DaskDB: Scalable Data Science with Unified Data Analytics and In Situ Query Processing

Alex Watson[§], Suvam Kumar Das[§], Suprio Ray
University of New Brunswick, Fredericton, Canada
Email: {awatson, suvam.das, sray}@unb.ca

***Abstract*—Due to the rapidly rising data volume, there is a need to analyze this data efficiently and produce results quickly. However, data scientists today need to use different systems, since presently relational databases are primarily used for SQL querying and data science frameworks for complex data analysis. This may incur significant movement of data across multiple systems, which is expensive. Furthermore, with relational databases, the data must be completely loaded into the database before performing any analysis.**

We believe that data scientists would prefer to use a single system to perform both data analysis tasks and SQL querying, without requiring data movement between different systems. Ideally, this system would offer adequate performance, scalability, built-in data analysis functionalities, and usability. We present DaskDB, a scalable data science system with support for unified data analytics and *in situ* SQL query processing on heterogeneous data sources. DaskDB supports invoking Python APIs as User-Defined Functions (UDF). So, it can be easily integrated with most existing Python data science applications. Moreover, we introduce a distributed index join algorithm and a novel distributed learned index to improve join performance. Our experimental evaluation involve the TPC-H benchmark and a custom UDF benchmark, which we developed, for data analytics. And, we demonstrate that DaskDB significantly outperforms PySpark and Hive/Hivemall.

I. Introduction

Due to the increasing level of digitalization in our modern society, large volumes of data are constantly being generated. To make sense of the deluge of data, it must be cleaned, transformed and analyzed. Data science offers tools and techniques to manipulate data in order to extract actionable insights from data. These include support for data wrangling, statistical analysis and machine learning model building. Traditionally, practitioners and researchers make a distinction between query processing and data analysis tasks. Whereas relational database systems (henceforth, databases or DBMSs) are used for SQL-style query processing, a separate category of frameworks are used for data analyses that include statistical and machine learning tasks. Currently, Python has emerged as the most popular language-based framework, for its rich ecosystem of data analysis libraries, such as Pandas, Numpy, scikit-learn. These tools make it possible to perform *in situ* analysis of data that is stored outside of any database, particularly as raw files (csv, txt, json, xml) or other formats such as Excel (xls). However, a significant amount of data is still stored in databases. To do analysis on this data, it must be moved from a database into the address space of the data analysis application that is written in Python (for example). Similarly, to do SQL query processing on data that is stored in a raw file, it must be loaded into a database using a loading mechanism, which is known as ETL (extract, transform, load). This movement of data and loading of data are both time consuming operations.

[§]Equal contribution

To address the movement of data across databases and data analysis frameworks, recently researchers have proposed several approaches. Among them, a few are in-database solutions, that incorporate data analysis functionalities within an existing database. These include PostgreSQL/Madlib [1] and AIDA [2]. In these systems, the application developers write SQL code and invoke data analysis functionalities through user-defined functions (UDF). There are several issues with these approaches. **First**, the vast body of existing data science applications that are written in a popular language (Python or R), need to be converted into SQL. **Second**, the data analysis features supported by databases are usually through UDF functions, which are not as rich as that of the language-based API ecosystem, such as in Python or R. **Third**, data stored in raw files needs to be loaded into a database through ETL. Although, some support for executing SQL queries on raw files exist, such as PostgreSQL's support for foreign data wrapper, this can easily break if the file is not well-formatted. In recent years several projects [3], [4], [5] investigated how to support *in situ* SQL querying on raw data files. However, they primarily focused on supporting database-like query processing, operating on a single machine. These systems lack sophisticated data wrangling and data science features that is available in Python or R. **Fourth**, most relational databases are not horizontally scalable. Even with parallel databases, the parallel execution of UDFs is either not supported or not efficient. "Big Data" systems such as Spark [6] and Hive/Hivemall [7] address some of these issues, however, they also have some drawbacks. A key challenge with these approaches is that they often involve more complex APIs and a steeper learning curve. Also, it is not practical to rewrite the large body of existing data science code with these APIs written in Python (or R, for that matter).

To address the issues with the existing approaches, we introduce a scalable data science system, DaskDB, which seamlessly supports *in situ* SQL query execution and data analysis using Python. DaskDB extends the scalable data analytics framework Dask [8] that can scale to more than one machine. Dask's high-level collections APIs mimic many of the popular Python data analytics library APIs based on

Pandas and NumPy. So, existing applications written using Pandas collections need not be modified. On the other hand, Dask does not support SQL query processing. In contrast, DaskDB can execute SQL queries *in situ* without requiring the expensive ETL step and movement of data from raw files into a database system. Furthermore, with DaskDB, SQL queries can have UDFs that directly invoke Python data science APIs. This provides a powerful mechanism of mixing SQL with Python and enables data scientists to take advantage of the rich data science libraries with the convenience of SQL. Thus, DaskDB unifies query processing and analytics in a scalable manner.

A key issue with distributed query processing and data analytics is the movement of data across nodes, which can significantly impact the performance. We propose a *distributed learned index*, to improve the performance of join that is an important data operation. In DaskDB, a relation (or dataframe) is split into multiple partitions, where each partition consists of numerous tuples of that relation. These partitions are distributed across different nodes. While processing a join, it is possible that not all partitions of a relation contribute to the final result when two relations are joined. The *distributed learned index* is designed to efficiently consider only those partitions that contain the required data in *constant* time, by identifying the data pattern in each partition. This minimizes the unnecessary data movement across nodes. Our *distributed partition-wise index join* uses the learned index to answer join queries, if one of the join column is sorted. DaskDB also incorporates intermediate data persistence and distributed in-memory data caching that significantly reduces serialization/de-serialization overhead and data movement.

We conduct extensive experimental evaluation to compare the performance of DaskDB against two horizontally scalable systems: PySpark and Hive/Hivemall. Our experiments involve workloads from a few queries from TPC-H [9] benchmark, with different data sizes (scale factors). We also created a custom UDF benchmark to evaluate DaskDB and PySpark. Our results show that DaskDB outperforms others in both of these benchmarks. For instance, DaskDB's performance was $5\times$ better than that of PySpark with TPC-H benchmark at scale factor 20 for Q5. For UDF evaluation, while computing K-Means clustering on dataset of SF 20, PySpark took too long to measure, whereas DaskDB took only 41 seconds. We also developed a microbenchmark and evaluate the effects of the proposed features on the overall performance of DaskDB.

The key contributions of this paper are:

- We propose DaskDB that integrates *in situ* query processing and data analytics in a scalable manner.
- DaskDB supports SQL queries with UDFs that can directly invoke Python data science APIs.
- We introduce a novel distributed learned index and a distributed index join algorithm that utilizes this.
- We present a few optimizations, including distributed in-memory data caching and intermediate data persistence.
- We present extensive experimental results involving TPC-H benchmark and a custom UDF benchmark.

