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

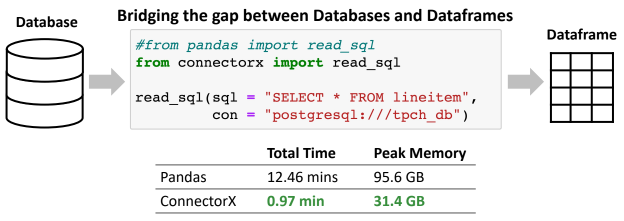
Data is often stored in a database management system (DBMS) but dataframe libraries are widely used among data scientists. An important but challenging problem is how to bridge the gap between databases and dataframes. To solve this problem, we present ConnectorX, a client library that enables fast and memory-efficient data loading from various databases (e.g.,PostgreSQL, MySQL, SQLite, SQLServer, Oracle) to different dataframes (e.g., Pandas, PyArrow, Modin, Dask, and Polars). We first investigate why the loading process is slow and why it consumes large memory. We surprisingly find that the main overhead comes from the client-side rather than query execution and 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.

1 INTRODUCTION

Dataframe libraries such as Pandas [42], Dask [48], and Modin [44] are widely used among data scientists for data manipulation and analysis. In contrast, enterprise environments often store their data in database management systems (DBMSs). 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 [2–5, 37], which easily leads to out-of-memory errors or performance degradation. Therefore, bridging the gap between databases and dataframes is of great interest to both academia and industry [29, 34, 37, 53].

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) .*

[*] Both authors contributed equally to this research.



```
#from pandas import read_sql
from connectorx import read_sql

read_sql(sql = "SELECT * FROM lineitem",
         con = "postgresql:///tpch_db")
```

	Total Time	Peak Memory
Pandas	12.46 mins	95.6 GB
ConnectorX	0.97 min	31.4 GB

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

This issue has plagued the data science community for a long time [2–5]. The research community has sought to "push the code to the data" by translating dataframe APIs into SQL [29, 34]. However, these systems are still at an early stage and only support a small fraction of the hundreds of e.g., Pandas DataFrame functions. Therefore, pulling data out of the DBMS remains the dominant approach. New DBMSs like Lakehouse [53] and DuckDB [47] use a common memory format between the execution system and client library to minimize loading costs, however this requires switching to a new DBMS, which is costly and infeasible for legacy applications.

This focus of this paper is to develop an efficient data loader for dataframes (i.e., `read_sql`) that is easily compatible with both existing client data science libraries and legacy DBMSs. Note that data scientists need only change a single line of client code to enjoy the benefits (see Figure 1).

Although there are some existing efforts in this direction, each only offers a partial solution [6, 44, 48, 52]. Chunking [52] reduces the memory pressure by loading data one chunk at a time, but does not reduce the overall load time. Partitioning the query into multiple subqueries loaded in parallel only partially reduces the runtime and does not address memory pressure [44, 48]. Other libraries like Turbodbc [6] tackle both memory and runtime issues, but are limited to a single driver (ODBC), suffer under an inefficient ODBC driver implementation (PostgreSQL), and do not leverage query partitioning.

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). As part of implementing ConnectorX, we sought to address four major questions.

First, where are the actual data loading bottlenecks? We profile the Pandas `read_sql` implementation (due to its popularity). We find that the runtime can be split into two parts: 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? To do so, 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 recieved from the network into objects in memory. The workflow executes in a pipelined fashion, and combines parallel execution, string allocation optimizations, and an efficient data representation.

Third, are current query partitioning techniques as good as they can get? Parallelization via query partitioning is the dominant way to reduce query execution (and potentially parallel loading) runtimes. Existing techniques partition the query on the client [44, 48]—for instance "`SELECT * FROM lineitem`" may be split into `SELECT * FROM lineitem WHERE l_orderkey < 1,500,000` and `SELECT * FROM lineitem WHERE l_orderkey ≥ 1,500,000`. The approach is popular since it is purely client-side, and ConnectorX adopts it as well. Unfortunately, we find that it introduces extra user burden (i.e., figuring out how to specify a proper range partitioning scheme), load imbalances, wasted server resources, and data inconsistencies. Thus, we study server-side result partitioning, where the DBMS directly partitions the query result and transfers them to the Pandas client in parallel. We prototype and demonstrate the benefits using PostgreSQL, and advocate DBMS vendors to add this support in the future.

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 100K+ downloads and 540+ Github stars within ten months. It has been applied to extracting data in ETL [7] and loading ML data from DBMS [8]. It has also be integrated into popular open source projects such as Polars [14] and DataPrep [11]. 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 [9]. Further, our experiments 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×.

In summary, our paper makes the following contributions:

(1) We perform an in-depth empirical analysis of the `read_sql` function in Pandas. We surprisingly find that the main overhead for `read_sql` comes from the client-side instead of query execution and data transfer.
(2) We design and implement ConnectorX that greatly reduces the overhead of `read_sql` with no requirement to modify existing database servers and client protocols.
(3) We propose a carefully designed architecture and interface to make ConnectorX easy to extend, and design a DSL to simplify the type mapping from databases to dataframes.
(4) We identify the issues of client-side query partitioning, and propose server-side result partitioning and implement prototype systems to address these issues.
(5) We conduct extensive experiments to evaluate ConnectorX and compare it with popular libraries. The results show that ConnectorX significantly outperforms existing solutions.

