing 6.4: TPC-H query 3 manually written with the heuristic-based method

consider configurations where at least one table resides in Pandas and another in the database.

For almost all queries, the pure Pandas approach takes the longest time, varying from a few seconds to around 70 seconds. LAD and the heuristic-based method in general have similar performance and take much less time. There are cases where the heuristic-based method outperforms LAD slightly by a margin of approximately 1 second. However, depending on the query and table configurations, for large tables and time-consuming queries, LAD can save up to tens of seconds even compared to the manual approach, which is a difference of magnitude.

The execution time of Pandas is mostly affected by the distribution of the tables. Loading the table from or to the database includes data serialization, transfer, and conversion to formats required by Pandas. This can take a long time if the data size is big as Pandas is notorious for its inefficient data loading process [27]. For the 770 MB *lineitem* table, the loading time from the database to a Pandas dataframe already requires 46 seconds. This explains why whenever the *lineitem* table is in the database, the execution time of Pandas always exceeds 50 seconds. The situation is alleviated for LAD and the heuristic-based method, as these two approaches usually do not load the entire table to the Pandas dataframe, in particular when the query contains selections or projections on the table. Using LAD, only the intermediate result is transferred to or from the database based on the execution plan generated by the LAD scheduler. For the heuristic-based method, as we rewrite the query in a way that some filter or projection operations are performed as SQL queries on the

database table, only the results of these operations are transferred from the database to Pandas space. Thus, the resulting execution times for the heuristic-based methods and LAD are similar for many configurations.

Additionally, the order of operations, how the tables are joined, and the output table cardinality can also affect the performance. The database engine has a query rewriter and a query optimizer, which rearrange the operations in a query for faster execution. However, this functionality is not present within Pandas due to its eager evaluation nature. Our hand-written Pandas code does projections and filters before the join operations, which actually favors Pandas. However, database systems are very optimized for relational operations and have an advantage over Pandas. Due to this reason, some queries that join large tables result in a long execution for Pandas, such as query 3 when both *customer* and *lineitem* tables are in the DBS (first subplot in Figure 6.3). The loading time for the *lineitem* table is around 50 seconds, while customer is a relatively small table whose loading time is negligible. Still, Pandas takes an extra 10 seconds to finish the execution compared to LAD who executes the join in the DBS and only transfer the result to Pandas. Additionally, the server usually has a higher computation power than the client's machine. In our setting, as we run both Pandas and the DBS on the same machine, this factor is not considered. However, in real life, the difference in computing power could have an additional impact.

The performance of LAD and the heuristic-based method are similar. Essentially, LAD splits the query into subqueries and executes each part at the system where the underlying data is located. In the heuristic-based method, we manually split and write the corresponding queries. For some table configurations of queries 4, 5, and 18, LAD greatly outperforms the heuristic-based method. The heuristic-based method only pushes down the filter and projection operations to the DBS should the data be within the database, a very simple optimization. In contrast, LAD uses the internal scheduler that gathers all execution plans, considering alternative execution places (Pandas or DBS) for each operation, then it calculates and estimates the size of the intermediate result sets and chooses the one that leads to the lowest data transfer cost. For example, for query 18, the first table configuration has the *customer* table in Pandas while the *lineitem* and *orders* tables are on the database server. LAD detects the final result set is a very small set, and directly sends the *customer* table to the DBS server instead of retrieving the reduced *lineitem* and *orders* tables to the client side. All the remaining operations are executed in the database as well until the final

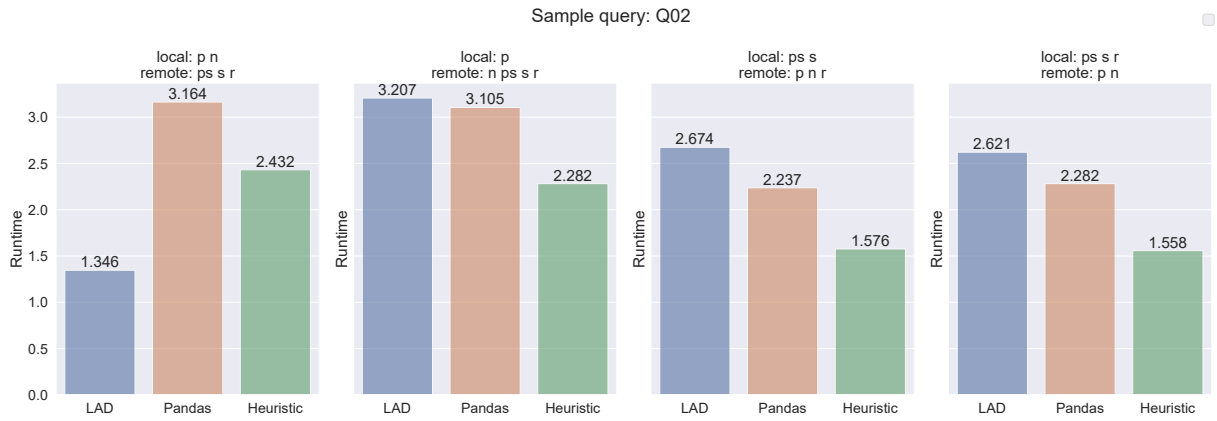


Figure 6.2: Runtime of TPC-H Query 2

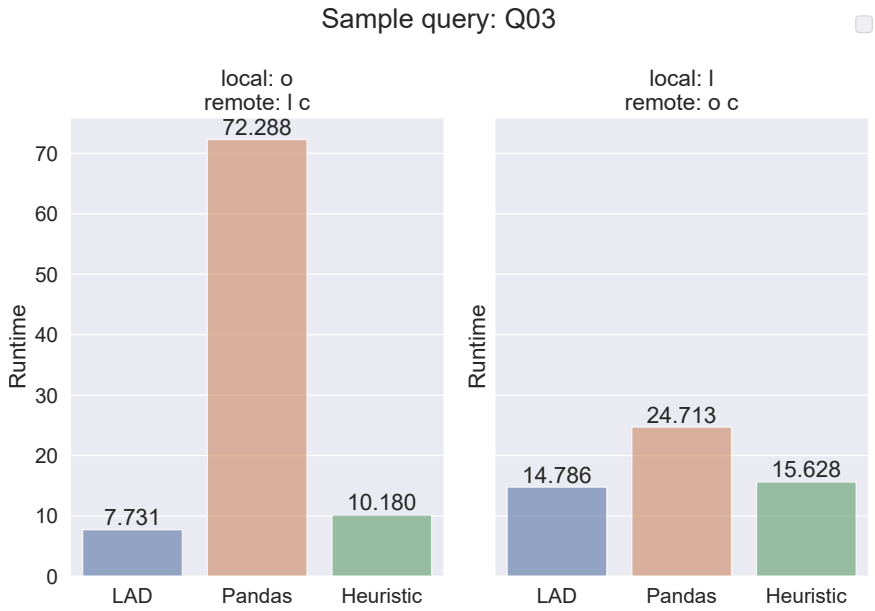


Figure 6.3: Runtime of TPC-H Query 3

result is sent back. That's how LAD gets a large performance gain over both Pandas and the heuristic-based method over these table configurations and queries.