II. Related Work

In this section, first we discuss about systems to perform data analytics and query processing. Next, we look at works related to learned index, followed by in situ query processing.

A. Data Analytics and Query Processing

First, we discuss about dedicated systems that perform data analytics. Next, we look at in-database analytics systems and then integration of data analysis and query processing.

1) Dedicated Data Analytics Frameworks: Some popular commercial data analytic systems include Tableau and MATLAB. Many open-source data analytic applications traditionally use R. More recently, Python has become very popular because of the Anaconda distribution [10]. It contains many data science and analytics packages, such as pandas, SciPy, matplotlib, and scikit-learn. They are heavily used by data scientists for data analysis.

2) In-Database Analytics: An increasing number of the major DBMSs now include data science and machine learning tools. For instance, PostgreSQL supports SQL-based algorithms for machine learning, data mining, and statistics with the Apache MADlib library [1]. However, interacting with a DBMS to implement analytics can be challenging [11]. The end user requires the knowledge of database specific language, such as SQL and stored procedure languages, which is DBMS specific (e.g., PL/pgSQL, T-SQL or PL/SQL). Although SQL is a mature technology, it is not rich enough for extensive data analysis. DBMSs typically support analytics functionalities through User Defined Functions (UDF). Since, a UDF may execute any external code written in R, Python, C++, Java or T-SQL, a DBMS usually treats a UDF as a black box because no optimization can be performed on it. It is also difficult to debug and to incrementally develop [2]. The in-database analytics approaches still have the constraint of ETL, which is a time-consuming process and not practical in many cases.

3) Integrating Analytics and Query Processing: There have been several attempts at creating more efficient solutions and they combine two or more of either dedicated data analytic systems, DBMS or big data frameworks. These systems can be classified into 2 categories that we describe next.

Hybrid Solutions. These solutions integrate two or more system types together into one and are primarily DBMS-centric approaches. AIDA [2] integrates a Python client directly to use the DBMS memory space, eliminating the bottleneck of transferring data. In [12], the authors proposed an embeddable analytical database DuckDB. The key drawback of these hybrid systems is ETL, since the data needs to be loaded into a database. Moreover, existing data science applications written in Python or R, need to be modified to work in such systems, since their interface is SQL-based.

"Big Data" Analytics Frameworks. The most popular big data frameworks are Hadoop [13] and Spark [6]. Spark supports machine learning with MLlib [14] and SQL like queries. Hive is based on Hadoop that supports SQL-like queries and supports analytics with the machine learning library Hivemall [7]. Some drawbacks of big data frameworks include more complicated development and steeper learning curve than most other analytics systems and the difficulty in integration with DBMS applications. To run any existing Python or R application within a big data system, it requires rewriting with new APIs, which is not the most viable option.

B. Learned Index

Data structures such as B+trees are the mainstay of indexing techniques. These approaches require the storage of all keys for a dataset. Recent studies have shown that learned models can be used to model the cumulative distribution function (CDF) of the keys in a sorted array. This can be used to predict their locations for the purpose of indexing and this idea was termed as learned index [15]. Subsequently, several learned indexes were proposed, such as [16], [17], [18]. However, these approaches were meant only for stand-alone systems. These ideas have not been incorporated as part of any database system yet, to the best of our knowledge. Also, no learned index has yet been developed for any distributed data system.

C. In Situ Query Processing

A vast amount of data is stored in raw file-formats that are not inside traditional databases. Data scientists, who frequently lack expertise in data modeling, database admin and ETL tools, often need to run interactive analysis on this data. To reduce the "time to query" and avoid the overhead associated with relational databases, a number of research projects investigated *in situ* query processing on raw data.

NoDB [3] was one of the earliest systems to support in situ query processing on raw data files. It utilizes a positional map data structure to ameliorate the cost of tokenizing and processing raw data. PostgresRaw [19] is based on the idea of NoDB and it supports SQL querying over CSV files in PostgreSQL. The SCANRAW [4] system exploits parallelism during in situ raw data processing. All these systems were focused on database-style SQL query processing on raw data and on **a single machine**. Our system, DaskDB supports *in situ* querying on heterogeneous data sources, and it also supports doing data science. Moreover, it is a distributed data system that can scale over a cluster of machines.

III. Dask Background

DaskDB was developed by extending Dask [8], an open-source library for distributed computing in Python. The main advantage of Dask is that it provides Python APIs and data structures that are similar to NumPy, pandas, and scikit-learn. Hence, programs written using Python data science APIs can easily be switched to Dask by changing the import statement.

Dask supports various collections (Arrays, Dataframes, etc.) and task execution primitives such as *Futures* and *Delayed*. The collections interfaces support scalable version of the APIs popularized by NumPy and pandas libraries.

The *dask.distributed* [20] library is responsible for distributed computation based on *Task Graph* in a cluster. The framework consists of a server, several clients and workers, and it comes with an efficient task scheduler. Dask is a quite scalable framework, as shown by previous research [21].

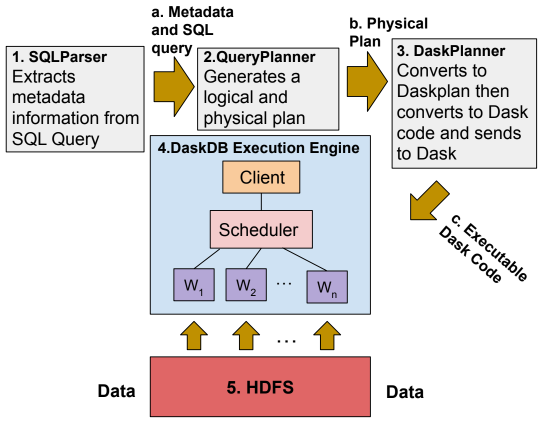

a. Metadata and SQL query
b. Physical Plan
1. SQLParser Extracts metadata information from SQL Query
2.QueryPlanner Generates a logical and physical plan
3. DaskPlanner Converts to Daskplan then converts to Dask code and sends to Dask
4.DaskDB Execution Engine
Client
Scheduler
W1
W2
...
Wn
c. Executable Dask Code
Data
5. HDFS
Data

Fig. 1: DaskDB System Architecture

IV. Our Approach: DaskDB

In this section, we present DaskDB. It addresses some of the issues discussed in the related work section. DaskDB was designed with the vision of a scalable data science system that supports data analytics and *in situ* query processing within a single system, without requiring any developer effort to convert the vast body of application code that was written in Python using its data science APIs. DaskDB, in addition to supporting all Dask features, also enables *in situ* SQL querying on raw data in a data science friendly environment. Next, we describe DaskDB system architecture and its components.