The remainder of this paper is organized as follows. We review related work in Section 2. We perform an in-depth empirical analysis of `read_sql` in Section 3, and propose ConnectorX in Section 4. Section 5 dives into the topic of query partitioning. We present evaluation results in Section 6, and conclude our paper in Section 7.

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.

Data Management for ML. ML tasks may need to access and manage raw input or intermediate data in auxiliary format. Data lake solutions [13, 24, 50] are usually adopted in this situation. Lakehouse [53] proposes a new architecture that combines the key benefits of data lakes and data warehouses. Recently, there is also an emerging trend of building ML-specific data versioning and feature store systems [10, 15, 30, 32, 40], which are developed to standardize and manage model features and workflows.

On the other hand, accessing these data from external tools is notoriously slow [34, 37, 53]. Previous work [46] shows that existing wire protocols suffer from redundant information and expensive (de)serialization, and thus propose a new protocol to tackle these issues. The same authors further develop an embedded analytical system DuckDB [47] that can avoid the bottlenecks of result set serialization and value-based APIs by making DBMS and analytic tools in the same address space. Li et. al [37] adopts Flight [27] to enable zero-copy on data export in Arrow IPC format. However, these solutions require users to modify the source code of a database system or switch to a new database system like DuckDB. Unlike these approaches, ConnectorX directly leverages existing DBMSs and client drivers, and achieves the maximum speed up within the current implementations.

SQL-Python Integration. ML tools usually adopt dataframes [26, 42, 44, 45, 48] as the abstraction for data manipulation. Data scientists are in general more familiar with dataframe operations, so they usually choose to transfer the complete data from databases to the client machine and process it using Python. To avoid moving data out of DBMS, some systems [23, 33, 36, 38, 49] try to run ML code inside database engines. Ibis [1] aims to convert dataframe operations to SQL queries and run them on a connected database. Declarative dataframe APIs [20, 39] are proposed to combine relational and procedural processing, which also allows cross-optimization between ML and database operators and has been studied by recent works [25, 29, 35]. ConnectorX complements these solutions and allows data scientist to efficiently move data out of DBMS to conduct sophistical analysis and build ML models using the Python data science ecosystem.

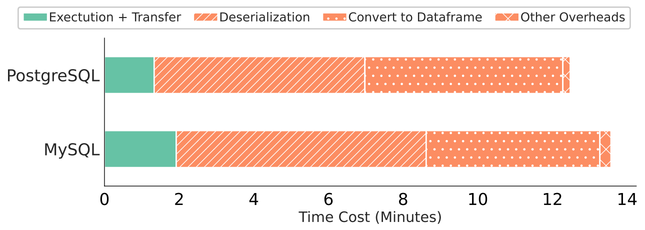


Figure 2: Time break down of `Pandas.read_sql`. (Orange parts happen on the client side.)

3 AN ANATOMY OF PANDAS.READ_SQL

In this section, we take an in-depth look at `Pandas.read_sql` [42]. There are other libraries that also provide the `read_sql` functionality and we will discuss and compare with them in Section 6. We study two questions: i) where does the time go? ii) where does the memory go? To answer them, we design and conduct an experiment between two AWS EC2 instances (r5.4xlarge, network bandwidth: 10 Gbit/s) using TPC-H benchmark (SF=10). A DBMS (PostgreSQL or MySQL) is deployed on one instance and `Pandas.read_sql` is executed on another instance to load the `lineitem` table (7.2 GB in CSV) from the DBMS. In the following, we will discuss our findings from breaking down the time and memory usage of this experiment.

3.1 Where Does the Time Go?

Under the hood, `read_sql` relies on driver libraries following the Python DB-API [28] 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 [31] 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. It means that accelerating query execution or compressing the data for wire transfer [46] is less effective in speeding up `read_sql` in this case. For example, on PostgreSQL, the query execution and data transfer only took less than 2 minutes, but the client side took more than 10 minutes (i.e., over 85% of the total running time). 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 surprising finding is that `read_sql` executes each step sequentially for PostgreSQL and MySQL by default [21][1]. 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.

[1]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.

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[2]	99.1GB

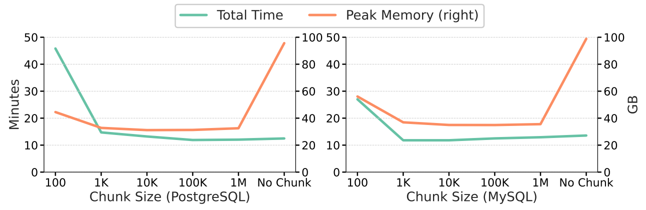


Figure 3: Time and memory change by varying chunk size.

