ConnectorX: Accelerating Data Loading From Databases to Dataframes

Xiaoying Wang[†*], Weiyuan Wu[†*], Jinze Wu[†], Yizhou Chen[†], Nick Zrymiak[†], Changbo Qu[†], Lampros Flokas[‡], George Chow[†], Jiannan Wang[†], Tianzheng Wang[†], Eugene Wu[‡], Qingqing Zhou[◇]

Simon Fraser University[†] Columbia University[‡] Tencent Inc.[◇]

{xiaoying_wang, youngw, jinze_wu, yizhou_chen_3, nzrymiak, changboq, kai_yee_chow, jnwang, tzwang}@sfu.ca[†]

{lamflokas, ewu}@cs.columbia.edu[‡] hewanzhou@tencent.com[◇]

ABSTRACT

Data is often stored in a database management system (DBMS) but dataframe libraries are widely used among data scientists. An important but challenging problem is how to bridge the gap between databases and dataframes. To solve this problem, we present ConnectorX, a client library that enables fast and memory-efficient data loading from various databases to different dataframes . We first investigate why the loading process is slow and consumes large memory. We surprisingly find that the main overhead comes from the client-side rather than query execution or data transfer. We integrate several existing and new techniques to reduce the overhead and carefully design the system architecture and interface to make ConnectorX easy to extend to various databases and dataframes. Moreover, we propose server-side result partitioning that can be adopted by DBMSs in order to better support exporting data to data science tools. We conduct extensive experiments to evaluate ConnectorX and compare it with popular libraries. The results show that ConnectorX significantly outperforms existing solutions. ConnectorX is open sourced at: https://github.com/sfu-db/connector-x.

PVLDB Reference Format:
Xiaoying Wang, Weiyuan Wu, Jinze Wu, Yizhou Chen, Nick Zrymiak, Changbo Qu, Lampros Flokas, George Chow, Jiannan Wang, Tianzheng Wang, Eugene Wu, Qingqing Zhou. ConnectorX: Accelerating Data Loading From Databases to Dataframes. PVLDB, 15(11): 2994 - 3003, 2022.
doi:10.14778/3551793.3551847

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/sfu-db/connectorx-bench.

1 INTRODUCTION

Dataframe libraries such as Pandas [59], Dask [65], and Modin [61] are widely used among data scientists for data manipulation and analysis. In contrast, enterprise environments often store their data in database management systems. Thus, the first step in most data science applications is to load data from the DBMS. Unfortunately, this data loading process is not only notoriously slow but also consumes inordinate amounts of client memory [4, 5, 7, 8, 53], which easily leads to out-of-memory errors or performance degradation. This issue is urgent since `read_sql` is on the critical path of many data science tasks such as ELT/ETL processes and exploratory data analysis, and it can take more than 50% of the time in a real-world ML pipeline [69]. Therefore, bridging the gap between databases and dataframes is of great interest to both academia and industry [45, 49, 53, 70].

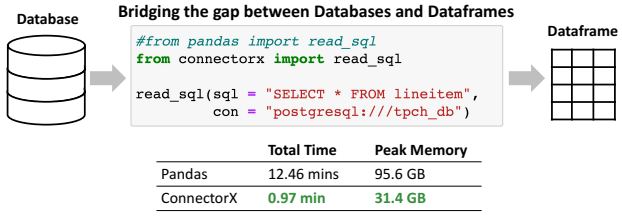

Figure 1: Speed up loading the lineitem table (7.2 GB in CSV) from database to dataframe with less memory usage.

Example 1.1. Pandas is the most widely used dataframe library in Python, with a total 1.2B downloads on PyPI as of Jan 2022. Suppose that a data scientist loads the TPC-H 'lineitem' table (7.2 GB) from PostgreSQL into a Pandas.DataFrame using the `Pandas.read_sql` *call in Figure 1. The function specifies a query string and database connection (e.g.,* `conn`*), retrieves the query results, and loads them into a DataFrame object. We conducted an experiment using two AWS instances, where PostgreSQL was deployed on one instance and the code was run on another instance (see Section 3 for details). The whole data loading process is highly inefficient—it takes 12.5 mins and consumes over 95.6 GB of memory. In fact, the actual time spent on query execution is less than 1 min (13× time overhead) and the final Pandas.DataFrame is only 24.4 GB (4× memory overhead) .*

This paper describes ConnectorX, a fast and memory-efficient data loading library that supports many DBMSs (e.g., PostgreSQL, SQLite, MySQL, SQLServer, Oracle) to client dataframes (e.g., Pandas, PyArrow, Modin, Dask, and Polars). We made our contributions while sought to address four major questions.

First, where are the actual data loading bottlenecks? We profile the `Pandas.read_sql` implementation in Section 3 (due to its popularity). We find that the runtime can be split into two parts:

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551847

[*] Both authors contributed equally to this research.

the server side runtime includes query execution, serialization, and network transfer, and the client side includes deserialization and conversion into a dataframe. We were surprised to find that >85% of time is spent in the client, and that the conversion materializes all intermediate transformation results in memory. These findings suggest that client-side optimizations are sufficient to dramatically reduce data loading costs.

Second, how do we both reduce the runtime and memory, while also making the system extensible to new DBMSs? In Section 4, we design a succinct domain-specific language (DSL) for mapping DBMS result representations into dataframes—this reduces the lines of code by 1-2 orders of magnitude as compared to not using our DSL. Under the covers, ConnectorX compiles the DSL to execute over a streaming workflow system that efficiently translates bytes received from the network into objects in memory. The workflow executes in a pipelined fashion, and seamlessly combines optimization techniques including parallel execution, string allocation optimizations, and an efficient data representation.

Third, are widely used client query partitioning scheme good enough? Parallelization via query partitioning is the dominant way to reduce execution and loading time. Existing libraries [56, 61, 65] including ConnectorX partition the query on the client, which is popular since it does not require modification to the server. Unfortunately, we find that it introduces extra user burden, load imbalances, wasted server resources, and data inconsistencies. Thus, we study server-side result partitioning in Section 5, where the DBMS directly partitions the query result and sends back in parallel without changing the protocol and access method. We prototype and demonstrate the efficacy using PostgreSQL, and advocate DBMS vendors to add this support to benefit data science applications.

Fourth, does a new data loading library matter? Since its first release in April 2021, ConnectorX has been widely adopted by real users, with a total of 300K+ downloads and 640+ Github stars in a year. It has been applied to extracting data in ETL [11] and loading ML data from DBMS [12]. It is also integrated into popular open source projects such as Polars [17], Modin [61] and DataPrep [14]. For example, Polars is the most popular dataframe library in Rust, and it uses ConnectorX as the default way to read data from various databases [13]. Our experiments in Section 6 show that ConnectorX significantly outperforms existing libraries (Pandas, Dask, Modin, Turbodbc) when loading large query results. Compared to Pandas, it reduces runtime by 13× and memory utilization by 3×.

2 RELATED WORK

Bridging the gap between DBMS and ML has become a hot topic in the database community. ConnectorX fits into the big picture by supporting efficient data loading from DBMSs to dataframes.

Accelerating Data Access From DBMS.