Among all the results, Figure 6.2 (Query 2) is an outlier. For this query, LAD, in three out of four table configurations, takes noticeably longer time than Pandas and the heuristic-based method. The difference between LAD and Pandas is around 0.5 seconds and the difference between LAD and the heuristic-based method varies from .5 seconds to 1 second. Regardless of the table configurations, the overall runtime of query 2 is relatively short compared to other queries, because the involved tables are among the smaller tables. LAD requires extra steps for query planning, such as traversing

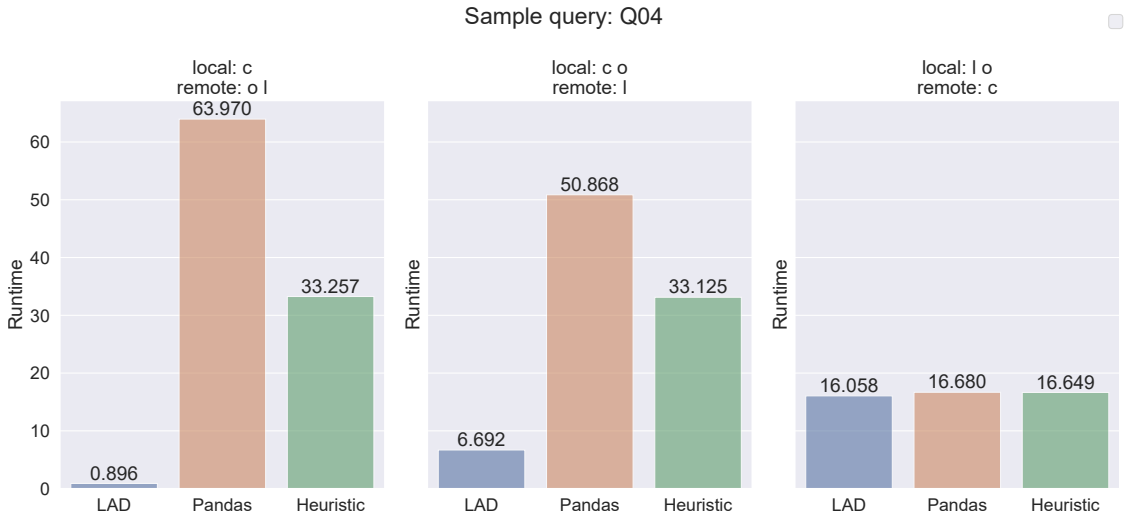


Figure 6.4: Runtime of TPC-H Query 4

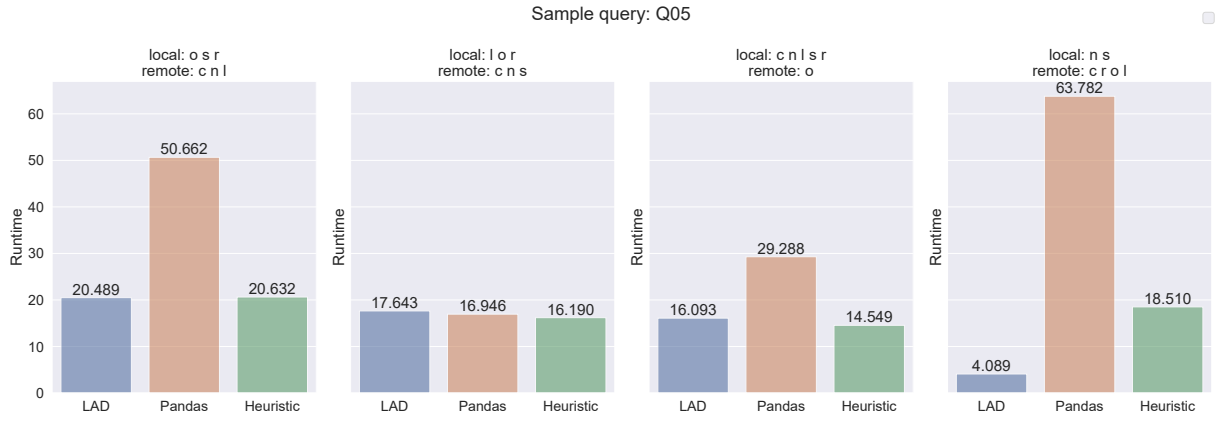


Figure 6.5: Runtime of TPC-H Query 5

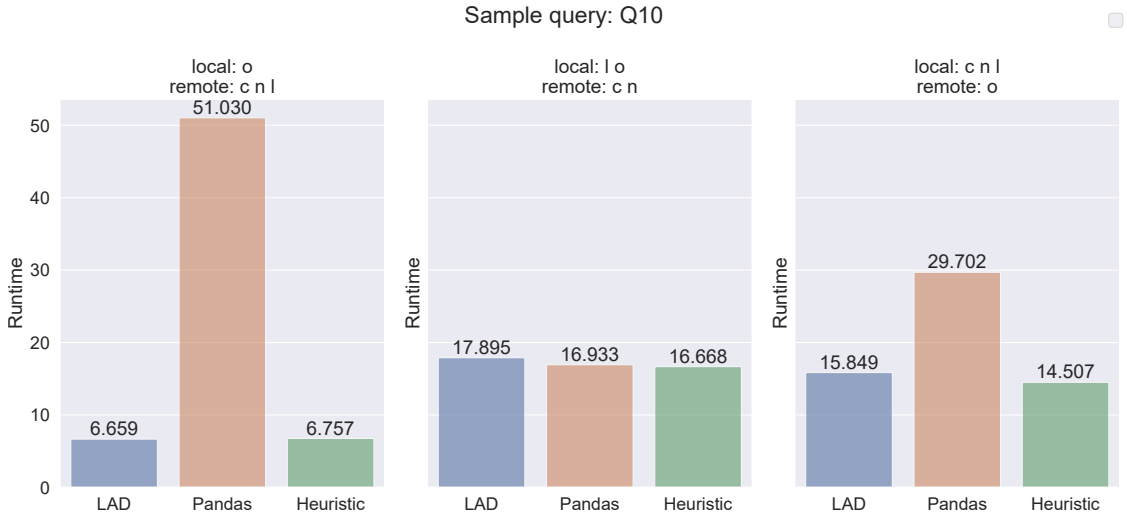


Figure 6.6: Runtime of TPC-H Query 10

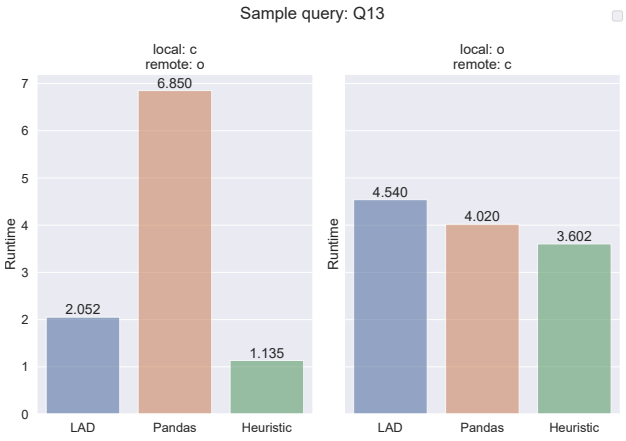


Figure 6.7: Runtime of TPC-H Query 13

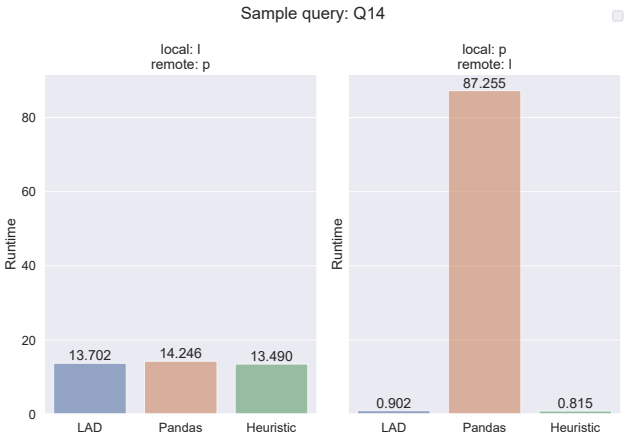


Figure 6.8: Runtime of TPC-H Query 14

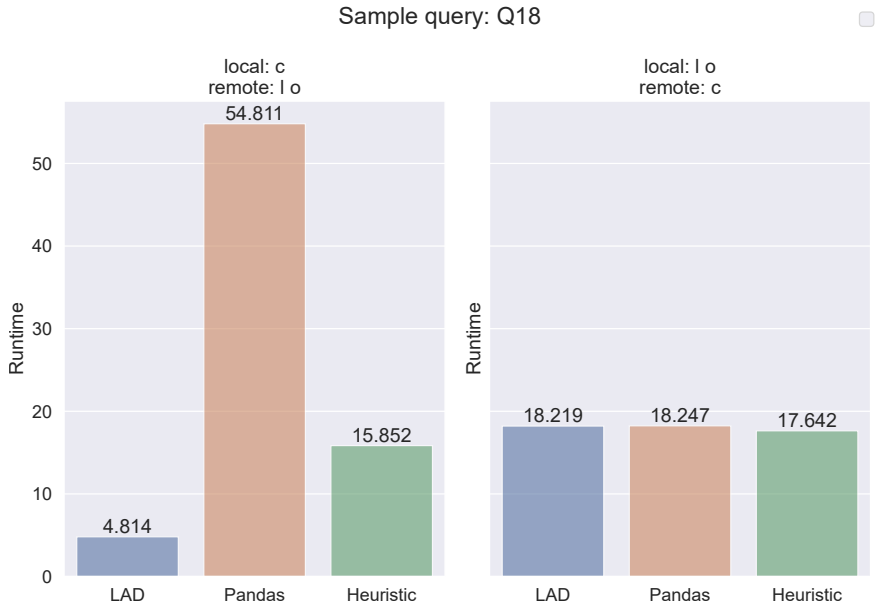


Figure 6.9: Runtime of TPC-H Query 18

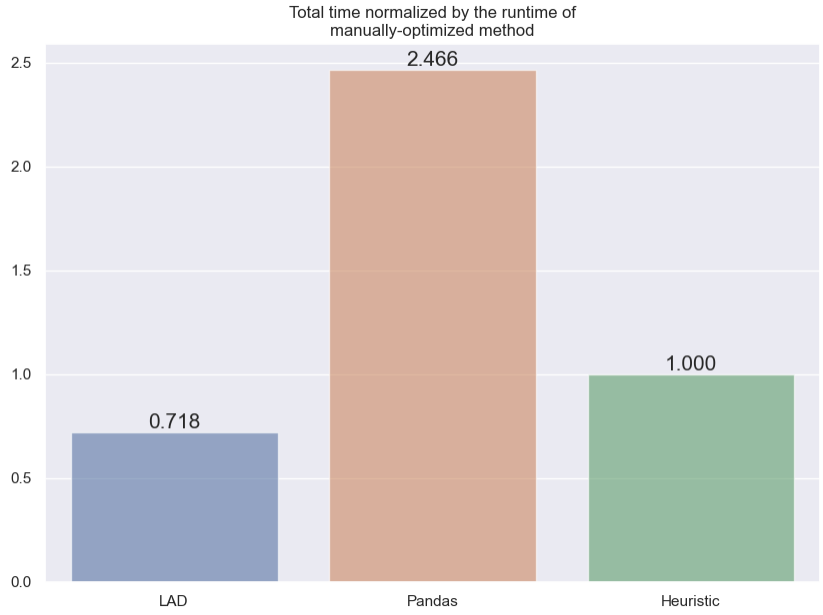


Figure 6.10: Normalized sum of execution times for all queries

the lineage tree, building a plan if some tables reside in the database, and querying the meta information from the database, etc. The planning itself can take tens of milliseconds to around a second depending on whether the scheduler needs to go through a certain number of local tuples to get estimated statistics about some columns. This overhead of LAD might be the reason why it's slower than Pandas and the heuristic-based method. Also, as LAD's estimation of the output table size is not 100% accurate, it can make wrong decisions when some small tables are involved and when the query is tricky. These might also be the reason behind LAD's slightly worse performance on some table configurations of other queries.

As a final comparison, we add the query execution times for all queries and table configurations together and normalize the data by dividing it by the total execution time of the heuristic-based method. The result is shown in Figure 6.10. Because we divide the result by the total runtime of the heuristic-based method, its value is 1. Pandas is sensitive to changes in table distribution and is largely affected by the location of the *lineitem* table. The long loading time makes it the worst of all. The performance of LAD is relatively consistent through all the queries, and overall, it is 3.43 times faster than Pandas and 1.39 times faster than the heuristic approach.

6.5 Test With Auto-Generated Queries