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. This high memory requirement 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, `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 [41, 52] 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 final dataframe size when we set the chunk size within a certain

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

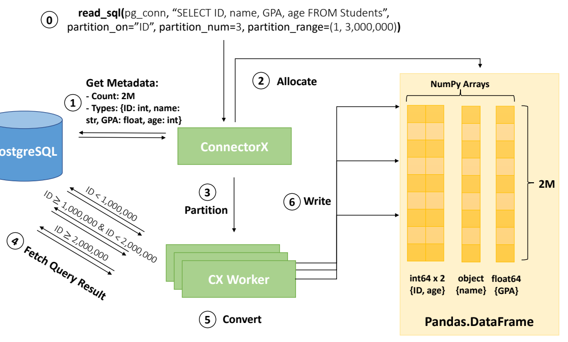


Figure 4: Workflow of ConnectorX.

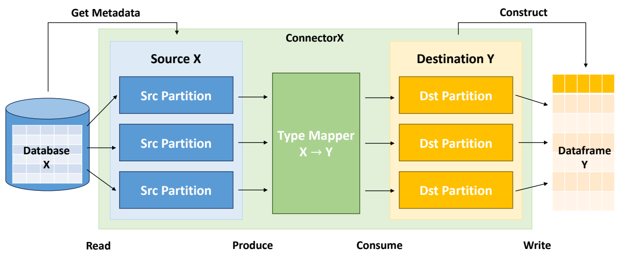


Figure 5: Overall architecture of ConnectorX.

range (e.g. 1K to 1M). However, chunking 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.

3.3 Opportunities

Through an in-depth analysis of `read_sql`, we identify four opportunities to improve the performance: 1) Study how to optimize the client side of `read_sql` because it is the main bottleneck; 2) Explore how to change the execution model of `read_sql` from sequential to parallel; 3) Investigate how to reduce data representation overhead; 4) Think about how to minimize the number of data copies.

4 CONNECTORX

In Section 4.1, we present system workflow and discuss how we leverage the above opportunities to improve the performance of `read_sql` from PostgreSQL to Pandas.DataFrame. ConnectorX can be easily extended to support a large number of databases and dataframes. We discuss how the system is architected and designed to achieve this goal in Section 4.2.

4.1 How to Speed Up?

Overall Workflow. Like 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 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 (ID $<$ 1,000,000; ID $\in [1{,}000{,}000, 2{,}000{,}000)$; ID $\geq$ 2,000,000) 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 Modin [44] and Dask [48]. We will discuss it further 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 6.2). 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.

[3]For complex queries, we can use nested queries to partition their query results. Suppose Q = "`SELECT * FROM Students, Courses GROUP BY ID`". Then, q_1 = `SELECT * FROM (SELECT * FROM Students, Courses GROUP BY ID) AS T WHERE ID < 1,000,000`.

```
pub trait Source {
    // get number of rows of query result
    fn result_rows(&mut self) -> usize;
    // get column names and types of query result
    fn fetch_metadata(&mut self) -> (Vec<String>, Vec<Self::TypeSystem>);
    // get min and max value of partition column
    fn get_partition_range(&mut self) -> (i64, i64);
    // partition a source
    fn partition(self) -> Vec<Self::SourcePartition>;
}
pub trait SourcePartition {
    // generate result of next cell
    fn produce<T>(&mut self) -> T;
}
```

Figure 6: Simplified Source interfaces (*trait* [19] in Rust is similar to *interface* in other languages).

For example, 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 approach, ConnectorX keeps the string bytes temporarily in memory and creates Python strings in batches. Therefore, instead of acquiring the GIL 100 times, ConnectorX only needs to do it once. 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.

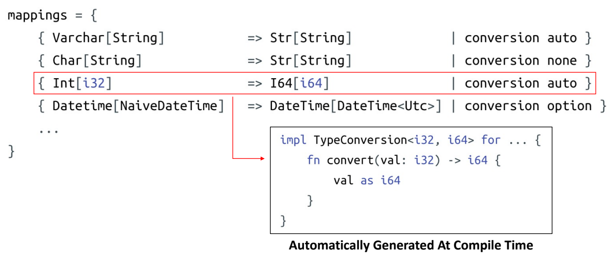
Efficient Data Representation. The limitation of Python shown in Section 3.2 indicates that a more efficient data representation is needed. Therefore, we decide to use a native programming language to implement ConnectorX. We choose Rust since it provides efficient performance and guarantees memory safety. 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 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?

In this section, we discuss how to architect and design ConnectorX to support various kinds of database systems and dataframes.

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, including both metadata and query results. To support parallel execution, 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



```
mappings = {
    { Varchar[String]          => Str[String]           | conversion auto }
    { Char[String]             => Str[String]           | conversion none }
    { Int[i32]                 => I64[i64]              | conversion auto }
    { Datetime[NaiveDateTime]  => DateTime[DateTime<Utc>] | conversion option }
    ...
}
```

```
impl TypeConversion<i32, i64> for ... {
    fn convert(val: i32) -> i64 {
        val as i64
    }
}
```

Figure 7: Example of defining type mapping in ConnectorX.