(1) Server-Side Enhancement. Accessing data from database systems through tuple-level protocol is notoriously slow [53, 63]. Previous work [63] shows that existing wire protocols suffer from redundant information and expensive (de)serialization, and propose a new protocol to tackle these issues. More approaches tend to leverage existing open data formats (e.g. Parquet [24], ORC [23], Avro [22], Arrow [6], Dermel [55], Pickle [1]) to speed up the process by avoiding tuple-level accessing. Li et. al [53] adopts Flight [42] to enable zero-copy on data export in Arrow IPC format. Data warehouses such as Redshift [19], BigQuery [16] and Snowflake [39] support unloading data into cloud stores (e.g. Amazon S3 [21], Azure Blob Storage [25], Google Cloud Storage [29]) in the format like CSV and Parquet directly [20, 28, 33]. Data lake and Lakehouse solutions [26, 67, 70] also advocate direct accessibility of open formats.

Parallelism is another effective way to speed up data transfer, supported by many tools [2, 10, 27] to move data between two file systems (e.g. HDFS, S3) or between file system and DBMSs. Databricks points out the single SQL endpoint bottleneck and proposes to tackle it with cloud fetch architecture [36], in which a query's result is exported to a cloud storage in multiple partitions, enabling parallel downloading from the client. Similar support is also provided by other cloud-native data warehouses [20, 28, 33].

However, all these server-side attempts require users to modify the source code of a database server or switch to a new one, which is often not feasible in real-world scenarios. Even if the solutions are supported, many of them require extra configuration efforts such as accessing to specific file systems (e.g. S3), which also differs among different vendors. Lastly, these works only provide part of the solution. To get the final dataframe for downstream tasks, there needs to be one extra conversion from the exported result. Unlike these approaches, ConnectorX provides a single-step solution leveraging existing DBMSs and client drivers, and achieves the maximum speed up by optimizing client executions within the current environmental setups. In addition, ConnectorX can also leverage and benefit from these server-side optimizations internally since it has no restriction in how to fetch data from DBMSs.

(2) Client-Side Optimization. ML and data analysis tools [38, 47, 56, 57, 60] tend to adopt dataframes [6, 17, 30, 32, 46, 59, 61, 62, 65] as the abstraction for data manipulation, many of which provide native DBMS I/O support with various optimization efforts. Pandas supports chunking [68] to reduce the memory pressure by loading data one chunk at a time. Modin, Dask, Spark leverage multiple cores through client-side query partitioning. Third-party library Turbodbc [9] provides state-of-the-art performance through batched data transfer on ODBC drivers. ConnectorX is superior than existing client libraries from three aspects. First, existing approaches are limited to client drivers with certain interfaces (e.g. Pandas, Turbodbc and Spark requires Python DB-API, ODBC and JDBC respectively), while ConnectorX has no such requirement and thus is able to leverage the fastest one. Second, they only target on one specific or a few dataframes. In contrast, ConnectorX is a general framework that can easily extend to any dataframe with high performance by implementing the corresponding interfaces. Third, comparing to these libraries, ConnectorX integrates more comprehensive optimization techniques, and thus could achieve the best performance.

Table 1: Memory analysis of `Pandas.read_sql`.

	Raw Bytes	Python Objects	Dataframe	Peak
PostgreSQL	12.4GB	52.6GB	24.4GB	95.6GB
MySQL	8.18GB	51.5GB	23.3GB[1]	99.1GB

[1]CHAR values are stripped in MySQL but not in PostgreSQL, which results in dataframes with different sizes.

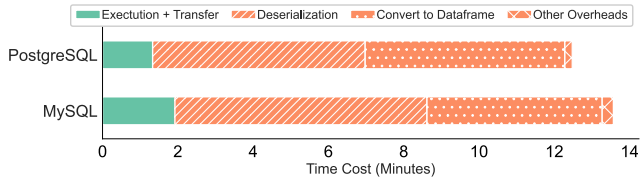

Figure 2: Time break down of `Pandas.read_sql`. (Orange parts happen on the client side.)

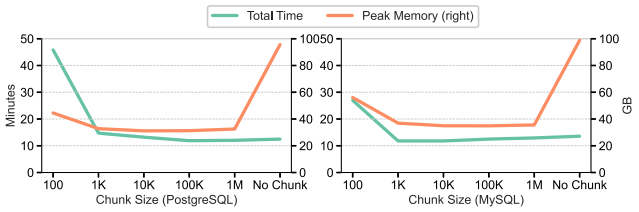

Figure 3: Time and memory change by varying chunk size.

Integrating DBMS with Data Science. Data scientists are in general more familiar with dataframe operations, so they usually choose to move the complete data from databases to the client machine and process it using data science tools. To avoid this costly data moving, some systems try to integrate with Python [37, 48, 49, 51, 54, 66] and R [40, 44, 52] environments in order to run ML operations inside database engines. Embedded analytical system DuckDB [64] is developed to avoid the bottlenecks of result set serialization and value-based APIs by making DBMS and analytic tools in the same address space. However, these approaches are still in an early stage and can only support a limited number of scenarios. Therefore, when there is no feasible integration system, it is still desirable to allow data scientists to move data out of the DBMS to conduct sophisticated analysis and build ML models on dataframe abstraction, requiring better design and implementation of `read_sql` like ConnectorX.

Other works are less aggressive in integrating DBMS with data science environments. Ibis [3] aims to convert dataframe operations to SQL queries and run them on a connected database. Declarative dataframe API [34, 56] is proposed to combine relational and procedural processing, which also allows cross-optimization between ML and database operators and has been studied by recent works [41, 45, 50]. These approaches still need to transfer data from DBMSs to the targeting environment, in which ConnectorX can be leveraged to speed up the movement.

3 AN ANATOMY OF PANDAS.READ_SQL

In this section, we take an in-depth look at `Pandas.read_sql` [59]. There are other libraries that also provide the `read_sql` functionality and we will discuss and compare with them in Section 6. We conduct an experiment between two AWS EC2 instances (r5.4xlarge, network bandwidth: 10 Gbit/s). A DBMS is deployed on one instance and `Pandas.read_sql` is executed on another instance to load the TPC-H `lineitem` table (SF=10) from the DBMS.

3.1 Where Does the Time Go?

Under the hood, `Pandas.read_sql` relies on driver libraries following the Python DB-API [43] to access databases. From the client's perspective, the overall process has three major steps:

(1) *Execution + Transfer*: Server executes the query and sends the result back to the client through network in bytes following a specific wire protocol.
(2) *Deserialization*: Client parses the received bytes and returns the query result in the form of Python objects.
(3) *Conversion to dataframe*: Client converts the Python objects into NumPy [46] arrays and constructs the final dataframe.

Figure 2 shows the time break down on PostgreSQL and MySQL, respectively. Note that orange parts all happen on the client side.

A surprising finding is that the majority of the time is actually spent on the client side rather than on the query execution or the data transfer. In this case, accelerating query execution or compressing the data for wire transfer [63] is less effective in speeding up `read_sql`. This result suggests that we should focus on optimizing the client side, which is dominated by two data conversions: *deserialization* and *conversion to dataframe* with each accounting for approximately 40% of the running time.

Another finding is that `read_sql` executes each step sequentially for PostgreSQL and MySQL by default [35][2]. That is, when the server side sends part of bytes to the client side, the client side does not process them right away but waits until all returned bytes are ready in a local buffer; when the client side derives part of Python objects, it does not convert them to a dataframe right away but waits until all Python objects are available. This will lead to two issues. First, all intermediate results will be temporarily kept in memory, which wastes too much memory as we will show in Section 3.2. Second, single thread execution cannot make full use of network and computational resources.

3.2 Where Does the Memory Go?

Next, we inspect the memory footprint of running `read_sql` and show the results in Table 1. Raw Bytes, Python Objects, and Dataframe represent the size of the bytes the client received, the intermediate Python objects, and the final dataframe, respectively.