A. System Architecture

The system architecture of DaskDB incorporates five main components: the SQLParser, QueryPlanner, DaskPlanner, DaskDB Execution Engine, and HDFS. They are shown in Figure 1. First, the SQLParser gathers metadata information pertaining to the SQL query, such as the names of the tables, columns and functions. This information is then passed along to the QueryPlanner. Next, in the QueryPlanner component, physical plan is generated from the information sent by SQL-Parser about the SQL query. The physical plan is an ordered set of steps that specify a particular execution plan for a query and how data would be accessed. The QueryPlanner then sends the physical plan to the DaskPlanner. In the DaskPlanner, a plan is generated, which includes operations that closely resemble Dask APIs, called the *Daskplan*. The Daskplan is produced from the physical plan, and it is then converted into Python code and sent to DaskDB Execution Engine. DaskDB Execution Engine then executes the code and gathers the data from the HDFS, and thus executes the SQL query. Further details are provided in the next sections.

1) SQLParser: The SQLParser is the first component of DaskDB that is involved in query processing. The input for the SQLParser is the original SQL query. It first checks for syntax errors and creates a parse tree with all the metadata information about the query. We then process the parse tree to gather the metadata information needed by the QueryPlanner. This metadata information includes table names, column names and UDFs. We then check if the table(s) exist, (for example, if there is a *.csv* file with that name) in the default directory.

Algorithm 1: DaskPlanner: Conversion of Physical Plan to Daskplan

```
Input: A physical plan (P) containing ordered groups of
       dependent operators (G). Each G consists of an
       ordered list of tuples (k, o, d), where k is an unique
       ID corresponding to each operation, o is the
       operation type and d contains the operation metadata
       information.
Output: The final result is a Daskplan DP, which consists
        of an ordered list of operators.
DP ← list()
for sorted($G \in P$) do
    for sorted($k, o, d \in G$) do
        dp ← dict() //create Daskplan operator
        dp[o] ← convertToDaskPlanOperator(o)
        dp[d] ← getMetadataInfo(d)
        dp[key] ← k //adds key (used to get data
          dependencies)
        DP.add(dp) //adds dp to Daskplan
//Each operation (other than table scan) needs intermediate
  results (table) from previous operations.
for sorted($dp \in DP$) do
    while dp has children (c) do
        dp[$t_i$] ← get table information from $c_i$
return DP
```

Algorithm 2: Conversion of Daskplan to Executable Python Code

```
Input: Daskplan DP
Output: Executable Code corresponding to DP
Procedure getExecutableCode(DP):
    for all $dp \in DP$ do
        operationType ← dp[o]
        metadata ← dp[d]
        if operationType == "read_csv" then
            table ← metadata.getTable1()
            EMIT("table = read_csv($table)") //$ will
              replace the variable with its value
        else if operationType == "Filter" then
            table ← metadata.getTable1()
            value ← metadata.getvalue()
            compType ← metadata.getCompType() // $\geq$, $\leq$,
              $\neq$, etc.
            EMIT("table = $table.filter($value,
              $compType)")
        else if operationType == "Join" then
            table1 ← metadata.getTable1()
            table2 ← metadata.getTable2()
            col1 ← metadata.getJoinCol1()
            col2 ← metadata.getJoinCol2()
            EMIT("Temp = $table1.join($table2, $col1,
              $col2)")
        else if ... then
            //the other cases are not shown due to space
              constraints
```

If the table exists, we dynamically generate a schema. The schema contains information about tables and column names and data types used in the SQL query. The schema, UDFs (if any) and the original SQL query are then passed to the QueryPlanner.

DaskDB can treat any file (with a supported file format) as a data table and hence DaskDB supports *in situ* heterogeneous data source querying [22]. In contrast, a DBMS would require an ETL process to load data into its native storage.

2) QueryPlanner: The QueryPlanner creates logical and preliminary physical plans. The schema and UDFs produced by SQLParser, along with the SQL query, are passed into the QueryPlanner. The QueryPlanner uses these to first create a logical plan and then an optimized preliminary physical plan. This plan is then sent to the DaskPlanner.

3) DaskPlanner: The DaskPlanner is used to transform the preliminary physical query plan from the QueryPlanner into Python code that is ready for execution. The first step in this process is for the DaskPlanner to go through the physical plan obtained from QueryPlanner and convert it into a Daskplan. This maps the operators from the physical plan into operators that more closely resemble the Dask API. This Daskplan also associates relevant information with each operator from the physical plan. This information includes columns and tables involved and specific metadata information for a particular operator. We also keep track of each operator's data dependency. This is needed to pass intermediate results from one operation to the next. Algorithm 1 shows how DaskDB converts the physical plan into the Daskplan.

In the next step, the DaskPlanner converts the Daskplan into the Python code, which utilizes the Dask API. All of the detail about each table, their particular column names and indexes are maintained in a dynamic dictionary throughout the query execution. This is because multiple tables and columns may be created, removed or manipulated during a query execution. For example, columns often become unnecessary after a particular filter or join operation is executed. These columns are dropped for optimization to avoid unneeded data movement. For these reasons, the names of the tables, columns, and indexes are dynamically maintained while transforming the Daskplan into Python code. Algorithm 2 shows how Daskplan is converted into executable Python code.

4) DaskDB Execution Engine: There are three main components of the DaskDB execution engine: the client, scheduler and workers. The client transforms the Dask Python code into a set of tasks. The scheduler creates a DAG (directed acyclic graph) from the set of tasks, automatically partitions the data into chunks, while taking into account data dependencies. The scheduler sends a task at a time to each of the workers according to several scheduling policies. The scheduling policies for task and workers depend on various factors including data locality. A worker stores a data chunk until it is not needed anymore and is instructed by the scheduler to release it.

5) HDFS: The Hadoop Distributed File System (HDFS) is a storage system used by Hadoop applications. HDFS provides high-performance and access to data across highly scalable Hadoop clusters. DaskDB uses HDFS to store and share the data files among its nodes.

B. Illustration of SQL query execution

An *in situ* query is executed within DaskDB by calling `query` function with the SQL string as argument. The query in Figure 2 is a simplified version of a typical TPC-H query.

The Daskplan, shown in Figure 3, is generated from the physical plan in the DaskPlanner component. The Daskplan

```
from daskdb_core import query

sql = """SELECT l_orderkey, sum(l_extendedprice *
    (1-l_discount)) as revenue
  FROM orders, lineitem
  WHERE l_orderkey = o_orderkey and
        o_orderdate >= '1995-01-01'
  GROUP BY l_orderkey
  ORDER BY revenue LIMIT 5 ; """
query(sql)
```

Fig. 2: Code showing SQL query execution in DaskDB

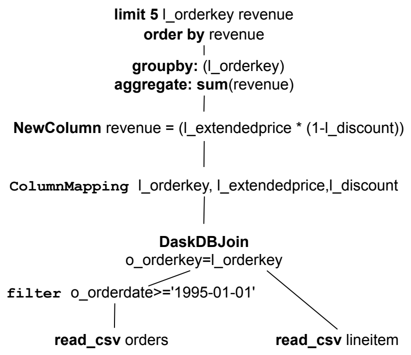

limit 5 l_orderkey revenue
order by revenue
groupby: (l_orderkey)
aggregate: **sum**(revenue)
NewColumn revenue = (l_extendedprice * (1-l_discount))
ColumnMapping l_orderkey, l_extendedprice,l_discount
DaskDBJoin
o_orderkey=l_orderkey
filter o_orderdate>='1995-01-01'
read_csv orders
read_csv lineitem

Fig. 3: Generated Daskplan for the code in Figure 2

operators more closely resemble the Dask API. For example, these include the `read_csv` and `filter` methods shown in the tree. This Daskplan is then converted into executable Python code, which is omitted due to space constraint.