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 (adding a 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. ConnectorX leverages existing client drivers to implement the functionality, which is the same as other libraries. However, other libraries require client drivers provide a certain API. For example, Pandas needs the input connection object to implement Python DB-API and Turbodbc only works on ODBC drivers. Unlike these approaches, ConnectorX has no requirement on the API of a client driver, which gives it the flexibility to choose the fastest client driver for each DBMS. ConnectorX abstracts the needed functionalities into a set of succinct interfaces. Adding a new Source only requires implementing these interfaces with an existing driver.

Figure 6 shows the non-trivial functionalities that are required to be supported for each Source in ConnectorX. Here we use simplified pseudo code in order to make the purpose of each interface more clear. In general, adding a new Source needs to define a Source class and a SourcePartition class. The Source class is used to initiate the connection information and collect information for preparation, including querying the number of rows (`result_rows`), column names and types (`fetch_metadata`) and range of the partition column (`get_partition_range`) of the query result. The SourcePartition is derived from the Source class with the same connection information. It executes the partitioned query and defines how to read and parse data (`produce`) for each supported type. We can see that the interface is succinct because it only contains necessary methods that are closely dependent on the database implementation.

(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. For example INT8, CHAR, and DATE in PostgreSQL can be converted to int64, object, and datetime64 in Pandas, and int64, 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 by implementing the corresponding conversion function. However, this approach

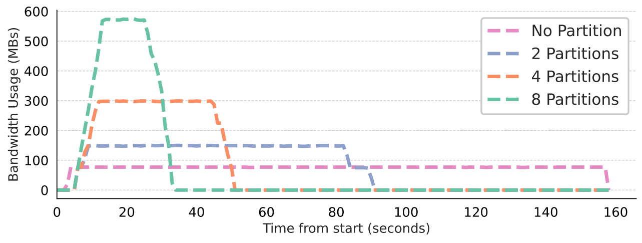


Figure 8: Network utilization by varying # partitions.

will lead to two pain points. First, there will be a lot of trivial code for types with the same physical representation. It is 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 already automatically supported (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, and we want to support five DBMSs and two dataframes. Then there will be 150 ($15 \times 5 \times 2$) type conversion functions that need to be implemented and maintained.

In order to mitigate the aforementioned issues, ConnectorX defines a succinct domain specific language (DSL) to help the developers define the type mappings, leveraging the modern macro support in Rust [18]. Figure 7 shows an example of the DSL, in which the mapping relations are defined in `mappings`. Each line consists of three parts: the 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 String to String and i32 to i64, a developer could specify the conversion as `auto` and ConnectorX will automatically generate the corresponding conversion functions like the example shown in Figure 7. `option` is used for non-trivial conversions, for which the developer is required to implement the corresponding type 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. It has helped shorten code related to type mapping by 97% (from 37k to 1k lines of code).

5 QUERY PARTITIONING

In this section, we first discuss client-side query partitioning and then propose server-side result partitioning.

5.1 Client-Side Query Partitioning

Client-side query partitioning partitions a query into multiple subqueries on the client side and query them independently in parallel. This approach can accelerate `read_sql` because it utilizes the high network bandwidth and CPU resource more efficiently. We show the network utilization of ConnectorX by varying the number of partitions in Figure 8. It is clear that No Partition (single connection) cannot saturate the network bandwidth at all. With more

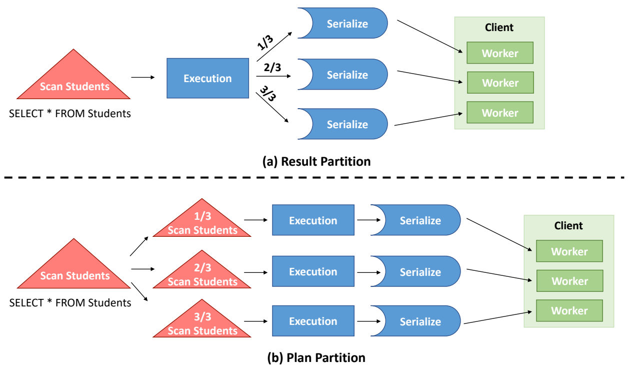


Figure 9: Illustration of server-side partition implementations on example "`DECLARE example CURSOR FOR SELECT * FROM Students INTO 3`".

connections fetching data in parallel, bandwidth could be more sufficiently leveraged and thus leading to better end-to-end performance (`read_sql` finishes when bandwidth usage drops to 0).

Client-side query partitioning can directly work with an existing DBMS. This is a big advantage. However, the downsides are: (1) *User Burden.* To enable query partitioning, the user has to take extra effort to specify a range partitioning scheme over the query result table. (2) *Load Imbalance.* If the query result table is not evenly partitioned, stragglers may arise, hurting overall performance. (3) *Data Inconsistency.* Since multiple subqueries are sent to the server from independent sessions, their results may derive from different snapshots of the database. (4) *Wasted Resource.* Different subqueries may share the same costly sub-plan (e.g., full-table scan). Since the DBMS processes each query independently, the sub-plan may be repeatedly executed for many times, thus wasting database resources.