We observe that the peak memory is approximately 4× larger than the size of the final dataframe. It is mainly caused by two reasons. First, the intermediate result is stored in Python objects. In Python, every object contains a header structure that maintains information like reference count and object's type in addition to the data itself. This will add some overhead on the size of the data. This overhead varies by different types. Take integer as an example: the actual data for an integer value only takes 8 bytes, but the header for this value has 20 bytes. Second, all the intermediate results are kept in memory until the final dataframe is generated. Specifically, `Pandas.read_sql` keeps three copies of the entire data in memory, which are stored in three different formats: Raw Bytes, Python Objects, and Dataframe. This unnecessary duplication of the same data is another cause of the high memory consumption.

How Much Can Chunking Help?. Chunking [58, 68] loads data chunk by chunk. For example, by specifying a chunk size of 1000, `read_sql` will fetch and process a chuck of 1000 rows of the query result at a time. We vary the chunk size and measure the running time and the peak memory of loading the lineitem table. Figure 3 shows the results. For fair comparison, we concatenate all the intermediate dataframes in the end to represent the entire query result. "No Chunk" represents that chunking is not used.

We see that chunking is indeed very effective in reducing memory usage because it does not hold all the intermediate results in memory. The peak memory usage can become almost equal to the

[2]For some other databases like Oracle, while the first two steps are conducted in parallel, the third step cannot start until the first two steps have finished.

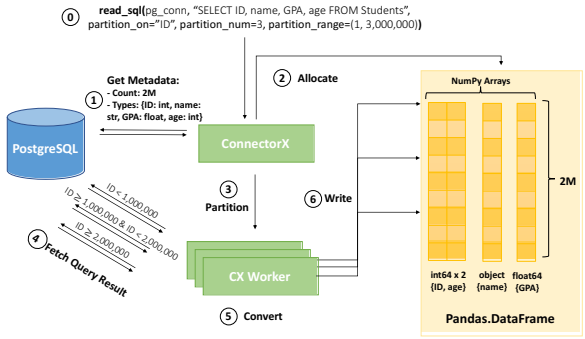

Figure 4: Workflow of ConnectorX.

final dataframe size when we set the chunk size within a certain range (e.g. 1K to 1M). However, it has little help in improving the running time of `read_sql`. In fact, it will introduce significant overhead when the chunk size is too small (e.g. 100). Moreover, the user needs to write extra code in order to enable chunking and handle a stream of dataframes.

Opportunities. Through an in-depth analysis of `read_sql`, we identify the opportunity to improve its performance in optimizing the client side execution through reducing Python overhead, minimizing data copies, pipelining and parallelization.

4 CONNECTORX

In this section, we first present how we leverage the above opportunities to improve the performance of `read_sql` from PostgreSQL to Pandas. Then we show how ConnectorX can be easily extended to support a large number of databases and dataframes.

4.1 How to Speed Up?

Overall Workflow. Learning from chunking, ConnectorX adopts a streaming workflow, where the client loads and processes a small batch of data at a time. In order to avoid the extra data copy and concatenation at the end, ConnectorX adds a Preparation Phase to its workflow. The goal of this phase is to pre-allocate the result dataframe so that the parsed values can be directly written to the corresponding final slots during execution. Figure 4 illustrates the overall workflow, which consists of two two phases: Preparation Phase (①-③) and Execution Phase (④-⑥).

In the Preparation Phase, ConnectorX ① queries the metadata of the query result, including the number of rows and the data type for each column. With this information, it ② constructs the final Pandas.DataFrame by allocating the NumPy arrays accordingly. In order to leverage multiple cores on the client machine, ConnectorX supports ③ partitioning the query for parallel execution.

The Execution Phase is conducted iteratively in a streaming fashion. ConnectorX assigns each partitioned query to a dedicated worker thread, which streams the partial query result from the DBMS into dataframe independently in parallel. Specifically, in each iteration of a worker thread, it ④ fetches a small batch of the query result from the DBMS, ⑤ converts each cell into the proper data format, and ⑥ writes the value directly to the dataframe. This process repeats until the worker exhausts the query result.

Parallel Execution. As shown above, ConnectorX leverages query partitioning for parallel execution. Suppose that the given query is denoted by Q. The user specifies a range partitioning scheme over the query result, which consists of a partition key, a partition

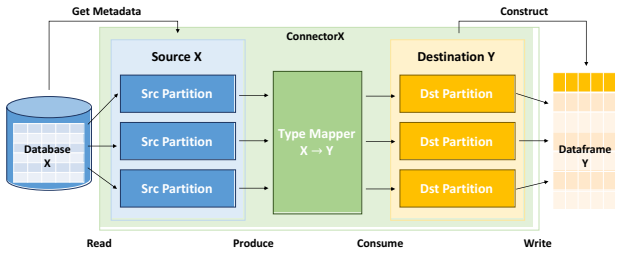

Figure 5: Overall architecture of ConnectorX.

number, and a partition range. Based on the scheme, Q can be partitioned into a set of subqueries, $q_1, q_2, \cdots, q_n$. The partition scheme guarantees that the union of the subquery results of $q_1, q_2, \cdots, q_n$ is equal to the query result of Q. Thus, by fetching the results of $q_1, q_2, \cdots, q_n$, ConnectorX obtains the result of Q.

In Figure 4, the partitioning scheme is shown in step ⓪: the partition column ID, the partition number 3, and a partition range (1, 3,000,000). If the range is not specified, ConnectorX automatically sets the range by issuing query `SELECT MIN(ID), MAX(ID) FROM Students`. Then, ConnectorX equally partitions the range into 3 splits and generate three subqueries[3]:

```
q1: SELECT ... FROM Students WHERE ID < 1,000,000
q2: SELECT ... FROM Students WHERE ID ∈ [1,000,000, 2,000,000)
q3: SELECT ... FROM Students WHERE ID ≥ 2,000,000
```

This partitioning strategy is also adopted by other client libraries [56, 61, 65]. We will further discuss it in Section 5.

String Allocation Optimization. ConnectorX pre-allocates the NumPy arrays in advance to avoid extra data copy. However, the buffers that the string objects point to have to be allocated on-the-fly after knowing the actual length of each value. Moreover, constructing a string object is not thread-safe in Python. It needs to acquire the Python Global Interpreter Lock (GIL), which could slow down the whole process when the degree of parallelism is large (Section 6.1). To alleviate this overhead, ConnectorX constructs a batch of strings at a time while acquiring the GIL instead of allocating each string object separately. To shorten the time of holding the GIL, we do not copy the real data during the construction, but write the bytes into the allocated buffer after releasing the GIL.

Suppose the query result contains 100 strings of 10 bytes each. A simple approach would be creating Python string objects on demand. That is, for each received string from the database driver, we (1) acquire the GIL, (2) allocate a Python string object of 10 bytes, (3) copy the content to the allocated buffer, (4) release the GIL. Unlike this, ConnectorX keeps the string bytes temporarily in memory and creates Python strings in batches. Therefore, ConnectorX only needs to acquire the GIL once instead of 100 times. Furthermore, it early releases the lock by exchanging the order of step (3) and step (4) because only string allocation requires holding the GIL. Consequently, the contention on the GIL is largely reduced.