C. Support for SQL query with UDFs

DaskDB supports UDFs in SQL as part of *in situ* querying. A UDF enables a user to create a function using Python code and embed it into the SQL query. Since DaskDB converts the SQL query and UDF back into Python code, the UDFs can reference and utilize features from any of the existing data science packages from Anaconda Python. Spark introduced UDF's in SQL queries since version 0.7, which operated one-row-at-a-time, and thus suffered from high serialization and invocation overhead. To address these issue, Spark came up with *Pandas UDF* since version 2.3, which provides low-overhead, high-performance UDFs entirely in Python. But these are restrictive to use as it also sometimes require to use Spark's own data types, which would be inconvenient for users who are not experienced in Spark. In contrast, in DaskDB UDFs for SQL queries can easily be written. Any native Python function (either imported from an existing package or custom-made), which accepts Pandas dataframes as parameters can be applied as UDFs to the SQL queries in DaskDB. The return type of the UDFs is not fixed like Spark's Pandas UDF, and hence allows the user to design UDFs with ease. Like a general Python function, UDFs with code involving machine learning, data visualization and numerous other functionalities can easily be developed and applied on queries in DaskDB.

D. Illustration of SQL query with UDF

In this section, we illustrate two examples of DaskDB using UDFs in SQL queries: **K-Means Clustering** and **Conjugate Gradient Optimization**. The UDFs are invoked in the same way as it would be in a typical DBMS system. Similar to Spark, the UDFs need to be registered to DaskDB system using the `register_udf` API. It takes as parameters a Python function and a list of numbers. Suppose a Python function `func` is used as a DaskDB UDF, which takes 3 pandas dataframes as parameters, where the panda dataframes consists of 2, 5 and 1 columns respectively, then `func` is registered to DaskDB as `register_udf(func, [2,5,1])`.

```
from daskdb_core import query, register_udf
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

def myKMeans(df):
    kmeans = KMeans(n_clusters=4).fit(df)
    col1 = list(df.columns)[0]
    col2 = list(df.columns)[1]
    plt.scatter(df[col1], df[col2],
        c= kmeans.labels_.astype(float), s=50)
    plt.xlabel(col1)
    plt.ylabel(col2)
    plt.show()

register_udf(myKMeans,[2])
sql_kmeans = """select myKMeans(l_discount, l_tax)
 from lineitem where l_orderkey < 50 limit 50; """
query(sql_kmeans)
```

Fig. 4: UDF code showing K-Means Clustering

K-Means Clustering. As shown in Figure 4, the UDF `myKMeans` takes as input a single pandas dataframe having 2 columns; hence the UDF is registered as `register_udf(myKMeans,[2])`. This UDF divides the data points into 4 clusters using the KMeans API (from scikit-learn package) and plots them graphically using the matplotlib package. The UDF in this query is invoked as `myKMeans(l_discount, l_tax)`, which means after application of the selection condition (l_orderkey < 50) and the limit (limit 50) to the lineitem relation, both the columns l_discount and l_tax together form a pandas dataframe and is passed to `myKMeans`.

Conjugate Gradient Optimization. Given a mathematical expression $2u^2 + 3uv + 7v^2 + 8u + 9v + 10$, an UDF `myConjugateGradOpt` is designed to minimize the expression using the Conjugate Gradient Optimization Technique. The initial values of u and v are passed to the UDF as two pandas dataframes. To solve this, the `optimize` module of SciPy package is used in the UDF. The code is shown in Fig. 5.

E. Distributed Learned Index

We propose a novel distributed learned index that can be used to improve the performance of a join, which involves combining two relations (dataframes). In DaskDB, a relation is constructed as a Dask dataframe by loading data from raw data file(s). It may consist of many partitions, where each partition stores a number of tuples of the relation. Within each partition, the tuples can be sorted based on a natural order (i.e., by the primary key of a relation). Our distributed learned index can be conceptualized as a distributed clustered index, and its purpose is to quickly locate the partition id of a search key.

A learned index typically has a learned model that is trained and then utilized to determine the position of a search key in a sorted in-memory array. The learned model is usually based on

```
from daskdb_core import query, register_udf
from scipy import optimize
import numpy as np

def myConjugateGradOpt(df):
    def f(x, *args):
        u, v = x
        a, b, c, d, e, f = args
        return a*u**2 + b*u*v + c*v**2 +\
            d*u + e*v + f

    def gradf(x, *args):
        u, v = x
        a, b, c, d, e, f = args
        gu = 2*a*u + b*v + d #u-component of gradient
        gv = b*u + 2*c*v + e #v-component of gradient
        return np.asarray((gu, gv))

    args = (2, 3, 7, 8, 9, 10)  # parameter values
    x0 = df
    val = optimize.fmin_cg(f, x0, fprime=gradf,\
        args=args)
    return val

register_udf(myConjugateGradOpt, [2])
sql_cgo ="""select myConjugateGradOpt(l_discount,
l_tax) from lineitem where l_orderkey<10 limit 1;"""
res = query(sql_cgo)
```

Fig. 5: UDF code showing Conjugate Gradient Optimization

a machine learning approach [15] and hence the predicted key position may involve some uncertainties. So, a local search may be necessary to rectify this. Our proposed distributed learned index takes a search key as input and determines the id of the data partition where the key is located. Our learned model can be considered as an interpolation based approach, such as [18] and is based on Heaviside step function [23]. It can accurately identify the partition id for a search key, which can avoid a local search. Next we describe our learned model.

A step function can be represented by a combination of multiple Heaviside unit step functions, which is the basis of our learned model. A Heaviside unit step function is defined:

$$H(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases} \tag{1}$$

The Heaviside unit step function is a binary function, which only identifies whether a value is negative or non-negative. To serve our purpose, we needed a function which if given a value x, could identify a predefined boundary (a,b), such that $x \in [a, b]$. Thus, we define a parameterized *Partition Function, F* using Heaviside functions as

$$F_{a,b,c}(y) = H((b-y)*(y-a))*c \mid a, b \in \mathbb{R}, c \in \mathbb{Z}^+ \tag{2}$$

which returns c whenever $a \leq y \leq b$, or returns 0 otherwise.

While constructing the distributed learned index, it is assumed that one of the relations is sorted by the join attribute. Hence, if a *partition table* is built on this column, which can maintain the first and last values of the keys for each partition, then given any key, the partition containing the key can be identified by a *Partition Function*.