To cover a wide range of use cases and test our framework's performance in a more generic way, we use 500 auto-generated queries for this experiment. The setup is similar to the TPC-H queries. However, we do not compare against the manually optimized queries as it is infeasible to do so for 500 queries. Thus we only compare LAD against Pandas.

For each of the auto-generated queries (like the one shown in Listing 6.3), we read it as a string, which is parsed to obtain all the involved tables. Then we load the corresponding tables according to a random table configuration. Same as before, only the tables local to Pandas and LAD are loaded into the dataframes, and the time for it is not counted. In the end, the end-to-end query execution time is recorded.

We repeat the above procedure for each of the three table configurations listed below.

1. Local to Pandas and LAD: *nation*, *orders*, *part*, *partsupp*, *region*

   Remote in the DBMS: *customer*, *lineitem*, *supplier*

2. Local to Pandas and LAD: *nation*, *customer*, *orders*, *lineitem*, *partsupp*, *region*, *supplier*

   Remote in the DBMS: *part*

3. Local to Pandas and LAD: *nation*, *lineitem*, *supplier*

   Remote in the DBMS: *customer*, *orders*, *part*, *partsupp*, *region*

6.5.1 Result Discussion

We display the results in bar plots through Figure 6.11 to Figure 6.14. In all plots, we use red to represent LAD's runtime and blue to represent Pandas' runtime. Figure 6.11 shows the results for all the configurations combined while the other figures focus on one configuration.

In the bar plots, each bin (the x-axis) is a small range of runtime in seconds. The y-axis represents the count of query executions that fall in this range of time. Because most of the queries are very short while a small portion of the queries takes a long time, in all the bar plots, there is a long tail on the right side. In order to better view this part of the data, we zoom in on the figure by having the x-axis only show the longer runtimes with wider ranges and setting the maximum y-axis value to 10, shown in the embedded yellow sub-figure.

For all configurations, it is visible that more data samples from LAD gather at the lower range of runtime, and in general, LAD takes less time to complete execution. Figure 6.11, that combines the results from all configurations, shows this trend clearly. We notice that around 76% of the queries executed with LAD stay within the lowest time range whereas only 50% of the queries executed with Pandas have such small runtime. There are also more data samples of Pandas that fall onto the right side as the shape of red bars is generally wider and taller as shown in the zoomed-in subfigure. The trend is more obvious with configurations 2 and 3.