Efficient Data Representation. The limitation of Python shown in Section 3.2 indicates that a more efficient data representation is needed. Therefore, we decide to use a native programming language to implement ConnectorX. We choose Rust since it provides efficient performance and guarantees memory safety. In addition, there is a variety of high-performance client drivers for different databases in Rust that ConnectorX can directly build on. In order to

[3]For complex queries, we use nested queries to partition their query results: q_1 = `SELECT * FROM (SELECT * FROM Students) AS T WHERE ID < 1,000,000`.

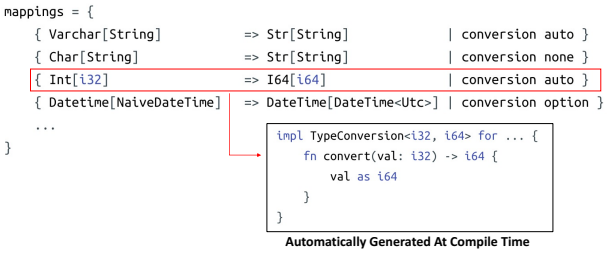

Figure 6: Example of using type mapping DSL in ConnectorX, with illustration of simplified generated code.

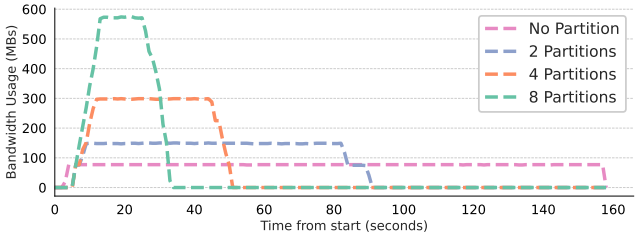

Figure 7: Network utilization by varying # partitions.

fit into the data science ecosystem in Python, ConnectorX provides a Python binding with an easy-to-use API. This allows data scientists to download ConnectorX using "`pip install connectorx`" and directly replace `Pandas.read_sql` with ConnectorX.`read_sql`.

4.2 How to Extend?

Overall Architecture. ConnectorX consists of three main modules: Source (e.g. PostgreSQL, MySQL), Destination (e.g. Pandas, PyArrow) and a bridge module Type Mapper in the middle to define how to convert the physical representation for the data from Source to Destination. Figure 5 illustrates the high-level architecture.

Each supported DBMS in ConnectorX has a corresponding Source module, which reads and parses data from the DBMS . The Source module is able to generate a group of Source Partition instances, each of which is assigned a subquery. The Destination module generates the final dataframe, including constructing the dataframe object and letting a dedicated Destination Partition to consume and write the data produced from a Source Partition to the correct position in the dataframe. A Type Mapper module consists of a set of rules that specify how to convert data from a specific Source type to a specific Destination type. During runtime, each subquery will be handled by a single thread, which forwards data from a Source Partition to a Destination Partition by looking up the conversion rules in the corresponding Type Mapper.

Interface Design. Adding a new Source involves two tasks (new Destination is similar): (1) Connecting to the new Source and supporting the functionalities required by ConnectorX (e.g. querying metadata, fetching query results); (2) Defining the type mapping from the new Source to existing Destinations.

(1) Connection. The same with other libraries, ConnectorX leverages existing client drivers to implement the functionality. However, other libraries require drivers provide a specific API (e.g. Pandas and Turbodbc requires Python DB-API and ODBC respectively). While ConnectorX has no such requirement, which gives it the flexibility to choose the fastest client driver for each DBMS. Adding a new Source only requires implementing a set of succinct interfaces with an existing driver. We leave details out to report [69].

(2) Type Mapping. Different database systems define their own type systems and physical type representations. Thus, a type mapping for each (database, dataframe) pair is needed (e.g. CHAR, and DATE in PostgreSQL can be converted to object, and datetime64 in Pandas, and large_utf8, and date64 in PyArrow, respectively).

A naive way to support this is to define how to convert each type from each Source to each Destination manually. However, this will lead to two pain points. First, there will be a lot of trivial code for types with the same physical representation. Because in many cases, Source and Destination choose the same physical representation (e.g. both PostgreSQL.INT8 and Pandas.int64 use 64-bit signed integer type i64 as physical type) or types that the conversion is trivial (e.g. casting from i32 to i64 for PostgreSQL.INT4 to Pandas.int64). Second, the code will become hard to maintain due to the large amount of conversion functions. Suppose each DBMS has 15 data types on average, there will be 150 ($15 \times 5 \times 2$) type conversion functions to support five DBMSs and two dataframes.

To mitigate the aforementioned issues, ConnectorX defines a domain specific language (DSL) to help the developers define the type mappings, leveraging the modern macro support in Rust [31]. Each line consists of three parts: logical type and corresponding physical type of Source, the matched logical type and physical type of Destination, and the conversion implementation choice including `auto`, `none`, and `option`. The physical types are specified in square brackets following the logical counterparts, which makes the mapping relation of both logical-physical and Source-Destination type pair clear. For trivial conversions that are automatically supported, like i32 to i64, a developer could set the conversion as `auto` and ConnectorX will automatically generate the corresponding conversion functions like the example shown in Figure 6. `option` is used for non-trivial conversions, for which the developer is required to implement the corresponding conversion function. To avoid repeated definitions, `none` indicates that the physical type pair is already handled. This simple DSL makes the relation of type mapping intuitive and easy to maintain. We found it has helped shorten code related to type mapping by 97% (from 37k to 1k lines of code).

5 QUERY PARTITIONING

5.1 Client-Side Query Partitioning

The client-side query partitioning scheme leveraged by ConnectorX and other libraries [56, 61, 65] can accelerate `read_sql` through utilizing the high network bandwidth and CPU resource more efficiently. Figure 7 shows the network utilization of ConnectorX by varying the number of partitions. It is clear that No Partition cannot saturate the network bandwidth at all. With more connections fetching data in parallel, bandwidth could be leveraged more sufficiently and thus leading to better end-to-end performance (`read_sql` finishes when bandwidth usage drops to 0).

The reason that this method is widely adopted by client libraries is because it does not require any modification to the database server. However, it has several downsides: (1) *User Burden.* To enable query partitioning, user has to take extra effort to specify a partitioning scheme over the query result. (2) *Load Imbalance.* If the result is not evenly partitioned, stragglers may arise, hurting overall performance. (3) *Data Inconsistency.* Since subqueries are sent to the server from independent sessions, their results may be derived from different snapshots. (4) *Wasted Resource.* Different subqueries may share the same costly subplan (e.g., full-table scan). Since the DBMS processes each query independently, the sub-plan may be repeatedly executed for many times.

Table 2: Comparison of different partitioning approaches. C (S) represents client (server)-side partition. Bold font indicates server-side outperforms client-side partition.

	# Scan		# Disk Block Miss		Total Time (s)	
No Partition	1		1.1M		156.1	
# Partitions	C	S	C	S	C	S
2	3	**1**	3.2M	**1.1M**	86.4	86.7
4	5	**1**	3.8M	**1.1M**	49.1	**45.7**
8	9	**1**	3.8M	**1.1M**	30.4	**23.9**
16	17	**1**	17.1M	**1.1M**	26.7	**19.6**

5.2 Server-Side Result Partitioning

On the other hand, partitioning the query on the database server side would address the aforementioned issues. Specifically, DBMS partitions the query result into n equally-sized partitions and allows the client to fetch them through n connections in parallel with existing protocol. Unlike client-side query partitioning, server-side result partitioning does not need the user to input any extra information. With the help of internal statistics and a cost estimator, the DBMS has a better chance to partition the result more evenly. It can also easily guarantee data consistency and avoid wasted resource since the DBMS has all the necessary information to partition and conduct executions on the same database snapshot. In the following, we discuss the potential design for this proposal.