Let there be a relation A with n partitions, sorted on a column. If a *partition table* is built for this relation on the sorted column, then it will also consist of n entries i.e. there will be an entry for each partition. Each entry stores the **begin** and **end** values of the sorted column for each partition. For each partition p_i, let the partition begins with a value b_i and ends with e_i. Then we can construct a *Learned Model Function*

Algorithm 3: Construct Learned Model

```
Input: Relations $A$ sorted on $col_A$.
Output: Learned Model Function $L_A$ on $A$
1  Procedure getLearnedModel($A$):
2      $S \leftarrow$ GeneratePartitionTable($A$)
3      $L_A \leftarrow$ NULL
4      foreach entry $E$ in $S$ do
5          $a \leftarrow E.Begin$
6          $b \leftarrow E.End$
7          $c \leftarrow E.Partition$
8          $F_{a,b,c}(y) \leftarrow H((b-y)*(y-a))*c$
9          $L_A \leftarrow L_A + F_{a,b,c}$
10     return $L_A$
```

L_A on relation A as :

$$L_A(y) = \sum_{i=1}^{n} F_{b_i,e_i,p_i}(y) \tag{3}$$

We illustrate this using a simplified example with the **customer** table from TPC-H, where *c_custkey* is the primary key. If there are 500 tuples in this relation and each table partition can store 100 tuples, then there will be total 5 partitions. The distribution of the keys is shown in Table I, and also plotted in Figure 7.

Begin	End	Partition
1	200	1
250	380	2
400	560	3
580	700	4
701	800	5

TABLE I: Partition table for **customer** relation

$$f(key) = \begin{cases} 1 & 1 \leq key \leq 200 \\ 2 & 250 \leq key \leq 380 \\ 3 & 400 \leq key \leq 560 \\ 4 & 580 \leq key \leq 700 \\ 5 & 701 \leq key \leq 800 \end{cases} \tag{4}$$

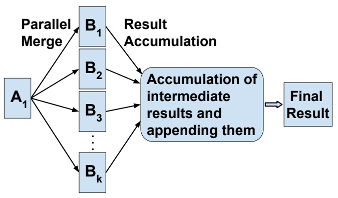
It can be seen that the plot is a step function f in Equation 4. which can equivalently be represented by summing several *Partition Functions*, which constitutes the *Learned Model Function* $L_{customer}$ on the **customer** table as

$$L_{customer}(key) = F_{1,200,1}(key) + F_{250,380,2}(key) + F_{400,560,3}(key) + F_{580,700,4}(key) + F_{701,800,5}(key)$$

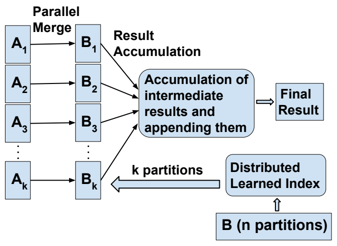
where, *a, b* and *c* of the *Partition Function F* represent the begin and end keys and the corresponding partition ids. Algorithm 3 shows this process in details. The idea may seem similar to a zone map [24], which maintains min/max value ranges over the entries of a table column situated within each partition of the table (i.e. zone). However, in a zone map, all the zones are traversed iteratively until the zone containing the required key is found. In our case, the *Learned Model Function* directly provides the partition id, since it is the summation of several *Partition Functions*. This is because given a key value, only one of the Partition Functions of the learned model will return the partition id, whereas the other ones return 0. So, given a key, we can find the corresponding partition id to which that key belongs.

F. Joining of Relations

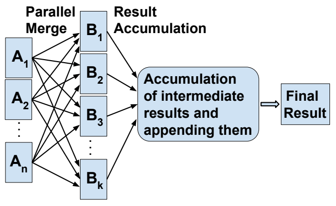
Join is considered as one of the most expensive data operations. Given two relations (dataframes) A and B, each



(a) Distributed Single-Partition Join



(b) Distributed Partition-wise Index Join



(c) Distributed Multi-Partition Join

Fig. 6: DaskDB Join Algorithms

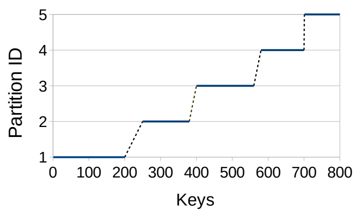


Fig. 7: Keys vs partition id

```
Algorithm 4: DaskDB join
  Input: Relations A and B where joining columns are $col_A$ and $col_B$.
  Output: Joined relation $M = A \bowtie B$
  Procedure join($A$, $B$):
      if $A.npartition$==1 OR $B.npartition$==1 then
          $M \leftarrow$ SinglePartitionJoin($A$, $B$)
      else if $col_A$ is sorted then
          $M \leftarrow$ PartitionwiseIndexJoin($A$, $B$)
      else if $col_B$ is sorted then
          $M \leftarrow$ PartitionwiseIndexJoin($B$, $A$)
      else
          $M \leftarrow$ MultiPartitionJoin($A$, $B$)
      return $M$
```

consisting of many partitions, our join algorithm (Algorithm 4) applies the following criteria:

- If any one of the relations is of single partition, we use a *Distributed Single-Partition Join* (line 3 in Algorithm 4), where the single partition of one relation is joined with all partitions of the other relation.
- If one of the joining columns is sorted, then we join them using a Distributed Partition-wise Index Join (lines 4 to 7) with our novel *Distributed Learned Index*.
- Otherwise we employ a *Distributed Multi-Partition Join*, where all partition of one relation is joined with all partitions of the other relation (line 9).

The details of each join algorithm (illustrated in Fig. 6) are described next.

1) Distributed Single-Partition Join: This is used if any one of the joining relations is of single partition. If A has a single partition, the distributed query scheduler can distribute A to those worker nodes where partitions of B are present. Then A is merged in parallel with all the individual partitions of B. Finally, the result of all this individual merge operations are combined by the scheduler to get the final result.

2) Distributed Partition-wise Index Join: This algorithm (Algorithm 5) is used when one of the joining columns is sorted. In this case we use our *distributed learned index* for partition identification and join. Let A and B be two relations, which need to be joined on col_A and col_B of A and B respectively. Without loss of generality, we assume col_A is sorted and a learned model $L_A(k)$ is constructed on this column. Since DaskDB internally uses Dask APIs, each relation corresponds to a Dask dataframe consisting of one or

```
Algorithm 5: Partition-wise Index Join
  Input: Relations A and B where joining columns are $col_A$ and $col_B$. A is sorted on $col_A$.
  Output: Joined relation $M = A \bowtie B$
  Procedure PartitionwiseIndexJoin($A$, $B$):
      // $L_A$ is the learned model built on $col_A$
      if $L_A$ is NULL then
          $L_A \leftarrow$ getLearnedModel($A$)
      foreach Partition $B_i$ of B do in parallel
          $data_i \leftarrow B_i[col_B]$ //fetch $col_B$ of the partition
          $p_i \leftarrow L_A(data_i)$ //get partition id of A for each element of $data_i$
          $B_i['Partition'] \leftarrow p_i$ //add $p_i$ as a column to the partition
      $B.repartition('Partition')$ //repartition B so that all records with the same $'Partition'$ are together
      $B.delete('Partition')$ //delete the $'Partition'$ column from B
      foreach Partition $B_i$ of B do in parallel
          $M_i \leftarrow$ merge($A_i$, $B_i$) //partitionwise merge
      M ← Combine all $M_i$
      return M
```

more partitions. For each partition B_i of B the following steps are performed in parallel (lines 5 - 8):

- For each entry k of col_B of B_i, $L_A(k)$ is determined, which is the partition id of A to which k belongs.
- Append this list of partition ids to B_i as a new $'Partition'$ column.