5.2 Server-Side Result Partitioning

So far we have considered the situation when the underlying DBMS cannot be modified. If the underlying DBMS could be modified, then 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. 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.

Implementation Alternatives. Naturally, there are many approaches to support server-side result partitioning. We discuss two alternatives in this paper and implement a prototype for each approach on PostgreSQL. Figure 9 illustrates the differences between the two approaches on example query: `"DECLARE example CURSOR FOR SELECT * FROM Students INTO 3"`.

(1) Result Partition. A straight forward solution is to directly partition the query result. As illustrated in Figure 9 (a), the plan for the query will be generated and executed exactly the same with issuing the query without partitioning. The difference is that the execution result will be evenly distributed to three connections instead of one.

Specifically, we can generate and temporarily store the query plan for "SELECT * FROM Students" when handling the declare cursor statement. Later there will be three fetch query issued to the database. The first arrived fetch query will be able to access the plan by looking up the name of the cursor. It will then execute the plan and distribute the result tuples to its own and other two processes. The later arrived fetch queries will register themselves to the first process, serialize the arrived tuples and send them back to their own connections.

In our prototype implementation, we leverage the dynamic shared memory and message queue mechanism in PostgreSQL to make the first process send tuples back to its own client as well as to other registered processes in turn.

Result Partition evenly distributes the query results to each connection without the need of any extra information. And there will be no data inconsistency nor redundant execution since it is essentially equal to executing the entire query in a single connection process. Furthermore, it does not require any modification to the query optimizer or executor, and can also leverage the existing parallel execution mechanism in database. Therefore it can be easily adopted by databases in general. However, there will be extra synchronization overhead. As we will discuss later in Section 5.3, this overhead could be non-trivial and become worse with more number of partitions.

(2) Plan Partition. To reduce the synchronization overhead, a possible solution is to partition the query plan instead of the query result and handle each plan independently as shown in Figure 9 (b).

Using the same example. In addition to generate the plan for the original query, which needs to scan the entire Student table, we further split it into three individual partitioned plans that with similar cost (e.g. each scans one third of the pages). Later, each fetch query can execute the corresponding partitioned plan using the partition_id on the same snapshot of the data to ensure data consistency. The procedure of partition the query plan may leverage the existing parallel query execution approach. The difference is that there will be no synchronization mechanism (e.g. Gather node in PostgreSQL) between the partitioned plans.

In Section 5.3, we use simple SELECT * query to show the efficacy of this approach and leave the design of partitioning more complex operators (e.g. Join) to future work. In our prototype, we partition a plan with a sequential scan operator by evenly splitting the pages and assigning to the derived plans.

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**

The advantage of Plan Partition is that all fetch query processes run independently without synchronization needs, and therefore can be very efficient. However, not every plan can be partitioned directly to the leaf node (e.g. ones with Aggregation operator), which means that for these queries the database may still need to execute the same sub-plan repeatedly. Also it requires non-trivial modification to the query optimizer.

5.3 Prototype Evaluation

In this subsection, we first show the efficacy of Plan Partition, and then discuss the overhead we found in our implementation of Result Partition. We use Arrow as the destination dataframe to focus on the server side since it does not require the extra COUNT query in advance.

Plan Partition. 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 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 only 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 required increases along with the number of partitions specified (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, the server-side Plan Partition shows the same statistics with the baseline no matter how many partitions it has. 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, Plan Partition 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.

Result Partition. On the other hand, our implementation of Result Partition shows worse performance than both client-side and Plan Partition approaches. Compared with the no-partition baseline, although it becomes 37% faster using 2 partitions, the performance shows 63% and 82% degradation under 4 and 16 partitions, respectively. With the help of *pg_top* [51] and *perf* [43], we narrow down the bottleneck to the inter-process communication for distributing tuples through message queues. However, we think this overhead should be mitigated in databases adopting multi-thread architecture such as MySQL.

6 EVALUATION

We conduct extensive experiments to evaluate ConnectorX.

6.1 Experimental Setup

Datasets & Workloads. (1) *TPC-H [17].* We generate the TPC-H benchmark dataset by setting the scale factor to 10. We select the entire lineitem table, which consists of around 60M rows and 16 columns with types of INTEGER, DECIMAL, DATE and VARCHAR. The table is approximately 7.2GB in CSV format. For experiments that involve query partitioning, we use column l_orderkey as the partitioning column, which is evenly distributed and thus the cardinality of each subquery is similar. We also generate 22 SPJ queries to test more complex queries, with the fetched result size ranging from 100K to 59M. (2) *DDoS [12].* 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 [42] is one of the most widely used library among data scientists. It provides the read_sql function that could get a SQL query's result in a Pandas dataframe. Underlying, Pandas relies on drivers (e.g. sqlite3 [16] and SQLAlchemy [22]) that follow the Python DB-API [28] to connect and access to databases.