SQL Syntax & Workflow. A key requirement of supporting server-side result partitioning is the mechanism of indicating the relationship between different independent connections. To achieve this, we can extend the existing concept of database cursor and define the SQL syntax for server-side result partitioning as follow:

```
DECLARE name CURSOR FOR query INTO n;
FETCH ALL FROM partition_id OF name;
```

In order to support accessing the same query result through different connections, the client first establishes a connection and declares a cursor for the original query with an associated name. By specifying the partition number partition_num, this cursor now becomes globally visible to the same user in other sessions as long as it is still valid. And its result can be then fetched through concurrent connections with different partition_id $(0, 1, ..., n-1)$. The cursor will be released eventually when all query results are consumed.

Prototype Evaluation. Naturally, there are many approaches to support server-side result partitioning. Briefly, our prototype modifies the PostgreSQL engine for SP queries by splitting the query plan into n subplans that could be executed in different connections in parallel. Unlike client-side query partitioning, where each partitioned query scans the entire table independently. Each subplan scans $\frac{1}{n}$ of the total disk pages on the same snapshot of the data, which avoids resource waste and inconsistency. Other approaches, such as executing the whole query and distributing the results to different connections are also feasible. The technical report [69] describes our prototype and alternative designs in more details.

We execute server-side partitioning prototype and client-side partitioning to fetch lineitem table from PostgreSQL (the same setup with Section 3). Table 2 shows the results of using ConnectorX in different scenarios. # Scan and # Disk Block Miss represent the number of times the table has been scanned, and the number of disk blocks read (subtracting the number of cache hit), respectively. We also show the time usage (Total Time) from initiating the query

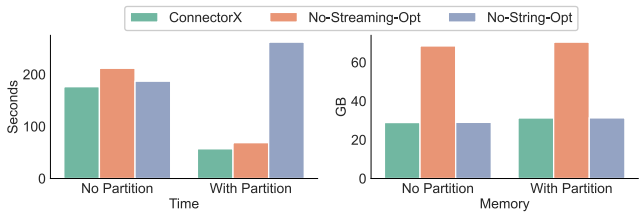

Figure 8: Ablation study.

to getting the final dataframe in order to illustrate the impact of each partitioning approach to the end-to-end `read_sql` procedure.

Without partitioning, PostgreSQL scans the entire table once and it takes 156.1 seconds to get the result dataframe. Although client-side partitioning improves the efficiency of `read_sql`, it also puts heavier burden on the DBMS. The number of scans increases along with the number of partitions (plus one extra scan to query the range of a given column for query partitioning). The number of blocks that need to be loaded from disk is approximately $\frac{3.8}{1.1} = 3.5\times$ larger when the number of partitions is small. With 16 partitions, it becomes 15.5× larger due to higher contention on the buffer pool. On the contrary, server-side partitioning shows the same statistics with no partition. Furthermore, with more partitions, the resource saving on the server side can further reduce end-to-end time ($\frac{26.1-19.6}{26.1} = 25\%$ on 16 partitions). To conclude, server-side partitioning allows the client to fully leverage the network and computation resources, without extra overhead on the database server. We hope that DBMS vendors can consider adding the support of server-side result partitioning in the future.

6 EVALUATION

Datasets & Workloads. (1) *TPC-H [18].* We use the TPC-H benchmark dataset with scale factor set to 10. We select lineitem table (60M rows, 16 columns, approximately 7.2GB in CSV format). We use column l_orderkey as the partitioning column, which is evenly distributed and thus the cardinality of each subquery is similar. We also test on 22 SPJ queries, with the fetched result size ranging from 100K to 59M. (2) *DDoS [15].* This dataset contains 12.8M traffic flows (6.3GB in CSV). This is an ML dataset with 84 feature columns and a label column. The majority of the columns are numerical (51 DECIMAL and 29 INTEGER), and the rest five are VARCHAR. The tested query is to load the entire table. The ID column is adopted as the partitioning column when needed. Notice that ID is not evenly distributed. When using four partitions, the size of each partition is approximately $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, and $\frac{1}{8}$ of the entire table.

Baselines. We compare ConnectorX with four popular libraries: Pandas [59], Dask [65], Modin [61] and Turbodbc [9]. Since Turbodbc does not support Pandas destination directly, we convert its NumPy array result to Pandas to ensure a fair comparison. The detailed approach is available in our technical report [69].

Hardware & Platform. Our experiments are conducted on two AWS EC2 r5.4xlarge instances (16 vCPUs, 128GB main memory, and 10Gbit/s network bandwidth) by default. We deploy the database on one machine and run `read_sql` from another. We also show the performance comparison under other network conditions, including when the server and client reside on the same r5.4xlarge instance (Local) and on two locally hosted machines with four Intel Xeon E7-4870 v4 CPUs (16 cores in total), 128GB memory and 200Mbit/s network. We use three open-source databases (PostgreSQL, MySQL, SQLite) and one commercial database (DBMS-A).

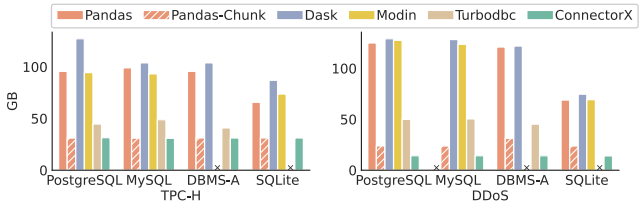

Figure 9: Memory comparison on four database systems. ("×" is placed if a method does not support the DBMS or cannot handle the large query result.)

Implementation. We have made the scripts, datasets and workloads publicly available at https://github.com/sfu-db/connectorx-bench. We run each experiment five times and report the averaged result. We conduct experiments under both without partitioning (No Partition) and with four partitions (With Partition) settings. We also evaluate the performance by varying the number of partitions and compare with a parallel file export baseline in [69].

6.1 Ablation Study

We conduct an ablation study to gain a deep understanding of ConnectorX's performance and verify the efficacy of the three optimization techniques: i) Query partitioning; ii) Streaming workflow; iii) String allocation optimization. Since Figure 7 has already shown the efficacy of query partitioning, here we evaluate the other two. We vary the implementation of ConnectorX and observe the performance change by loading the lineitem table from PostgreSQL to Pandas. The results are shown in Figure 8. No-Streaming-Opt represents that the streaming workflow is disabled; No-String-Opt represents that the string allocation optimization is disabled.

In terms of running time, ConnectorX (with all optimizations) is the fastest both with and without partitioning. Without the streaming workflow, the performance drops approximately by 20% in both cases. The impact of string allocation optimization varies in different numbers of partitions. It slows down the process by only 6% when no partition, but becomes 4.6× slower with partitioning. This is because more partitions lead to more contention on the GIL, thus slowing down the process. For peak memory usage, applying partitioning has little impact on memory consumption. No-Streaming-Opt needs 2.3× more memory due to the large intermediate results. However, it (70.4GB) still saves more than 20GB of memory comparing to the 95.6GB peak memory usage of Pandas's batch solution shown in Table 1. This validates the efficiency of using Rust in terms of data representation.

6.2 Performance Comparison

We compare ConnectorX with four baselines to fetch query result into a Pandas.DataFrame. Due to space limitation, we leave the discussion on more dataframes to our technical report [69].