Then, B is repartitioned (line 9) based on the $'Partition'$ column such that all the tuples within the same $'Partition'$ value are together in the same partition of B. After this, the $'Partition'$ column is dropped from B (line 10). The number of partitions of both A and B are now equal. This allows us to uniquely identify the partition pairs of (A_i, B_i) which needs to be merged. Finally all the intermediate merged results are accumulated together to form the final result (lines 11 - 13). This algorithm is also illustrated in Fig. 6b.

3) Distributed Multi-Partition Join: This algorithm is used when none of the two previously mentioned algorithms can be applied. The idea is similar to a block nested loop join, as each data partition can be considered as a block. The key difference is that instead of iterating through all the blocks of one relation in a loop and merging it with all the blocks of another relation, the entire operation is performed in parallel. Similar to *Distributed Single-Partition Join*, the query scheduler is responsible for distributing the necessary partitions among the worker nodes, where the intermediate merge results get computed and are finally accumulated to get the result. Fig 6c illustrates this process, where all n partitions of A are merged in parallel with all k partitions of B.

G. Distributed In-Memory Data Caching

In DaskDB, raw data files are stored in the HDFS. During the execution of a query/analytics task, dataframes are constructed by reading from these files. However, this incurs significant overhead due to serialization/de-serialization (S/D). It was shown that S/D may account for 30% of the execution time in Spark [25]. Also, shuffling of intermediate data across nodes may incur many S/D operations and data movement.

To address these issues, DaskDB performs distributed in-memory data caching. Raw data files are read only once at the time of system initialization, and split into partitions and are persisted in the distributed memory. Each time a task is executed, instead of reading files from HDFS, the in-memory cached data are used. Our benchmark evaluation (Section V) shows that we get 3× to 4× speed-up when data are cached.

H. Intermediate Data Persistence

DaskDB, built over the Dask framework, performs lazy computation by default. In this case, the entire computation begins in the cluster only when `compute` API of Dask is encountered at the end. Hence a task cannot begin its computation if it has data dependency on some other task. In this case the task has to wait until the previous task finishes and provides the result to it. To overcome this, Dask supports the computation of intermediate tasks as soon as they are created. This intermediate results are also persisted in the distributed memory during the execution, so that they can be immediately provided to the future tasks, if required. DaskDB utilizes this feature to speed up query executions. Computations for operations, such as merge and groupby, are initiated and persisted as soon as they are encountered. The intermediate results are later provided to future operations. For example, when joining three relations *A, B* and *C*, *A* and *B* are joined together and the result is persisted as soon as the join operation is encountered. Later, this result is joined with *C* to get the final result. In this case, the second join did not need to wait for the result of the first join, if it was ready. If we do not apply data persistence, then the second join has to wait for the first join (and all other operations prior to it) to finish and provide the intermediate result to it. Our benchmark results show that this approach of data persistence can improve query execution time, which entails a speed-up of 2× to 3×.

I. Distributed Task Scheduling

DaskDB is designed to support distributed *in situ* query execution by utilizing multiple nodes in a cluster, where each node may include many processing cores. To implement distributed query scheduling, DaskDB uses Dask's task scheduler. We explain this in the context of the example query shown in Fig. 2. DaskDB's query planner generates executable Python code for this, and the scheduler performs the required computation as per the Task Graph shown in Fig. 8.

Both relations *orders* and *lineitem* are stored in HDFS as csv files and are loaded from there. Rectangle 1 in Fig. 8 indicates that the *orders* relation is loaded. Then a filter condition is applied to column *o_orderdate*, as per the generated Daskplan shown in Fig. 3, and the column is finally dropped from the remaining tuples. A distributed learned index is constructed based on *o_orderkey*. This step does not appear in the Task Graph because this a preprocessing step performed in advance.

The rectangle 2 in Fig. 8 depicts that the relation *lineitem* is also similarly read and the partitions are brought into memory. Rectangle 3 involves the partition identification and repartitioning for *lineitem*, as described in Section IV-F2. Rectangle 4 shows the partition-wise joining of the two relations. This task is executed in parallel for each partition. A new column *revenue* is created in the rectangle 5 by multiplying the column *l_extendedprice* with (1 - *l_discount*). Rectangles 6 and 7 respectively denote the groupby on *l_orderkey* and aggregate *sum* operation on the *revenue* column. Rectangle 8 denotes the *orderby* and *limit* operations. In this case as per the query only the top 5 rows having maximum *revenue* are selected.

V. Evaluation

In this section, we present the experimental setup, TPC-H benchmark results and a custom UDF benchmark results. We also present microbenchmark results.

A. Experimental Setup

DaskDB was implemented in Python by extending Dask. The SQLParser of DaskDB utilizes the tool sql-metadata [26]. The QueryPlanner of DaskDB extends Raco [27]. We ran experiments on a cluster of 8 nodes, each running Ubuntu 16.04 OS. Each node has 16 GB memory and 2 Intel(R) Xeon(R) CPUs (with 4 cores per CPU), running at 3.00 GHz.

We evaluated DaskDB against two systems that support both SQL query execution and data analytics: PySpark and Hive/Hivemall (*henceforth referred to as Hivemall*). HDFS was used to store the datasets for each system. The software versions of Python, PySpark and Hive were 3.7.6, 3.0.1 and 2.1.0 respectively. PySpark and Hivemall were allocated maximum resources available (i.e. cores and memory).

We conducted experiments with three different benchmarks and the experimental settings are summarized in Table III. Each query/task was executed four times, with one cold run and three warm runs, and the average time taken for the three warm runs were taken into account.

B. TPC-H Benchmark Evaluation Results

We evaluated the systems with several queries from TPC-H decision support benchmark [9]. We used 4 scale factors (SF): 1, 5, 10 and 20, where SF 1 indicates roughly 1 GB.