However, there are a few LAD executions requiring a long time. These are due to configuration 1 (Figure 6.12). For table configuration 1, overall the execution time is much longer compared to the other two configurations as the *lineitem* table is located within the DBS and either the *lineitem* table itself or at least the result, which tends to be large, has to be moved to Pandas. When inspecting the execution plans generated by LAD and PostgreSQL, which LAD's scheduler heavily relies on, we observe that the table output size estimation provided by PostgreSQL can be very inaccurate. PostgreSQL's query planner uses a cost-based optimization strategy to evaluate the best way to execute a query. This strategy relies on the statistics collected by PostgreSQL about the tables and indexes in the database. However, since the statistics are collected using random sampling if the table size is big, that can cause problems. In this case, the incorrect estimation of table output size has impacted the decision-making process of LAD by making it think the final output table size is 1 million tuples less than it actually was. Thus, LAD chooses to execute everything within the database and retrieve the final result instead of exporting the intermediate results from the database.

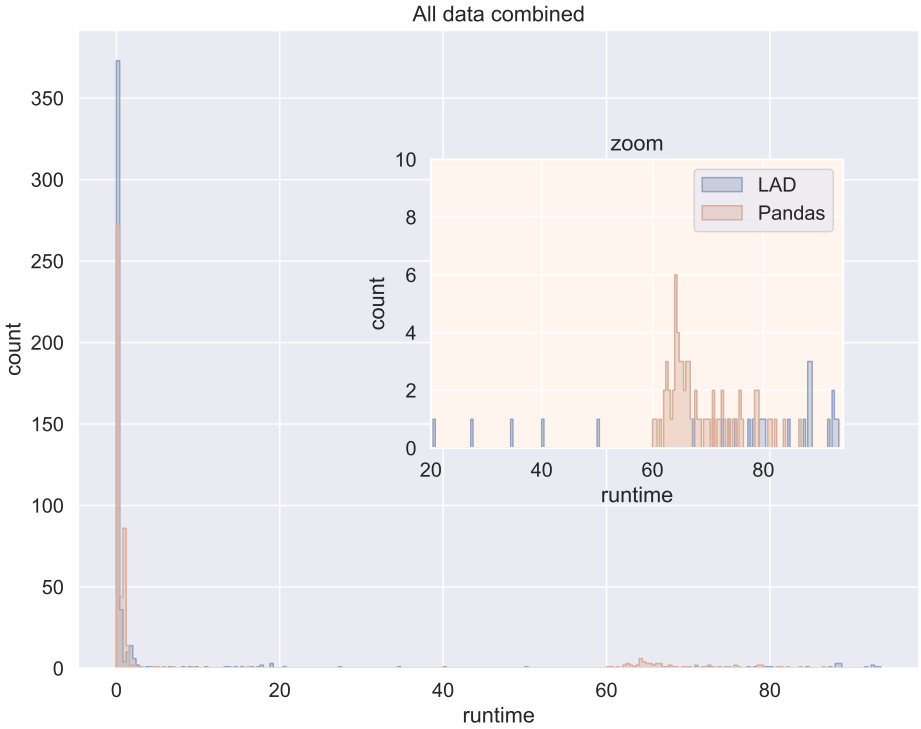


Figure 6.11: Histogram of random query runtimes for all configurations

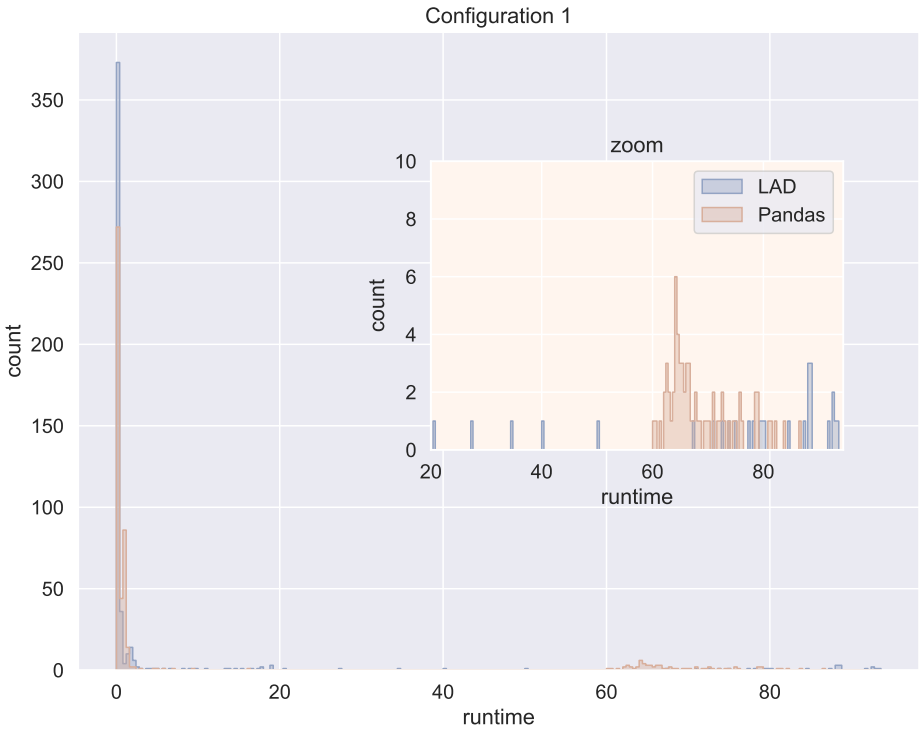


Figure 6.12: Histogram of random query runtimes for configuration 1

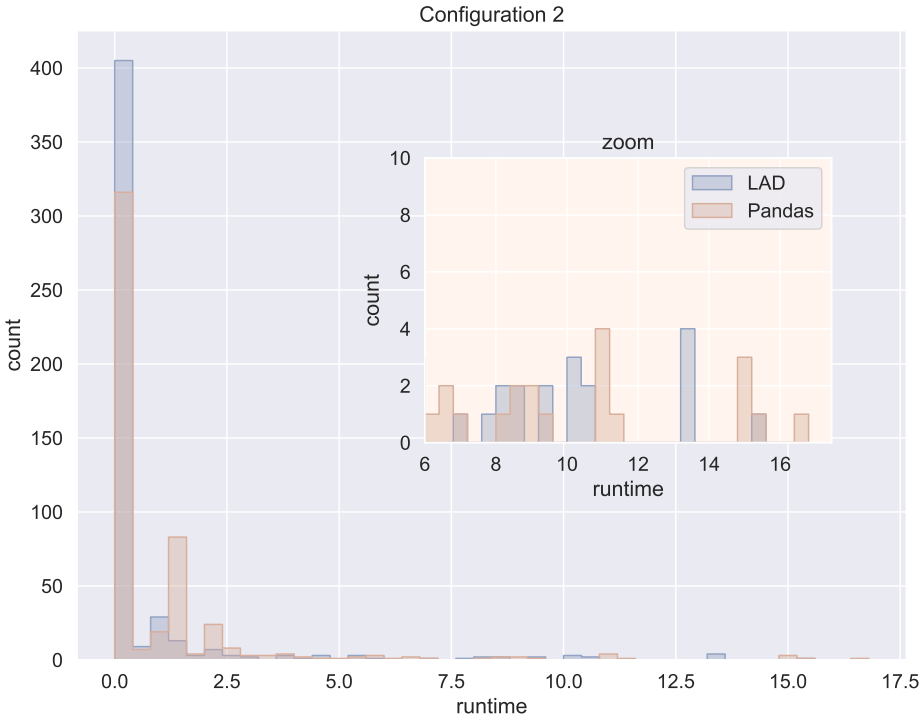


Figure 6.13: Histogram of random query runtimes for configuration 2

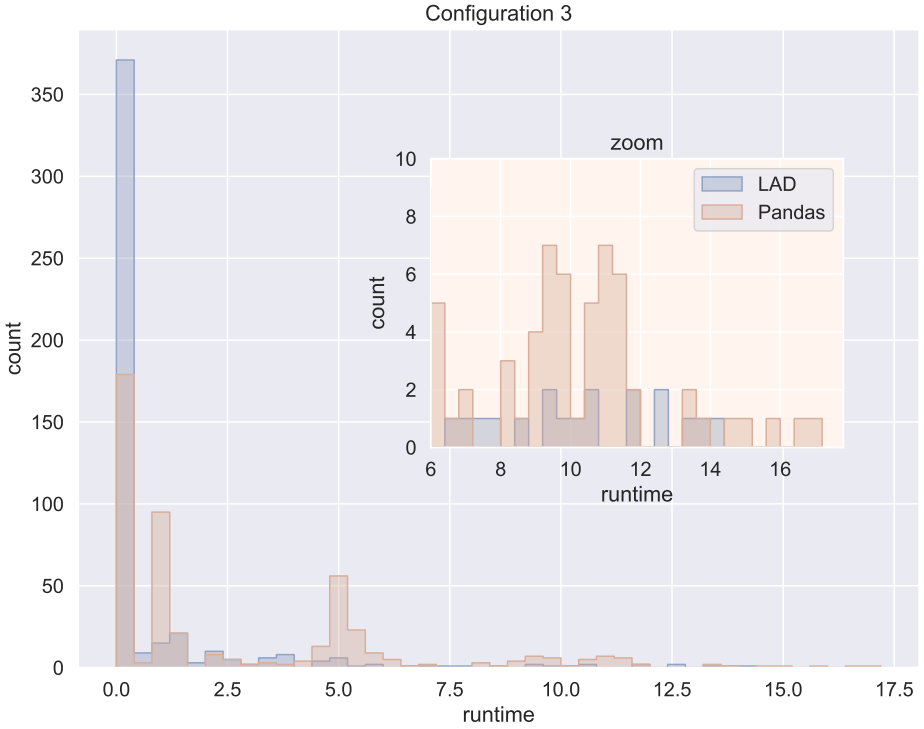


Figure 6.14: Histogram of random query runtimes for configuration 3

Another reason for the overall slow time is due to the slower execution within the PostgreSQL engine on these queries. We randomly selected 10 queries from these queries on which LAD executes for over 60 seconds. It takes on average 22.76 seconds for PostgreSQL to finish these queries alone assuming all the required tables are already within the database, and having a very large return table size only worsens the problem.