Dask [48] is an open source library providing scalable analytics in Python by partitioning data into chunks, and runs computations over chunks in parallel. Its read_sql_table function supports partitioning the query on a given column linearly between min/max values (the same as ConnectorX introduced in Section 4.1), and then invoke Pandas.read_sql for each partitioned query in parallel.

Modin [44] is an extremely light-weight parallel Dataframe that provides seamless integration and compatibility with existing pandas code. Similar to Dask, it also wraps the Pandas.read_sql function and supports partitioning the query on the client side. We use the Dask engine for Modin in our experiment. Modin does not support DBMS-A due to the hard coding in synthesizing the queries.

Turbodbc [6] is a Python module that accesses relational databases via the Open Database Connectivity (ODBC) interface. It exploits buffering to speed up result retrieving and conversion. Unlike other libraries that could simply import the library for usage, Turbodbc requires the user to setup the ODBC configuration on their machine. We follow Turbodbc's documentation to configure the environment for PostgreSQL, MySQL and DBMS-A. Since Turbodbc does not support Pandas destination directly, we convert its NumPy array result to Pandas eventually to ensure a fair comparison.

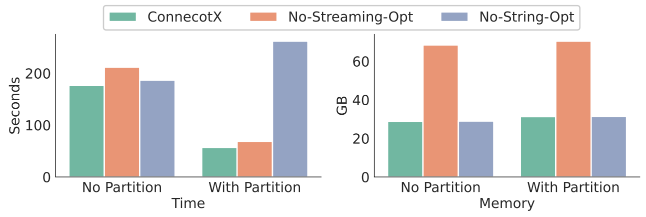


Figure 10: Ablation study.

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).

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. ConnectorX needs to issue an extra query to get the min and max values of the partition column during the preparation phase. 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 in Section 6.3.

6.2 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 8 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 10. 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. While the impact of string allocation optimization varies in different numbers of partitions, it slows down the process by only 6% under the no partition setting. However, it becomes 4.6× slower with partitioning. This is because of the overhead in acquiring GIL during string allocation in Python. With more partitions running in parallel, there will be more contention on the GIL, thus slowing down the process.

In terms of peak memory usage, we can see that applying partitioning has little impact on memory consumption. Without the streaming workflow optimization, No-Streaming-Opt needs 2.3× more memory due to the large intermediate results, however, although it needs 70.4GB of memory, it still saves more than 20GB of memory comparing to the 95.6GB peak memory usage of Pandas's

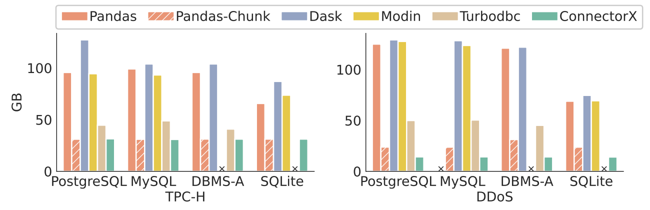


Figure 11: Memory comparison on four database systems. ("×" is placed if a method does not support the database or cannot handle the large query result.)

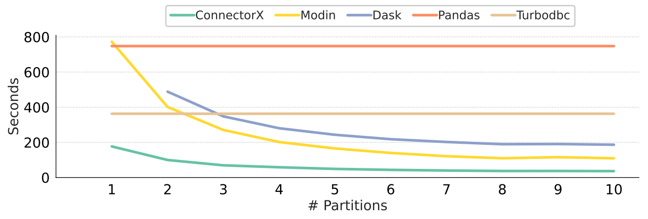


Figure 12: Speed comparison on PostgreSQL by varying the number of partitions.

batch solution shown in Table 1. This validates the effectiveness of using Rust in terms of data representation.

6.3 Performance Comparison

We compare ConnectorX with four baselines when running `read_sql` to fetch the same query result into a Pandas.DataFrame. Notice that we do not convert the corresponding dataframe result from Dask and Modin into Pandas.DataFrame since they are targeting on replacing Pandas easily. We first load the TPC-H lineitem and DDoS tables, and then test how ConnectorX works under different network conditions and with more complex queries. We also show the performance comparison when loading the result to Apache Arrow format.

Memory Comparison. Figure 11 evaluates the peak memory usage of loading the entire TPC-H lineitem and DDoS tables. For the approaches that support query partitioning (Modin, Dask, and ConnectorX), we show the result of with partitioning, which is usually no less than without partitioning. Pandas-Chunk enables chunking for Pandas (chunk size: 10K according to Figure 3).

On TPC-H, we can see that the memory consumptions of ConnectorX and Pandas-Chunk are almost the same on all DBMS. Their peak memory values are consistently 3× less than Pandas on the three 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 is 13× larger in Python objects than in the final dataframe. Another interesting finding is that compared to 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.

Speed Comparison. Next, we compare the speed of each method. We first show the speed comparison under high bandwidth network setting (10Gbit/s, except for SQLite, which can only reside on the same client instance) in Figure 13. In order 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). Here we do not show the result of chunking since which cannot improve the speed.