We executed 5 queries from TPC-H benchmark and the results are plotted in Figure 10. As can be seen, DaskDB outperforms PySpark and Hivemall on all queries for all the scale factors. Hivemall performs worse than both DaskDB and

Tasks	Query
LR	select **myLinearFit(l_discount, l_tax)** from lineitem where l_orderkey < 10 limit 50
K-Means	select **myKMeans(l_discount, l_tax)** from lineitem, orders where l_orderkey = o_orderkey limit 50
Quantiles	select **myQuantile(l_discount)** from lineitem, orders where l_orderkey = o_orderkey limit 50
CGO	select **myCGO(l_discount, l_tax)** from lineitem where l_orderkey < 10 limit 1

TABLE II: UDF benchmark (queries with custom UDF)

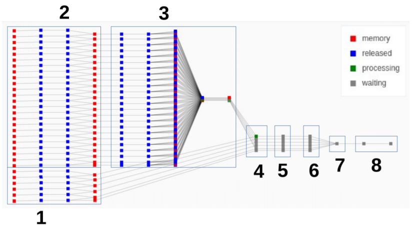
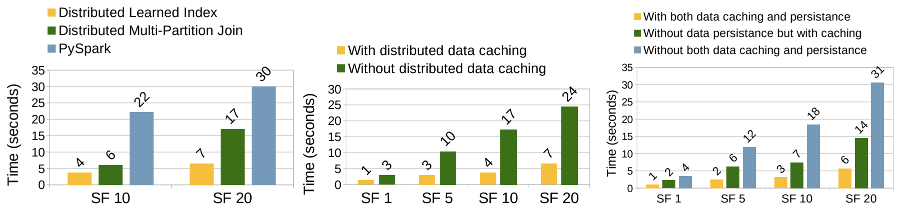


Fig. 8: Scheduler Task Graph generated for query in Figure 2

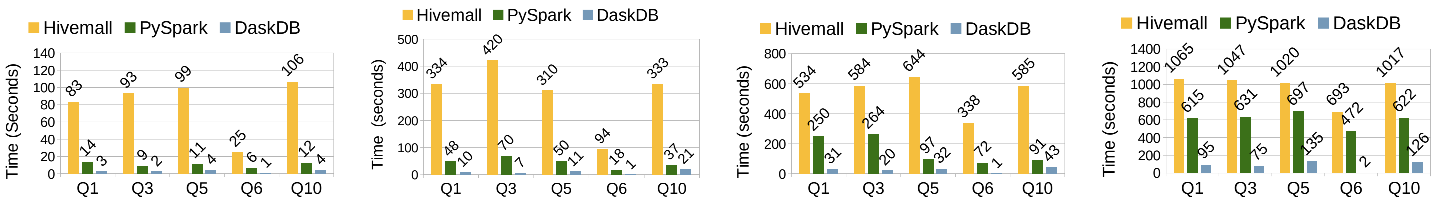
	Systems evaluated	Queries/ Tasks	Scale factors
TPC-H benchmark	DaskDB, PySpark and Hivemall	Q1, Q3, Q5, Q6 and Q10	1, 5, 10 and 20
UDF benchmark	DaskDB and PySpark	LR, K-Means, Quantiles and CGO (Table II)	1, 5, 10 and 20
Micro benchmark	DaskDB and PySpark	Custom query (Fig 12,13)	[1, 5,] 10 and 20

TABLE III: Experimental setting



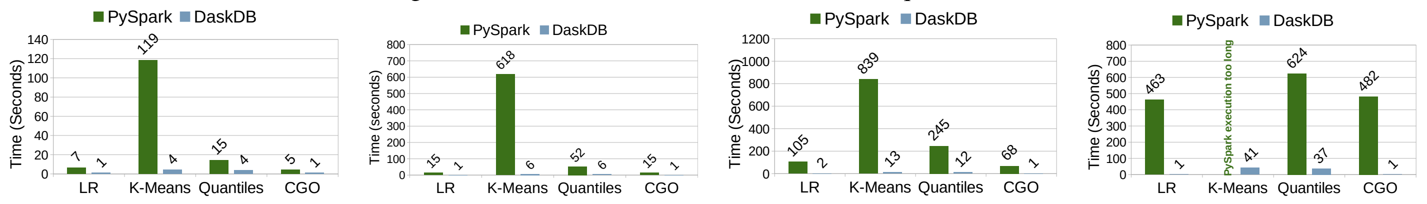
(a) Distributed learned index
(b) Distributed in-memory data caching
(c) Intermediate data persistence

Fig. 9: Microbenchmark results (DaskDB, unless PySpark is mentioned)



(a) Scale factor 1
(b) Scale factor 5
(c) Scale factor 10
(d) Scale factor 20

Fig. 10: Execution times - TPC-H benchmark queries



(a) UDF on scale factor 1
(b) UDF on scale factor 5
(c) UDF on scale factor 10
(d) UDF on scale factor 20

Fig. 11: Execution times - SQL queries with UDFs

PySpark in all cases. However, in general higher the SF, larger the performance gap between them. For instance, with Q10, DaskDB is 3× faster than PySpark at SF 1 and 5× faster than PySpark at SF 20. DaskDB achieves a speedup of 8.5× with Q3 at SF 20. The superior performance of DaskDB can be credited to its efficient join implementation using distributed learned index, data caching, and data persistence.

The results also show that Hivemall execution was time consuming in all the scale factors and for SF 20, it took over 1000 seconds on average to execute in four out of the five queries. This is the reason why we decided to stop the evaluation at SF 20 i.e. with 20GB dataset.

C. UDF Benchmark Evaluation Results

We developed a custom UDF benchmark, shown in Table II. This consists of four machine learning tasks, executed as SQL queries with UDFs: **LR** (Linear Regression), **K-Means** (K-Means Clustering), **Quantiles** (Quantiles Estimation) and **CGO** (Conjugate Gradient Optimization). They were developed using the available machine learning packages in Python. Here, DaskDB was evaluated against PySpark, whereas Hivemall results are skipped due to poor performance.

The results are plotted in Figure 11. Similar to the TPC-H benchmark results, DaskDB outperforms PySpark here as well for all the machine learning tasks for all scale factors. Among all the tasks, K-Means performs worst in PySpark. With respect to K-Means, DaskDB performs 28.5× faster than PySpark at SF 1 and 64× faster at SF 10. For SF 20, PySpark took too long to perform K-Means and hence could not be measured, whereas DaskDB took only 41s approximately. For Quantiles, DaskDB was 4× and 16.6× faster than PySpark for SF 1 and SF 20 respectively. These results also show that larger the SF, the better DaskDB performs compared to PySpark.

Here too, we stopped evaluating beyond SF 20, because

```
select n_name, sum(o_totalprice) as total_orders
from customer, orders, nation, region
where c_custkey=o_custkey and c_nationkey=n_nationkey
and n_regionkey = r_regionkey and r_name = 'AMERICA'
and o_orderdate >= date '1995-01-01'
and o_orderdate < date '1995-01-01'+interval '1' year
group by n_name order by total_orders desc limit 1;
```

Fig. 12: Microbenchmark query for learned index and distributed in-memory data caching

```
select p_name, (p_retailprice * ps_availqty)
    as total_price
from part, partsupp, supplier, nation
where p_partkey=ps_partkey and s_suppkey=ps_suppkey
and n_nationkey = s_nationkey and n_name='CANADA'
order by total_price desc limit 1;
```

Fig. 13: Microbenchmark query for data persistence

PySpark runtimes were much higher than DaskDB.