To better view how the table distribution affects the results, we also include scatter plots for the three different configurations in Figure 6.15 to 6.17. A scatter plot displays the relationship between the runtime (in seconds) and the ratio of the local input size of data in dataframes to the remote input size of tables in the DBS, that is the sum of the sizes of local tables divided by the sum of the sizes of remote tables. This ratio is not continuous as there are a limited number of combinations to distribute these tables. Additionally, the ratio can be largely affected by larger tables like *lineitem* and *orders*, which is why there are 0.0 and 1.0 on the x-axis even though our setup does not allow all tables to be at one location.

The Pandas data points are represented by red dots, while the LAD data points are represented by blue dots. There are 500 sample points for each scatter plot representing 500 queries. The color is set to half transparent, which means the more data points gather close to each other, the darker the color is. Since we plot LAD after Pandas, the red dots might be shielded by the blue dots and become invisible if the two methods take about the same amount of time.

Upon examination of the scatter plot, LAD data points are mostly clustered towards the lower side of the figure, indicating that LAD has generally lower runtimes than Pandas for the majority of data points, especially for configurations 2 and 3. We can observe a trend that LAD outperforms Pandas when more input data is within the database, and as the local input size increases, it gradually performs similarly to Pandas.

We can also observe the same pattern for configuration 1 that we already saw in the histogram plot of Figure 6.12 where LAD actually spends a longer time than Pandas to finish a query when the local/remote size ratio is between 0.0 and 0.14. These all correspond to having large tables in the database, which can lead to the inaccurate estimation made by Postgres and LAD scheduler as well as relatively large return table cardinality, which then can hinder the performance of LAD.