Memory Comparison. Figure 9 evaluates the peak memory usage of loading the entire TPC-H lineitem and DDoS tables. We show the result of with partitioning Modin, Dask, and ConnectorX, which is usually no less than without partitioning. Pandas-Chunk enables chunking for Pandas (chunk size: 10K according to Figure 3).

On TPC-H, the memory consumption of ConnectorX and Pandas-Chunk are almost the same on all DBMS. Their peak memory values are consistently 3× less than Pandas on the client-server databases and 2× less on SQLite. Dask and Modin show similar results with Pandas. Turbodbc is more memory efficient, but it still needs around 10GB more memory than ConnectorX. As for DDoS, ConnectorX outperforms other baselines to a much larger extent because of its efficient handling of the DECIMAL type, which is the majority type in DDoS and Python inflates it 13×. Another interesting finding is that unlike TPC-H, ConnectorX uses approximately 2× less memory than Pandas-Chunk on DDoS. When concatenating the chunked dataframes at the end of Pandas-Chunk, the memory of NumPy arrays will be doubled since they need to be copied to a larger continuous buffer. But string objects that NumPy arrays point to only need to increase the reference count by one without copying. Since string values only take a small proportion of the memory usage in DDoS dataframe, the concatenation overhead of Pandas-Chunk is much higher than on TPC-H.

Table 3: Speed compared to ConnectorX (With Partition) on PostgreSQL under different network bandwidth.

Bandwidth	Local	10 Gbit/s	200 Mbit/s
Pandas	14.26×	12.80×	3.05×
Dask	6.09×	4.80×	5.31×
Modin	3.88×	3.45×	1.58×
Turbodbc	6.64×	6.21×	1.90×
ConnectorX-NoPart	**3.16×**	**3.03×**	**1.08×**

Speed Comparison. We show the speed comparison under high bandwidth network setting (10Gbit/s, except for SQLite, which reside on the same client instance) in Figure 10. To fairly compare with baselines that do not support query partitioning, we show the result of Modin, Dask and ConnectorX when no partitioning is applied (the left two figures). We also test them when using partitions to observe the potential speedup under multiple cores on the client instance (the right two figures).

(1) No Partitioning. ConnectorX performs the best in almost all cases. It outperforms Pandas by 4.2×, 5.2×, 3.0× and 2.3× on PostgreSQL, MySQL, DBMS-A and SQLite respectively on TPC-H, and 7.1×, 6.0× and 5.2× on PostgreSQL, DBMS-A and SQLite on DDoS. Modin and Dask have extra overhead in transferring the result from worker processes. In addition, they could not finish in many cases when no partition is applied due to out-of-memory issue. Compared with ConnectorX, Turbodbc can achieve similar or even better performance on DBMS-A, but is 2.0× (1.8×) and 2.3× (1.3×) slower for PostgreSQL and MySQL on TPC-H (DDoS). This variance comes from how efficient the DBMS's ODBC driver is implemented, which highly determines Turbodbc's performance.

(2) With Partitioning. ConnectorX is the fastest one in all experiments. Only Dask and Modin support query partitioning among baselines. To make the comparison more clear, we copy the results of Pandas and Turbodbc without partitioning to the same figure. With partitioning, ConnectorX further accelerates and becomes up to 12.8× (14.5×) and 6.2× (3.8×) faster compared to Pandas and Turbodbc respectively on TPC-H (DDoS). Modin's speed also gets improved. But it is still 2.5× to 6.7× slower than ConnectorX. Dask benefits from partitioning as well but is less stable. Sometimes it needs to restart the workers when reaching to the memory limit, which makes it slower than ConnectorX and Modin, and may even slower than Pandas.

(3) Different Network Conditions. We test all methods on PostgreSQL under different network conditions. To see the gap clearly, we use ConnectorX with partitioning as baseline and show its speedup w.r.t. each method in Table 3. ConnectorX-NoPart represents the result of ConnectorX when no partition is applied, and

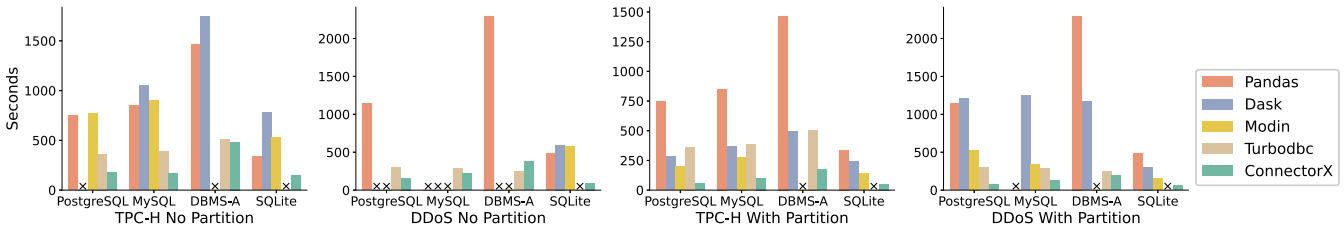

Figure 10: Speed comparison ("×" is placed if a method does not support the DBMS or cannot handle the large query result.)

Table 4: Speed up of ConnectorX to Pandas on SPJ queries. (Different color means ConnectorX is faster / slower than Pandas.)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Result Row# (M)	59.1	4.6	17.3	10.4	1.5	20.9	4.0	2.4	3.3	37.2	7.7	1.2	15.3	27.3	27.3	1.2	1.8	5.6	0.6	0.1	0.9	0.1
No Partition	3.8×	2.2×	2.5×	1.7×	1.2×	3.8×	1.6×	1.6×	2.3×	2.5×	3.3×	1.2×	2.5×	2.7×	2.8×	2.5×	1.0×	1.6×	1.2×	1.1×	1.0×	1.0×
With Partition	8.4×	3.0×	3.2×	1.2×	0.4×	3.4×	0.5×	0.4×	1.0×	3.3×	5.8×	0.5×	2.2×	3.1×	4.3×	1.1×	0.7×	0.7×	0.3×	0.6×	0.5×	0.5×

both Modin and Dask leverage partitioning. It is clear that ConnectorX remains the fastest in all cases since all values > 1. Also, the gap between ConnectorX and other baselines becomes larger when the bandwidth is higher, which shows the efficiency of ConnectorX in leveraging the bandwidth resource.

(4) SPJ Queries. We evaluate ConnectorX using more complex queries. We consider the queries with joins and predicates because it will affect the server's query execution and data transfer time. Specifically, we generate 22 queries[4] (one from each TPC-H query template). For each query, we keep SELECT, PROJECT and JOIN operators. We also alter the predicates manually to make sure the result size is in a large range (100K to 59M) and flatten some of the nested queries to have more variety in terms of query complexity. We choose the first numerical column as the partition column for query partitioning on ConnectorX. For complex queries, getting metadata like the number of result rows becomes slower. In order to avoid the potentially costly COUNT query, in this situation we choose and also suggest our users to use Arrow as an intermediate destination from ConnectorX and convert it into Pandas using Arrow's to_pandas[5] API.

We run all 22 queries on PostgreSQL and compare the performance of ConnectorX with Pandas. The result in Table 4 shows the speedup of ConnectorX to Pandas. It is clear that without partitioning, ConnectorX is faster than Pandas by up to 3.8× or at least shows similar performance. Partitioning could sometimes further speed up the process by up to 8.4×. Surprisingly, it could also further complicate the query, which may result in generating slower query plans and also may have the overhead in partition column range querying. In our experiment, some queries show performance degradation with partitioning by up to 3.3× (Q19) especially when the result set is small. This finding further motivates the support of server-side result partitioning discussed in Section 5.2.