(1) No Partitioning. In almost all cases, ConnectorX performs the best without query partitioning. It outperforms Pandas by 4.2×, 5.2×, 3.0× and 2.3× on PostgreSQL, MySQL, DBMS-A and SQLite respectively for TPC-H, and 7.1×, 6.0× and 5.2× on PostgreSQL, DBMS-A and SQLite for DDoS (Pandas fails to fetch the entire DDoS table from MySQL). Modin and Dask have the extra overhead in transferring the result from worker processes, and thus is slower than Pandas without parallelism. In addition, they could not finish in many cases when no partition is applied due to the out-of-memory issue. Turbodbc is the fastest among all baselines. Compared with ConnectorX, it 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. Unlike Turbodbc, ConnectorX has no requirement on the driver and thus is more flexible in terms of switching to a faster driver anytime.

(2) With Partitioning. ConnectorX is the fastest one in all our experiments with partitioning. Only Dask and Modin support query partitioning among baselines. To make the comparison more clear, we copy the results of Pandas and Turbodbc without query 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 and it becomes the fastest baseline approach on TPC-H. 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.

We also compare ConnectorX with Dask and Modin by varying the number of partitions on TPC-H using PostgreSQL. The result is shown in Figure 12. We add the result of Pandas and Turbodbc to make the comparison clear and complete. We can see that with more partitions, the performances of ConnectorX, Dask and Modin all get improved first and then become stable. ConnectorX stays the fastest in all scenarios and outperforms Modin and Dask by 3× and 5× respectively when the time converges.

(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 baseline. The speedups are shown in Table 3. ConnectorX-NoPart represents the result of ConnectorX when no

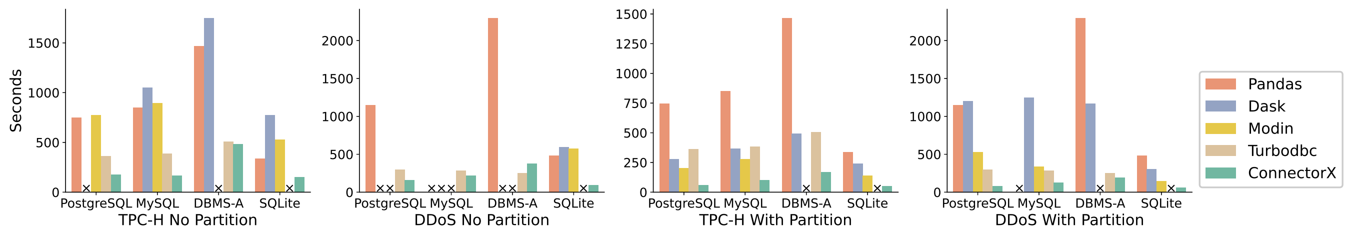


Figure 13: Speed comparison on four database systems. ("×" is placed if a method does not support the database or cannot handle the large query result.)

partition is applied, and both Modin and Dask leverage partitioning. It is clear that ConnectorX remains the fastest in all environments 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. In addition,ConnectorX and ConnectorX-NoPart perform similarly when the bandwidth is 200Mbit/s since a single thread could be enough when the bandwidth is low.

(4) SPJ Queries. We evaluate ConnectorX using more complex queries. We consider the queries with joins and predicates because adding joins will increase the server's query execution time and adding predicates will affect the data transfer time between the server and the client. By considering a wide variety of queries, we can have a better understanding of how well ConnectorX will perform in different scenarios. 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 on 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

[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

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**×

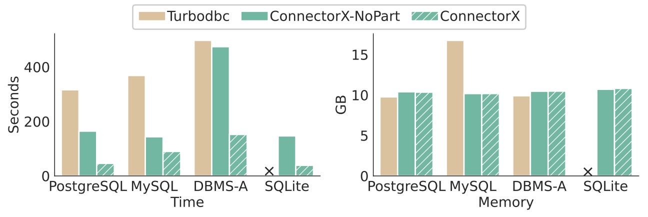


Figure 14: Performance comparison on four database systems when deriving Arrow format. ("×" is placed if a method does not support the database or cannot handle the large query result.)

result sets), however, ConnectorX brings less benefit and sometimes it can be even slower due to the overhead in fetching metadata.

(5) Arrow. Finally we evaluate ConnectorX with existing methods when deriving dataframes other than Pandas.DataFrame. ConnectorX currently implements two main dataframe formats: Pandas and Arrow. For Python users, we support returning other dataframes including Dask and Modin by converting from Pandas, and Polars is derived from Arrow internally, and Polars by deriving from intermediate Arrow format. Here we show the comparison between ConnectorX with existing methods when the destination format is Apache Arrow in Figure 14 using TPC-H lineitem table.

Among the four baselines, only Turbodbc supports returning Arrow directly. Although other libraries could also derive the Arrow result by converting from the corresponding original dataframe, the performance would be much worse. Therefore, here we only compare ConnectorX with Turbodbc on four DBMSs. In terms of speed, the gap between ConnectorX and Turbodbc on Arrow format is very similar to Pandas.DataFrame. ConnectorX is faster on all databases. It outperforms Turbodbc by up to 2.5× and 7.0× without and with partitioning respectively. As for peak memory, ConnectorX needs approximately 10GB of memory in all cases,

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**×
Four Partitions	**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**×

while Turbodbc is similar on PostgreSQL and DBMS-A but needs around 67% more on MySQL.