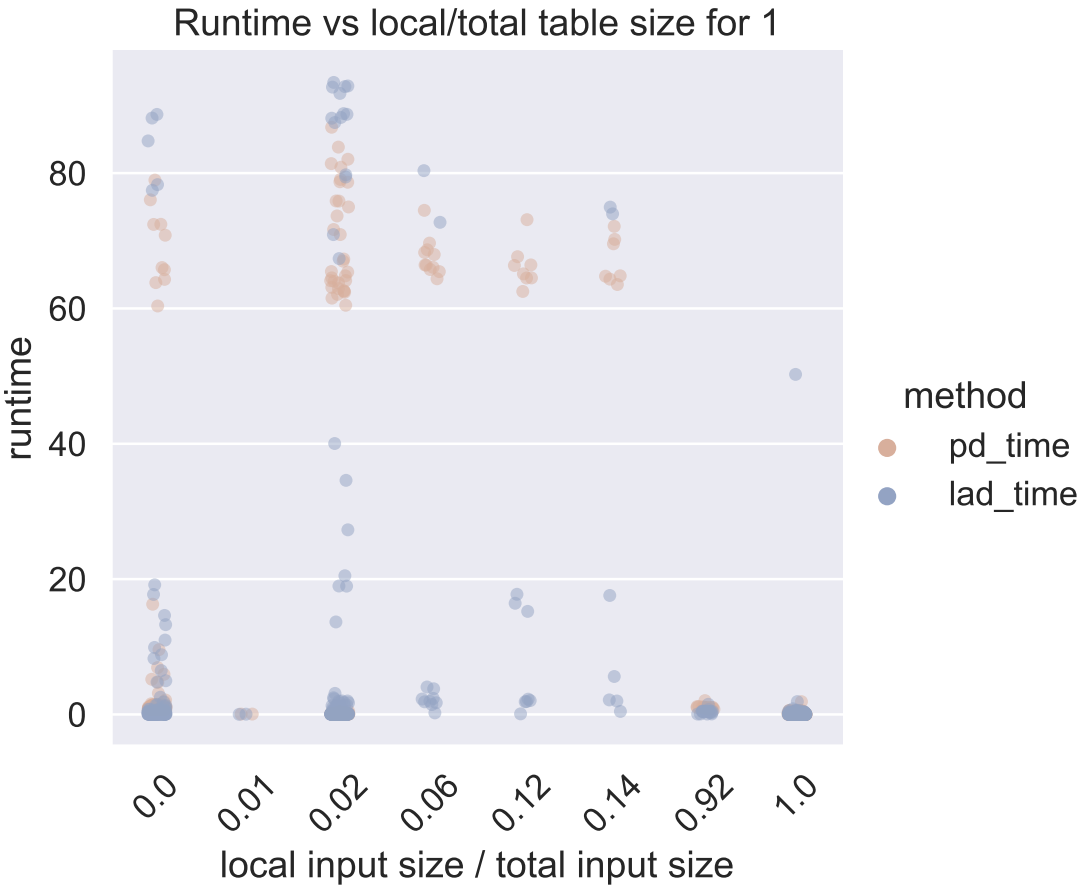


Figure 6.15: Scatter plot of random query runtimes for configuration 1

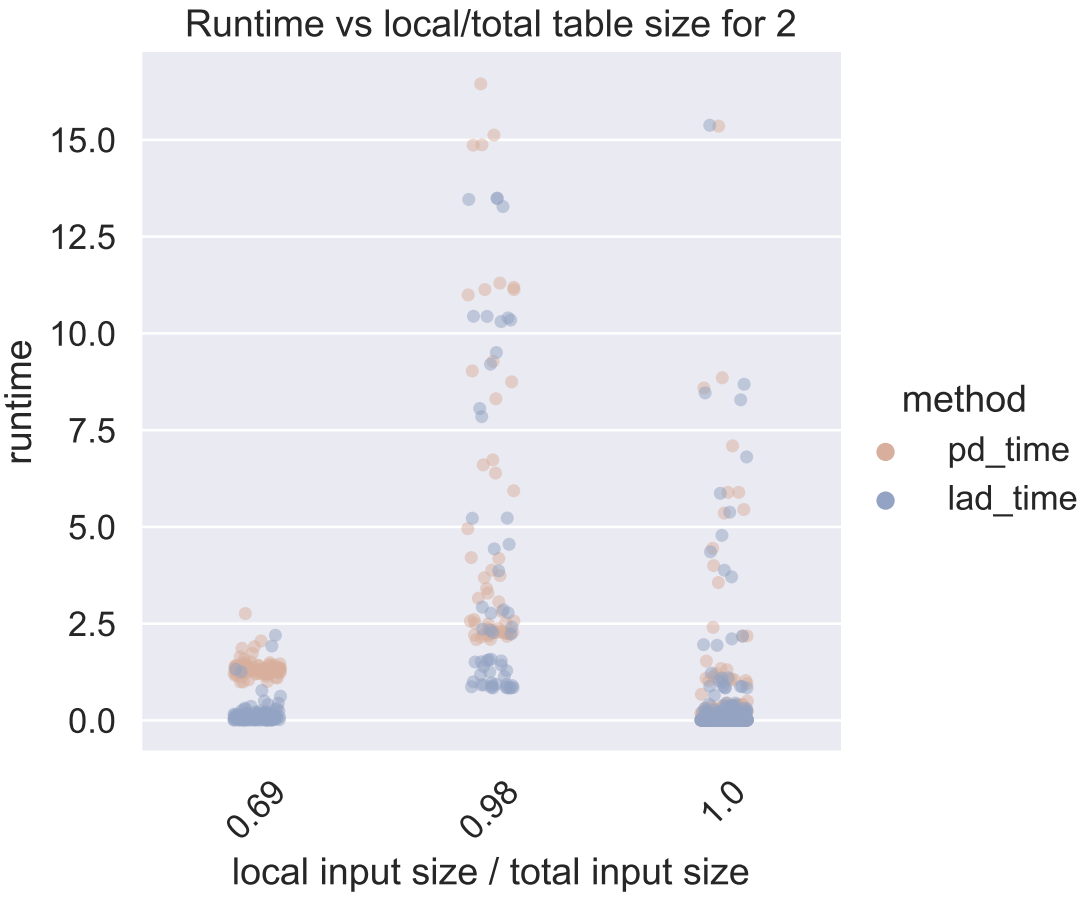


Figure 6.16: Scatter plot of random query runtimes for configuration 2

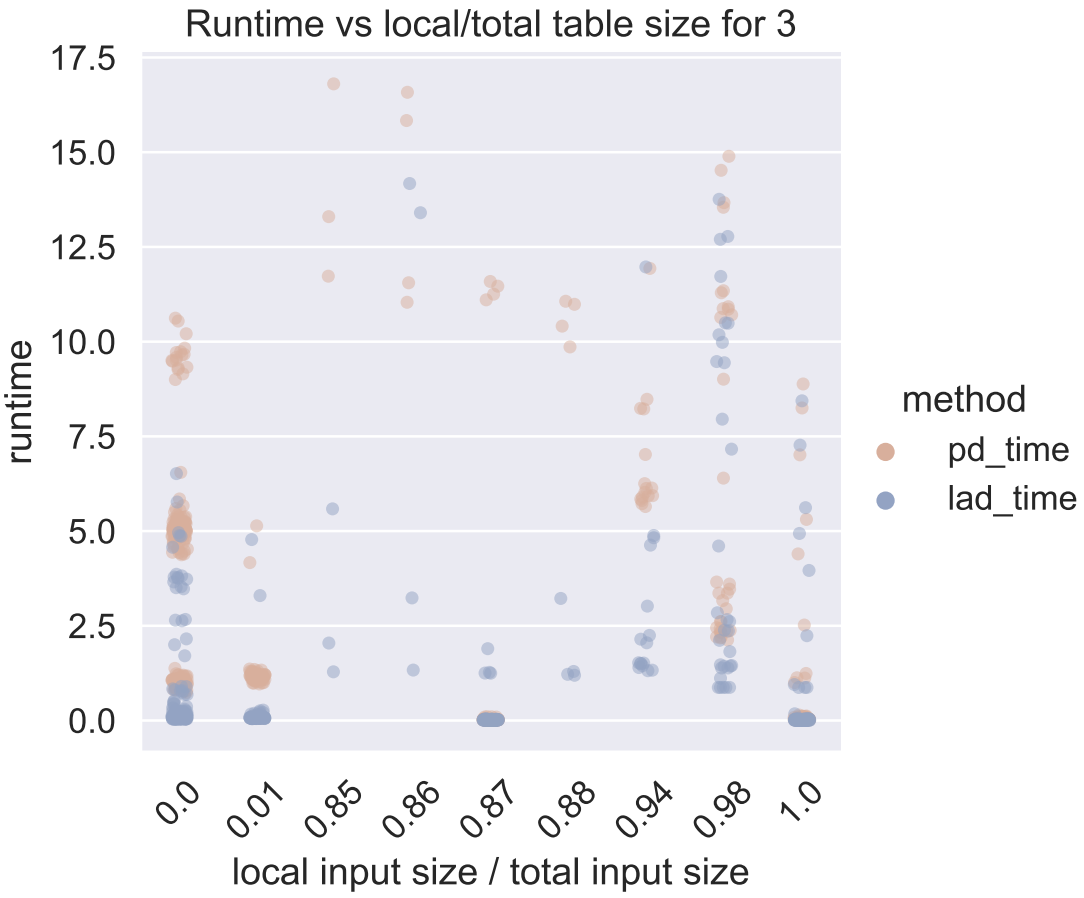


Figure 6.17: Scatter plot of random query runtimes for configuration 3

6.6 Experiments Discussion

Our experimental results show that LAD generally outperforms a pure Pandas approach on heterogenous data queries by a large margin, especially when many tables reside in a remote DBS. LAD is faster than the pure Pandas approach by a factor of 3.43 over TPC-H benchmarks. Most of the time, LAD has a similar or better performance than a heuristic-based approach with manually optimized queries. At the same time, LAD also reduced the manual work required from users as they can treat data sets the same, independently of whether the data comes from the local file system or resides in a DBS, and do not need to worry about writing SQL or considering data transfer.

However, for the randomly generated query tests, even though in most scenarios, LAD is better than Pandas, it exhibits worse performance in a few configurations, which mainly results from LAD's inaccurate estimation of the intermediate table size, hence leading to suboptimal query execution plans. In some queries with a very short runtime, LAD can also suffer from its overhead, such as traversing the lineage tree, building a plan, and querying meta information from the database, which can result in slower performance.

7

Conclusion

In this thesis we propose LAD, a dataframe library with the same interface as Pandas, that supports distributed relational queries where some of the tables reside in a relational DBS and others might be in a file system and/or local to the LAD execution environment.

LAD provides users with a unified abstraction of a dataframe so that they do not need to know where execution actually happens. LAD automatically executes relational operators at the data source that allows for faster execution because of less data transfer costs. LAD offers a lazy evaluation strategy over data that resides in the database system, allowing several relational operators to be bundled in a single SQL statement for optimized execution. LAD's scheduler then uses heuristics to decide where to execute operations to keep data transfer as low as possible and takes care to automatically transfer data from one data source to another. Our experimental results show that LAD generally outperforms approaches where execution happens only locally within a Pandas execution environment and even when experienced programmers push some of the operations

manually to a remote database system — and that often by a large margin. Its planning phase, however, has some overhead that can have an impact if query execution itself is relatively short, and its scheduler depends on good cardinality estimation for query results in order to determine the best execution plan.

Overall, LAD's benefits outweigh its drawbacks, and we recommend it as the preferred method for relational and analyzing tasks on heterogeneous data. The ability to work on tables that reside at different locations and run complex queries in a distributed manner while offering the familiar and popular Pandas API makes LAD a good tool for improving the efficiency of data analysis processes during the data exploratory phase.

7.1 Future Work

We see several opportunities for future work.

The architecture we proposed for LAD considers the possibility of having multiple data sources. However, currently, the LAD scheduler can only plan and automate the distributed query execution with two data sources, namely, using the local Pandas environment and a remote PostgreSQL DBS. We would like to support more data sources by extending the LAD scheduler's functionalities. This potentially would enlarge the search space of the execution plans, thus current methods should be optimized as well. Additionally, other row-based DBS or column-based DBS should be supported. For each of them, a new database adaptor module needs to be implemented. Furthermore, in order to transfer data to the DBS, we currently use temporary tables. It might be interesting to explore other possibilities such as foreign data wrappers that were introduced into the SQL standard as part of the SQL/MED extension [17].

So far, our scheduler does not reorder operations in the lineage tree. If all sources of a lineage tree (or branch of a lineage tree) are in the database, then LAD will send the entire tree, including, selections, projections and the joins to the DBS as a single SQL statement, and the DBS planner will often do some automatic reordering in order to execute selections and projections before joins as this typically reduces the size of intermediate results and improves query execution time. However, if the lineage tree contains a join between two tables that reside in different data sources, and only afterwards some selections and/or projections, then our LAD scheduler will execute the join

before the selections/projections. An extension to our LAD scheduler thus could deploy traditional query rewrite mechanisms to perform an algebraic optimization of the lineage tree when creating the execution tree pushing selections and projections below the join whenever possible.

A further line of research would be to improve the cost estimation used by the scheduler. We have already seen that with better cost estimations, LAD might achieve better performance. To do so, more features could be considered such as the execution time of each operation, the network bandwidth, and the available computational resources of each data source. Potentially, an ML model (such as [13] and [16]) could be used to substitute the heuristic-based algorithm in the scheduler or optimize the construction of an execution plan.

Bibliography

[1] Shadi Abdul Khalek and Sarfraz Khurshid. Automated SQL query generation for systematic testing of database engines. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, page 329–332, New York, NY, USA, 2010. Association for Computing Machinery.

[2] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery.

[3] Jianhao Cao. Data transfer between PostgreSQL and its embedded Python environment for in-database analytics. Master Thesis, School of Computer Science, McGill University, 2021.

[4] Transaction Processing Performance Council. TPC benchmark H. [https://www.tpc.org/, accessed 12-September-2022].

[5] The Psycopg Team Daniele Varrazzo. Postgresql driver for python — psycopg. [https://www.psycopg.org/, accessed 18-April-2022].

[6] Umeshwar Dayal. Processing queries over generalization hierarchies in a multidatabase system. In *Proceedings of the 9th International Conference on Very Large Data Bases*, VLDB '83, page 342–353, San Francisco, CA, USA, 1983. Morgan Kaufmann Publishers Inc.

[7] Claudio de la Riva, María José Suárez-Cabal, and Javier Tuya. Constraint-based test database generation for SQL queries. In *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, page 67–74, New York, NY, USA, 2010. Association for Computing Machinery.

[8] The Pandas development team. Pandas. [https://pandas.pydata.org/, accessed 28-March-2022].

[9] Joseph Vinish D'silva, Florestan De Moor, and Bettina Kemme. Aida: Abstraction for advanced in-database analytics. *Proc. VLDB Endow.*, 11(11):1400–1413, Jul 2018.

[10] The Apache Software Foundation. Quickstart: Dataframe — pyspark 3.3.2 documentation. [https://spark.apache.org/docs/latest/api/python/getting_started/quickstart_df.html, accessed 02-December-2022].

[11] The PostgreSQL Global Development Group. Row estimation examples. [https://www.postgresql.org/docs/current/row-estimation-examples.html, accessed 15-March-2022].

[12] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. Putting Pandas in a box. CIDR, 01 2021.

[13] Benjamin Hilprecht and Carsten Binnig. Zero-shot cost models for out-of-the-box learned cost prediction. *Proc. VLDB Endow.*, 15(11):2361–2374, Jul 2022.

[14] Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, Dec 2000.

[15] Dailun Li. Automated query generation for pandas. Undergrad project, School of Computer Science, McGill University, 2022.

[16] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Learning to steer query optimizers. *CoRR*, abs/2004.03814, 2020.

[17] Jim Melton, Jan-Eike Michels, Vanja Josifovski, Krishna Kulkarni, Peter Schwarz, and Kathy Zeidenstein. SQL and management of external data. *SIGMOD Rec.*, 30(1):70–77, Mar 2001.

[18] H. Naacke, G. Gardarin, and A. Tomasic. Leveraging mediator cost models with heterogeneous data sources. In *Proceedings 14th International Conference on Data Engineering*, pages 351–360, 1998.

[19] Travis Oliphant. *Guide to NumPy*. 2006.

[20] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems (2nd Ed.)*. Prentice-Hall, Inc., USA, 1999.

[21] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. Towards scalable dataframe systems. *Proc. VLDB Endow.*, 13(12):2033–2046, Jul 2020.

[22] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. *SIGMOD Rec.*, 14(2):256–276, jun 1984.

[23] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company, 2020.

[24] The Kaggle team. State of data science and machine learning. [https://www.kaggle.com/kaggle-survey-2021 accessed 07-November-2022].

[25] The Numpy team. State of data science and machine learning. [https://numpy.org/, accessed 19-December-2022].

[26] Uwe. The one Pandas internal I teach all my new colleagues: the blockmanager. [https://uwekorn.com/2020/05/24/the-one-pandas-internal.html, access 04-November-2021].

[27] Xiaoying Wang, Weiyuan Wu, Jinze Wu, Yizhou Chen, Nick Zrymiak, Changbo Qu, Lampros Flokas, George Chow, Jiannan Wang, Tianzheng Wang, Eugene Wu, and Qingqing Zhou. Connectorx: Accelerating data loading from databases to dataframes. *Proc. VLDB Endow.*, 15(11):2994–3003, Jul 2022.

[28] Gio Wiederhold. Intelligent integration of information. *SIGMOD Rec.*, 22(2):434–437, Jun 1993.

[29] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, 2012. USENIX Association.