Note that ConnectorX targets the scenarios that require fetching large query result sets. It speeds up the process by optimizing client-side execution and saturating both network and computational resources through parallelism. When network or query execution on the DBMS is the bottleneck (e.g. complex queries with small result sets), however, ConnectorX brings less benefit and sometimes it can be even slower due to the overhead in fetching metadata.

(5) Scalability. Finally, we evaluate the scalability of each approach by varying the scale factor of TPC-H lineitem table from 1 to 100. We run the client side on an AWS EC2 r5.16xlarge instance (64 vCPUs, 512GB main memory) in this experiment in order to hold large query result dataframes in memory. The result is shown in Figure 11, in which we report the speed when partitioning is enabled for Modin, Dask and ConnectorX. We can clearly see that ConnectorX scales linearly and consistently outperforms other baselines to a large extent.

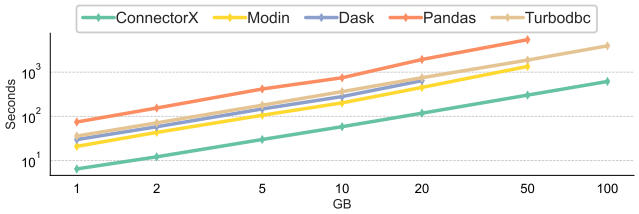

Figure 11: Speed comparison on PostgreSQL under different scale of TPC-H.

7 CONCLUSION

In this paper, we proposed ConnectorX, an open-source library for loading query results from DBMSs to dataframes in a fast and memory-saving way. We conducted a thorough analysis on the popular `Pandas.read_sql` function, and identified optimization opportunities on client-side execution. We developed ConnectorX targeting at optimizing client-side execution of `read_sql` without modifying the existing implementation of database servers as well as client drivers. ConnectorX also provides modular interfaces for contributors to add support for more DBMSs and dataframes easily. We further identified the drawbacks of current client-side query partitioning approaches that ConnectorX and other libraries are using, and proposed that database systems should support server-side result partitioning in order to tackle the issues. We performed experiments showing that ConnectorX significantly outperforms Pandas, Dask, Modin and Turbodbc in terms of both speed and memory usage under different scenarios.

ACKNOWLEDGMENTS

This work was supported in part by Mitacs through an Accelerate Grant, NSERC through a discovery grant and a CRD grant as well as NSF awards 1845638, 1740305, 2008295, 2106197, 2103794, and Amazon, Google, and Columbia SIRS. All opinions, fndings, conclusions and recommendations in this paper are those of the authors and do not necessarily refect the views of the funding agencies.

[4]https://github.com/sfu-db/connectorx-bench/tree/main/tpch-spj-workload/spj
[5]https://arrow.apache.org/docs/python/generated/pyarrow.Table.html?pyarrow.Table.to_pandas

REFERENCES

[1] 2001-2022. pickle — Python object serialization. https://docs.python.org/3/library/pickle.html. Accessed: 2022-05-01.

[2] 2011-2019. Apache Sqoop. https://sqoop.apache.org/. Accessed: 2022-05-01.

[3] 2014-2021. Ibis: Write your analytics code once, run it everywhere. http://ibis-project.org. Accessed: 2022-01-27.

[4] 2016. pandas read_sql is unusually slow. https://stackoverflow.com/questions/40045093/pandas-read-sql-is-unusually-slow. Accessed: 2022-01-27.

[5] 2016. Pandas using too much memory with read_sql_table. https://stackoverflow.com/questions/41253326/pandas-using-too-much-memory-with-read-sql-table. Accessed: 2022-01-27.

[6] 2016-2022. Apache Arrow. https://arrow.apache.org/. Accessed: 2022-01-27.

[7] 2017. Program (Time) Bottleneck is Database Interaction. https://stackoverflow.com/questions/44154430/program-time-bottleneck-is-database-interaction. Accessed: 2022-01-27.

[8] 2017. Use Turbodbc/Arrow for read_sql_table. https://github.com/pandas-dev/pandas/issues/17790. Accessed: 2022-01-27.

[9] 2017-2021. Turbodbc - Turbocharged database access for data scientists. https://turbodbc.readthedocs.io/en/latest/. Accessed: 2022-01-27.

[10] 2018. AWS CLI s3. https://awscli.amazonaws.com/v2/documentation/api/latest/reference/s3/index.html. Accessed: 2022-05-01.

[11] 2021. ConnectorX for ETL workload. https://github.com/sfu-db/connector-x/discussions/133. Accessed: 2022-01-27.

[12] 2021. ConnectorX for ML feature fetching. https://github.com/sfu-db/connector-x/issues/140#issuecomment-948918848. Accessed: 2022-01-27.

[13] 2021. ConnectorX integrates with dataframe system. https://pola-rs.github.io/polars-book/user-guide/howcani/io/read_db.html. Accessed: 2022-01-27.

[14] 2021. DataPrep: The easiest way to prepare data in Python. https://dataprep.ai/. Accessed: 2021-01-27.

[15] 2021. DDoS Dataset. https://www.kaggle.com/devendra416/ddos-datasets. Accessed: 2022-01-27.

[16] 2021. Google BigQuery. https://cloud.google.com/bigquery. Accessed: 2022-01-27.

[17] 2021. Polars: Fast multi-threaded DataFrame library in Rust and Python. https://github.com/pola-rs/polars. Accessed: 2022-01-27.

[18] 2021. TPC-H Homepage. http://www.tpc.org/tpch. Accessed: 2022-01-27.

[19] 2022. Amazon Redshift. https://aws.amazon.com/redshift/. Accessed: 2022-05-01.

[20] 2022. Amazon Redshift: Unloading data to Amazon S3. https://docs.aws.amazon.com/redshift/latest/dg/t_Unloading_tables.html. Accessed: 2022-05-01.

[21] 2022. Amazon S3. https://aws.amazon.com/s3/. Accessed: 2022-05-01.

[22] 2022. Apache Avro. https://avro.apache.org/. Accessed: 2022-05-01.

[23] 2022. Apache ORC. https://orc.apache.org/. Accessed: 2022-05-01.

[24] 2022. Apache Parquet. https://parquet.apache.org/. Accessed: 2022-05-01.

[25] 2022. Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs/. Accessed: 2022-05-01.

[26] 2022. Azure Data Lake. https://azure.microsoft.com/en-us/solutions/data-lake/. Accessed: 2022-05-01.

[27] 2022. DistCp. https://hadoop.apache.org/docs/r3.1.3/hadoop-distcp/DistCp.html. Accessed: 2022-05-01.

[28] 2022. Google BigQuery: Exporting table data. https://cloud.google.com/bigquery/docs/exporting-data. Accessed: 2022-05-01.

[29] 2022. Google Cloud Storage. https://cloud.google.com/storage. Accessed: 2022-05-01.

[30] 2022. Ray Data. https://docs.ray.io/en/latest/data/getting-started.html. Accessed: 2022-05-01.

[31] 2022. The Rust Programming Language - Macro. https://doc.rust-lang.org/book/ch19-06-macros.html. Accessed: 2022-01-27.

[32] 2022. Spark SQL, DataFrames and Datasets Guide. https://spark.apache.org/docs/latest/sql-programming-guide.html. Accessed: 2022-05-01.

[33] 2022. Unloading Data from Snowflake. https://docs.snowflake.com/en/user-guide-data-unload.html. Accessed: 2022-05-01.