D. Microbenchmark Results

To show the effects of individual features, including, distributed learned index, in-memory data caching and intermediate data persistence on the performance of DaskDB, each of them were benchmarked separately. We developed a microbenchmark, which includes the query in Fig. 12 for learned index and distributed in-memory data caching and the query in Fig. 13 for intermediate data persistence. They were executed on TPC-H datasets with different scale factors (SF).

The effect of using distributed learned index for joining relations is shown in Fig. 9a. Here, the same query was executed in 3 scenarios: (i) DaskDB with distributed learned index, (ii) DaskDB with the default distributed multi-partition join, and (iii) PySpark. Higher TPC-H SFs were chosen in this case because higher the SF, the more advantageous it is to use a distributed learned index. As can be seen from Fig. 9a, DaskDB with distributed learned index attains a significant speed-up, for instance, a speed-up of 1.5$\times$ and 5.5$\times$ respectively, compared to DaskDB with distributed multi-partition join and PySpark on SF 10.

The advantage of distributed in-memory data caching is illustrated in Fig. 9b. In this case, the dataframes of the participating relations were cached in the distributed memory among the worker nodes in the first run (cold run) and were subsequently used in the following three runs (warm runs). The execution time was compared with the scenario where dataframes were not cached at all. To maintain consistency, distributed learned index was used in both cases, along with intermediate data persistence. We observe a speed-up of 3$\times$ to 4$\times$ with all the SFs when data caching was performed.

Intermediate data persistence also has a positive impact on query execution as shown in Fig. 9c. For this benchmarking, query of Fig. 13 was executed on TPC-H datasets of SF 1, 5, 10 and 20. We measured the query execution times in 3 scenarios: (i) both distributed in-memory data caching and persistence, (ii) without data persistence but with distributed in-memory data caching, and (iii) without both distributed in-memory data caching and persistence. Our results show that DaskDB performs best with scenario (i), when both distributed in-memory data caching and intermediate data persistence are supported. This offers a speed-up of 2$\times$ to 3$\times$ over scenario (ii) and a speed-up of 4$\times$ to 6$\times$ over scenario (iii).

VI. Conclusion

We presented DaskDB, a scalable data science system. It brings *in situ* SQL querying to a data science platform in a way that supports high usability, performance, scalability and built-in capabilities. Moreover, DaskDB also has the ability to incorporate any UDF into the input SQL query, where the UDF could invoke any Python library call. Furthermore, we introduce a novel distributed learned index that accelerates distributed join/merge operation. We evaluated DaskDB against two state-of-the-art systems, PySpark and Hive/Hivemall, using TPC-H benchmark and a custom UDF benchmark. We show that DaskDB significantly outperforms these systems.

VII. Acknowledgements

We would like to thank Arindum Roy and Jacob Shaw.

References

[1] J. M. Hellerstein *et al.*, “The MADlib Analytics Library: Or MAD Skills, the SQL,” *PVLDB*, vol. 5, no. 12, pp. 1700–1711, 2012.
[2] J. V. D'silva, F. De Moor, and B. Kemme, “AIDA: Abstraction for Advanced In-database Analytics,” *PVLDB*, vol. 11, no. 11, 2018.
[3] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki, “NoDB: efficient query execution on raw data files,” in *SIGMOD*, 2012.
[4] Y. Cheng and F. Rusu, “Parallel in-situ data processing with speculative loading,” in *SIGMOD*, 2014, p. 1287–1298.
[5] M. Olma *et al.*, “Adaptive partitioning and indexing for in situ query processing,” *VLDB J.*, vol. 29, no. 1, pp. 569–591, 2020.
[6] M. Zaharia *et al.*, “Spark: Cluster computing with working sets,” in *USENIX HotCloud*, 2010, pp. 10–10.
[7] “Hivemall,” https://hivemall.apache.org/.
[8] M. Rocklin, “Dask: Parallel computation with blocked algorithms and task scheduling,” in *Python in Science Conference*, 2015, pp. 130 – 136.
[9] “TPC-H,” http://www.tpc.org/tpch/.
[10] “Anaconda,” https://www.continuum.io/.
[11] E. Fouché, A. Eckert, and K. Böhm, “In-database analytics with ibmdbpy,” in *SSDBM*, 2018.
[12] M. Raasveldt and H. Mühleisen, “Data management for data science - towards embedded analytics,” in *CIDR*, 2020.
[13] “Apache Hadoop,” http://hadoop.apache.org/.
[14] X. Meng *et al.*, “MLlib: Machine Learning in Apache Spark,” *Journal of Machine Learning Research*, vol. 17, no. 34, pp. 1–7, 2016.
[15] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, “The case for learned index structures,” in *SIGMOD*, 2018, p. 489–504.
[16] A. Galakatos *et al.*, “FITing-Tree: A Data-Aware Index Structure,” in *SIGMOD*, 2019, p. 1189–1206.
[17] P. Ferragina and G. Vinciguerra, “The PGM-index: a fully-dynamic compressed learned indexwith provable worst-case bounds,” in *PVLDB*, vol. 13, no. 8, 2020, pp. 1162–1175.
[18] S. Zhang, S. Ray, R. Lu, and Y. Zheng, “SPRIG: A Learned Spatial Index for Range and kNN Queries,” in *SSTD*, 2021.
[19] “PostgresRAW,” https://github.com/HBPMedical/PostgresRAW.
[20] Dask Development Team, “Dask.distributed,” 2016. [Online]. Available: http://distributed.dask.org/
[21] A. Watson, D. S. V. Babu, and S. Ray, “Sanzu: A data science benchmark,” in *IEEE Big Data*, 2017, pp. 263–272.
[22] J. Chamanara, B. König-Ries, and H. V. Jagadish, “Quis: In-situ heterogeneous data source querying,” *Proc. VLDB Endow.*, Aug. 2017.
[23] “Heaviside Function,” https://en.wikipedia.org/wiki/Heaviside_step_function.
[24] M. Ziauddin *et al.*, “Dimensions based data clustering and zone maps,” *Proc. VLDB Endow.*, vol. 10, no. 12, p. 1622–1633, Aug. 2017.
[25] K. Nguyen *et al.*, “Skyway: Connecting managed heaps in distributed big data systems,” in *ASPLOS*, 2018, p. 56–69.
[26] “sql-metadata,” https://pypi.org/project/sql-metadata/.
[27] J. Wang *et al.*, “The Myria Big Data Management and Analytics System and Cloud Services,” in *CIDR*, 2017.