A

TPC-H queries in Pandas Syntax

```
p = p[p['p_size'] == 15]
p = p[p['p_type'].str.contains('^.*BRASS$')]
p = p[['p_partkey', 'p_mfgr']]
ps = ps[['ps_suppkey', 'ps_supplycost', 'ps_partkey']]

s = s[['s_nationkey', 's_suppkey', 's_acctbal', 's_name', 's_address', 's_phone', 's_comment']]
n = n[['n_nationkey', 'n_regionkey', 'n_name']]
r = r[r['r_name'] == 'EUROPE']
r = r[['r_regionkey']]

j = ps.merge(s, left_on='ps_suppkey', right_on='s_suppkey')
j = j.merge(n, left_on='s_nationkey', right_on='n_nationkey')
j = j.merge(r, left_on='n_regionkey', right_on='r_regionkey')

ti = j[['ps_partkey', 'ps_supplycost']].groupby('ps_partkey').min()
ti.reset_index(inplace=True)

t = j.merge(p, left_on='ps_partkey', right_on='p_partkey')
t = t.merge(ti, left_on=['ps_partkey', 'ps_supplycost'], right_on=['ps_partkey', 'ps_supplycost'])
t = t[['s_acctbal', 's_name', 'n_name', 'p_partkey', 'p_mfgr', 's_address', 's_phone', 's_comment']]
t.sort_values(['s_acctbal', 'n_name', 's_name', 'p_partkey'], ascending=[False, True, True, True])
```

Listing A.1: Query 02

```
c = c[c['c_mktsegment'] == 'BUILDING']['c_custkey']
o = o[o['o_orderdate'] < np.datetime64('1995-03-15')]
o = o[['o_orderdate', 'o_shippriority', 'o_orderkey', 'o_custkey']]
l = l[l['l_shipdate'] > np.datetime64('1995-03-15')]
l['revenue'] = l['l_extendedprice'] * (1 - l['l_discount'])
l = l[['l_orderkey', 'revenue']]

t = c.merge(o, left_on='c_custkey', right_on='o_custkey', how='inner')
t = t.merge(l, left_on='o_orderkey', right_on='l_orderkey', how='inner')
t = t[['l_orderkey', 'revenue', 'o_orderdate', 'o_shippriority']]
t = t.groupby(['l_orderkey', 'o_orderdate', 'o_shippriority'], sort=False).agg({'revenue': 'sum'})
```

Listing A.2: Query 03

```
l = l[l['l_commitdate'] < l['l_receiptdate']]
o = o[(o['o_orderdate'] >= np.datetime64('1993-07-01'))
      &(o['o_orderdate'] < np.datetime64('1993-10-01'))]
t = o.merge(l, left_on='o_orderkey', right_on='l_orderkey')
t = t[['o_orderpriority', 'o_orderkey']]

t = t.groupby('o_orderpriority').count()
t.sort_values('o_orderpriority')
```

Listing A.3: Query 04

```
c = c[['c_custkey', 'c_nationkey']]

s = s[['s_nationkey', 's_suppkey']]
n = n[['n_name', 'n_nationkey', 'n_regionkey']]

o = o[(o['o_orderdate'] >= np.datetime64('1994-01-01'))
     &(o['o_orderdate'] < np.datetime64('1995-01-01'))]
o = o[['o_orderkey', 'o_custkey']]
l = l[['l_suppkey', 'l_orderkey', 'l_discount', 'l_extendedprice']]
r = r[r['r_name'] == 'ASIA']
r = r[['r_regionkey']]

t = c.merge(o, left_on='c_custkey', right_on='o_custkey')
t = t.merge(l, left_on='o_orderkey', right_on='l_orderkey')
t = t.merge(s, left_on=['l_suppkey', 'c_nationkey'], right_on=['s_suppkey', 's_nationkey'])
t = t.merge(n, left_on='s_nationkey', right_on='n_nationkey')
t = t.merge(r, left_on='n_regionkey', right_on='r_regionkey')
t['revenue'] = t['l_extendedprice'] * (1 - t['l_discount'])
t = t[['n_name', 'revenue']]
t = t.groupby(('n_name'), sort=False).sum()
t.reset_index(inplace=True)
t.sort_values('revenue', ascending=False)
```

Listing A.4: Query 05

```
c = c[['c_custkey', 'c_nationkey', 'c_name', 'c_acctbal', 'c_address', 'c_phone', 'c_comment']]

o = o[(o['o_orderdate'] >= np.datetime64('1993-10-01'))
     &(o['o_orderdate'] < np.datetime64('1994-01-01'))]
o = o[['o_orderkey', 'o_custkey']]

l = l[l['l_returnflag'] == 'R']
l['revenue'] = l['l_extendedprice'] * (1 - l['l_discount'])
l = l[['l_orderkey', 'revenue']]
n = tbs['nation']
n = n[['n_name', 'n_nationkey']]

t = c.merge(o, left_on='c_custkey', right_on='o_custkey')
t = t.merge(l, left_on='o_orderkey', right_on='l_orderkey')
t = t.merge(n, left_on='c_nationkey', right_on='n_nationkey')
t = t[['c_custkey', 'c_name', 'c_acctbal', 'c_phone', 'n_name', 'c_address', 'c_comment', 'revenue']]
t = t.groupby(['c_custkey', 'c_name', 'c_acctbal', 'c_phone', 'n_name', 'c_address', 'c_comment'])
    .agg({'revenue': 'sum'})
t.sort_values('revenue', ascending=False)
```

Listing A.5: Query 10

```
c = c[['c_custkey']]
o = o[o['o_comment'].str.contains('^.*special.*requests.*$', regex=True)]
o = o[['o_orderkey', 'o_custkey']]
t = c.merge(o, left_on='c_custkey', right_on='o_custkey', how='left')
t = t[['c_custkey', 'o_orderkey']]
t = t.groupby('c_custkey').count()
t.reset_index(inplace=True)
t = t.groupby('o_orderkey').count()
t.reset_index(inplace=True)
t.sort_values(['c_custkey', 'o_orderkey'], ascending=[False, False])
```

Listing A.6: Query 13

```
c = c[['c_custkey']]
o = o[o['o_comment'].str.contains('^.*special.*requests.*$', regex=True)]
o = o[['o_orderkey', 'o_custkey']]
t = c.merge(o, left_on='c_custkey', right_on='o_custkey', how='left')
t = t[['c_custkey', 'o_orderkey']]
t = t.groupby('c_custkey').count()
t.reset_index(inplace=True)
t = t.groupby('o_orderkey').count()
t.reset_index(inplace=True)
t.sort_values(['c_custkey', 'o_orderkey'], ascending=[False, False])
```

Listing A.7: Query 14

```
c = c[['c_custkey', 'c_name']]
o = o[['o_orderkey', 'o_orderdate', 'o_totalprice', 'o_custkey']]
l = l[['l_orderkey', 'l_quantity']]

ti = l[['l_orderkey', 'l_quantity']]
ti = ti.groupby('l_orderkey').sum()
ti = ti[ti['l_quantity'] > 300]
ti.reset_index(inplace=True)

t = c.merge(o, left_on='c_custkey', right_on='o_custkey')
t = t.merge(l, left_on='o_orderkey', right_on='l_orderkey')
t = t.merge(ti['l_orderkey'], left_on='o_orderkey', right_on='l_orderkey')

t = t[['c_name', 'c_custkey', 'o_orderkey', 'o_orderdate', 'o_totalprice', 'l_quantity']]
t = t.groupby(['c_name', 'c_custkey', 'o_orderkey', 'o_orderdate', 'o_totalprice']).sum()
t.reset_index(inplace=True)
t.sort_values(['o_totalprice', 'o_orderdate'], ascending=[False, True])
```

Listing A.8: Query 18