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 also 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 (1) optimizations applied in ConnectorX are indeed effective at boosting the performance, (2) ConnectorX significantly outperforms Pandas, Dask, Modin and Turbodbc in terms of both speed and memory usage under different scenarios.

REFERENCES

[1] 2014-2021. Ibis: Write your analytics code once, run it everywhere. http://ibis-project.org. Accessed: 2022-01-27.

[2] 2016. pandas read_sql is unusually slow. https://stackoverflow.com/questions/40045093/pandas-read-sql-is-unusually-slow. Accessed: 2022-01-27.

[3] 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.

[4] 2017. Program (Time) Bottleneck is Database Interaction. https://stackoverflow.com/questions/44154430/program-time-bottleneck-is-database-interaction. Accessed: 2022-01-27.

[5] 2017. Use Turbodbc/Arrow for read_sql_table. https://github.com/pandas-dev/pandas/issues/17790. Accessed: 2022-01-27.

[6] 2017-2021. Turbodbc - Turbocharged database access for data scientists. https://turbodbc.readthedocs.io/en/latest/. Accessed: 2022-01-27.

[7] 2021. ConnectorX for ETL workload. https://github.com/sfu-db/connector-x/discussions/133. Accessed: 2022-01-27.

[8] 2021. ConnectorX for ML feature fetching. https://github.com/sfu-db/connector-x/issues/140#issuecomment-948918848. Accessed: 2022-01-27.

[9] 2021. ConnectorX integrates with dataframe system. https://pola-rs.github.io/polars-book/user-guide/howcani/io/read_db.html. Accessed: 2022-01-27.

[10] 2021. Data Vesion Control (DVC). https://dvc.org/. Accessed: 2022-01-27.

[11] 2021. DataPrep: The easiest way to prepare data in Python. https://dataprep.ai/. Accessed: 2021-01-27.

[12] 2021. DDoS Dataset. https://www.kaggle.com/devendra416/ddos-datasets. Accessed: 2022-01-27.

[13] 2021. Google BigQuery. https://cloud.google.com/bigquery. Accessed: 2022-01-27.

[14] 2021. Polars: Fast multi-threaded DataFrame library in Rust and Python. https://github.com/pola-rs/polars. Accessed: 2022-01-27.

[15] 2021. Serve your features in production. https://feast.dev/. Accessed: 2022-01-27.

[16] 2021. sqlite3 — DB-API 2.0 interface for SQLite databases. https://docs.python.org/3/library/sqlite3.html. Accessed: 2022-01-27.

[17] 2021. TPC-H Homepage. http://www.tpc.org/tpch. Accessed: 2022-01-27.

[18] 2022. The Rust Programming Language - Macro. https://doc.rust-lang.org/book/ch19-06-macros.html. Accessed: 2022-01-27.

[19] 2022. The Rust Programming Language - Traits: Defining Shared Behavior. https://doc.rust-lang.org/book/ch10-02-traits.html. Accessed: 2022-01-27.

[20] 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

[21] 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.

[22] Michael Bayer. 2012. SQLAlchemy. In *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks*, Amy Brown and Greg Wilson (Eds.). aosabook.org. http://aosabook.org/en/sqlalchemy.html

[23] 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

[24] 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

[25] 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

[26] The Apache Software Foundation. 2016-2021. Apache Arrow. https://arrow.apache.org/. Accessed: 2022-01-27.

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

[28] 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.

[29] 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

[30] K. Hammar and J. Dowling. 2021. Feature Store: The missing data layer for Machine Learning pipelines? https://www.hopsworks.ai/post/feature-store-the-missing-data-layer-in-ml-pipelines. Accessed: 2022-01-27.

[31] 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

[32] Jeremy Hermann and Mike Del Balso. 2017. Meet Michelangelo: Uber's Machine Learning Platform. https://eng.uber.com/michelangelo-machine-learning-platform/. Accessed: 2022-01-27.

[33] 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

[34] 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*.

[35] 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

[36] 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

[37] 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

[38] 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

[39] 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

[40] Laurel J. Orr, Atindriyo Sanyal, Xiao Ling, Karan Goel, and Megan Leszczynski. 2021. Managing ML Pipelines: Feature Stores and the Coming Wave of Embedding Ecosystems. *Proc. VLDB Endow.* 14, 12 (2021), 3178–3181. http://www.vldb.org/pvldb/vol14/p3178-orr.pdf

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

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

[43] Linux perf framework. 2020. Linux perf framework. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed: 2022-01-27.

[44] 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

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

[46] 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

[47] 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

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

[49] 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

[50] 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

[51] PostgreSQL top Project. 2020. PostgreSQL top Project. https://pg_top.gitlab.io/. Accessed: 2022-01-27.

[52] 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.

[53] 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