[34] 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. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1383–1394. https://doi.org/10.1145/2723372.2742797

[35] SQLAlchemy authors and contributors. 2007-2022. Using Server Side Cursors (a.k.a. stream results). https://docs.sqlalchemy.org/en/14/core/connections.html#using-server-side-cursors-a-k-a-stream-results. Accessed: 2022-01-27.

[36] Stefania Leone Bogdan Ionut Ghit, Juliusz Sompolski and Reynold Xin. 2021. How We Achieved High-bandwidth Connectivity With BI Tools. https://databricks.com/blog/2021/08/11/how-we-achieved-high-bandwidth-connectivity-with-bi-tools.html. Accessed: 2022-01-27.

[37] Lingjiao Chen, Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. 2017. Towards Linear Algebra over Normalized Data. *Proc. VLDB Endow.* 10, 11 (2017), 1214–1225. https://doi.org/10.14778/3137628.3137633

[38] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. ACM, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[39] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 215–226. https://doi.org/10.1145/2882903.2903741

[40] Sudipto Das, Yannis Sismanis, Kevin S. Beyer, Rainer Gemulla, Peter J. Haas, and John McPherson. 2010. Ricardo: integrating R and Hadoop. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, Ahmed K. Elmagarmid and Divyakant Agrawal (Eds.). ACM, 987–998. https://doi.org/10.1145/1807167.1807275

[41] Joseph Vinish D'silva, Florestan De Moor, and Bettina Kemme. 2018. AIDA - Abstraction for Advanced In-Database Analytics. *Proc. VLDB Endow.* 11, 11 (2018), 1400–1413. https://doi.org/10.14778/3236187.3236194

[42] The Apache Software Foundation. 2016-2021. Apache Arrow Flight. https://arrow.apache.org/docs/format/Flight.html. Accessed: 2022-01-27.

[43] The Python Software Foundation. 2001. PEP 249 – Python Database API Specification v2.0. https://www.python.org/dev/peps/pep-0249/. Accessed: 2022-01-27.

[44] Philipp Große, Wolfgang Lehner, Thomas Weichert, Franz Färber, and Wen-Syan Li. 2011. Bridging Two Worlds with RICE Integrating R into the SAP In-Memory Computing Engine. *Proc. VLDB Endow.* 4, 12 (2011), 1307–1317. http://www.vldb.org/pvldb/vol4/p1307-grosse.pdf

[45] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. 2021. Putting Pandas in a Box. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper07.pdf

[46] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[47] J. D. Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering* 9, 3 (2007), 90–95. https://doi.org/10.1109/MCSE.2007.55

[48] Matthias Jasny, Tobias Ziegler, Tim Kraska, Uwe Röhm, and Carsten Binnig. 2020. DB4ML - An In-Memory Database Kernel with Machine Learning Support. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 159–173. https://doi.org/10.1145/3318464.3380575

[49] Alekh Jindal, K Venkatesh Emani, Maureen Daum, Olga Poppe, Brandon Haynes, Anna Pavlenko, Ayushi Gupta, Karthik Ramachandra, Carlo Curino, Andreas Mueller, et al. 2021. Magpie: Python at speed and scale using cloud backends. In *CIDR*.

[50] Konstantinos Karanasos, Matteo Interlandi, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Doris Xin, Supun Nakandala, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino. 2020. Extending Relational Query Processing with ML Inference. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p24-karanasos-cidr20.pdf

[51] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. In-Database Learning with Sparse Tensors. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, Jan Van den Bussche and Marcelo Arenas (Eds.). ACM, 325–340. https://doi.org/10.1145/3196959.3196960

[52] Jonathan Lajus and Hannes Mühleisen. 2014. Efficient data management and statistics with zero-copy integration. In *Conference on Scientific and Statistical Database Management, SSDBM '14, Aalborg, Denmark, June 30 - July 02, 2014*, Christian S. Jensen, Hua Lu, Torben Bach Pedersen, Christian Thomsen, and Kristian Torp (Eds.). ACM, 12:1–12:10. https://doi.org/10.1145/2618243.2618265

[53] Tianyu Li, Matthew Butrovich, Amadou Ngom, Wan Shen Lim, Wes McKinney, and Andrew Pavlo. 2020. Mainlining Databases: Supporting Fast Transactional Workloads on Universal Columnar Data File Formats. *Proc. VLDB Endow.* 14, 4 (2020), 534–546. https://doi.org/10.14778/3436905.3436913

[54] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. 2017. MLog: Towards Declarative In-Database Machine Learning. *Proc. VLDB Endow.* 10, 12 (2017), 1933–1936. https://doi.org/10.14778/3137765.3137812

[55] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (2020), 3461–3472. https://doi.org/10.14778/3415478.3415568

[56] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.* 17 (2016), 34:1–34:7. http://jmlr.org/papers/v17/15-237.html

[57] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 561–577. https://www.usenix.org/conference/osdi18/presentation/nishihara

[58] The pandas development team. 2008-2021. pandas.read_sql. https://pandas.pydata.org/docs/reference/api/pandas.read_sql.html. Accessed: 2022-01-27.

[59] The pandas development team. 2020. *pandas-dev/pandas: Pandas.* https://doi.org/10.5281/zenodo.3509134

[60] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[61] Devin Petersohn, William W. Ma, Doris Jung Lin Lee, Stephen Macke, Doris Xin, Xiangxi Mo, Joseph Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya G. Parameswaran. 2020. Towards Scalable Dataframe Systems. *Proc. VLDB Endow.* 13, 11 (2020), 2033–2046. http://www.vldb.org/pvldb/vol13/p2033-petersohn.pdf

[62] R Core Team. 2021. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org/

[63] Mark Raasveldt and Hannes Mühleisen. 2017. Don't Hold My Data Hostage - A Case For Client Protocol Redesign. *Proc. VLDB Endow.* 10, 10 (2017), 1022–1033. https://doi.org/10.14778/3115404.3115408

[64] Mark Raasveldt and Hannes Mühleisen. 2020. Data Management for Data Science - Towards Embedded Analytics. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p23-raasveldt-cidr20.pdf

[65] Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*. Citeseer.

[66] Maximilian E. Schüle, Matthias Bungeroth, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. MLearn: A Declarative Machine Learning Language for Database Systems. In *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning, DEEM@SIGMOD 2019, Amsterdam, The Netherlands, June 30, 2019*, Sebastian Schelter, Neoklis Polyzotis, Stephan Seufert, and Manasi Vartak (Eds.). ACM, 7:1–7:4. https://doi.org/10.1145/3329486.3329494

[67] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. 2010. Hive - a petabyte scale data warehouse using Hadoop. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras (Eds.). IEEE Computer Society, 996–1005. https://doi.org/10.1109/ICDE.2010.5447738

[68] Itamar Turner-Trauring. 2021. Loading SQL data into Pandas without running out of memory. https://pythonspeed.com/articles/pandas-sql-chunking/. Accessed: 2022-01-27.

[69] Jinze Wu Yizhou Chen Nick Zrymiak Changbo Qu Lampros Flokas George Chow Jiannan Wang Tianzheng Wang Eugene Wu Qingqing Zhou Xiaoying Wang, Weiyuan Wu. 2021. [Technical Report] ConnectorX: Accelerating Data Loading From Database to Dataframe. http://raw.githubusercontent.com/sfu-db/connector-x/main/assets/Technical_Report__ConnectorX.pdf.

[70] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf