ConnectorX: Accelerating Data Loading From Database to Dataframe

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

Pandas is a widely used data analysis library in Python. To analyze the data stored in a database management system (DBMS), data scientists often need to use the `read_sql` function to load large datasets from DBMS to Pandas DataFrame. However, `read_sql` is notoriously slow and memory-consuming. In this paper, we present ConnectorX, a new Python 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 `read_sql` 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 a new functionality (named server-side query partitioning) that a database system can add in order to better support ConnectorX. 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

Pandas is a widely used data analysis library in Python, with a total of 1.2B downloads on PyPI and over 32.3K stars on Github as of Jan 2022. In enterprise environments, however, data is often stored in database management systems (DBMS). It is a common practice to load data into a Pandas DataFrame because it enables various downstream data science tasks, such as explore data analysis, feature engineering, and model training. This data loading process bridges the gap between Database and DataFrame, and plays an important role in the data science lifecycle.

Pandas provides a function called `read_sql` to facilitate the data loading process. `read_sql` takes as input a `conn` object that specifies a connection to a DBMS (e.g., PostgreSQL or MySQL) and a query string that describes what data to load. Figure 1 shows an example. While easy to use, `read_sql` is not only notoriously slow but also memory-consuming when loading large data [2–5, 38]. In our experimental analysis (see Section 2), we used `read_sql` to load the lineitem table (7.2 GB) of TPC-H from PostgreSQL. The total time took 12.5 mins and the peak memory reached to 95.6 GB.

The Python data science community has been suffering from this issue for a long time [2–5]. There are some recent systems that aim to avoid moving data out of the DBMS by translating Pandas APIs into SQL [30, 35]. However, these systems are still at an early stage and support a limited number of Pandas APIs. Therefore, we allow data scientists to take data out of the DBMS and focus on speeding up the `read_sql` function. This kind of approach is very easy to adopt since it only needs to change a single line of code without modifying the underlying DBMS (see Figure 1).

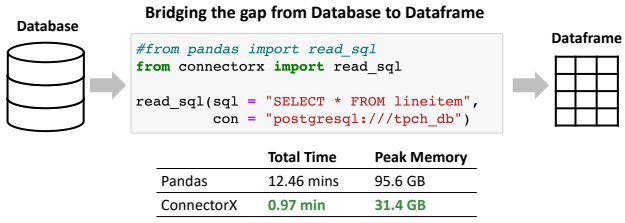

Figure 1: Speed up loading the lineitem table (7.2 GB in CSV) from database to dataframe with less memory usage.

Existing efforts of speeding up `read_sql` fall short due to various reasons [6, 45, 49, 53]. Chunking [53] loads one chunk of data at a time. While it addresses the memory issue, the running time is still too long. Modin [45] and Dask [49] accelerate `read_sql` by partitioning a query into multiple subqueries and running them in parallel. However, it does not address the memory issue, and also the running time still has a big room to improve. Turbodbc [6] is a Python library, which aims to address both memory and efficiency issues. However, it is restricted to the ODBC driver, and does not work well if a database system (e.g., PostgreSQL) have an inefficient ODBC driver. Furthermore, Turbodbc does not support the query partitioning technique adopted by Modin and Dask.

To this end, we propose ConnectorX, a new Python library that enables fast and memory-efficient data loading from various database systems (e.g., `PostgreSQL`, `SQLite`, `MySQL`, `SQLServer`, `Oracle`) to different dataframes (e.g., Pandas, PyArrow, Modin, Dask, and Polars). To design and implement such a system, we need to address a number of challenging problems.

Firstly, why is `read_sql` *slow and why does it consume large memory?* These fundamental questions have not been well explored in previous studies. To answer them, we conduct an in-depth analysis on using `read_sql` to load a large table from DBMS to Pandas in a cloud environment. The running time of `read_sql` can be roughly divided into two parts. The first part is spent on the database server side, which involves executing the query to get the table data and transferring it to the client side. The second part is spent on the Pandas client, which involves deserializing the table data and converting it to DataFrame. We surprisingly find that most of the time (over 85%) is spent on the second part (i.e., client-side), and all the intermediate results on the client-side are kept in memory. These findings suggest that the performance of `read_sql` can be significantly improved by optimizing its client-side.

[*] Both authors contributed equally to this research.

Secondly, how to build ConnectorX to achieve two design goals? The first design goal is to make data loading fast and memory-efficient. We adopt a streaming workflow to load data one batch at a time, and propose three optimization techniques: parallel execution, and string allocation optimization, and efficient data representation. While some of these optimization techniques have been adopted before, the novelty lies in the integration of these optimization techniques into one single system. The second design goal is to make the system easy to extend. Users want to use ConnectorX to load data from various DBMSs [10]. We carefully design our system architecture and interface, and leverage a programming language feature called Macro [19] to simplify the type mapping from DBMS to DataFrames. With this design, the number of lines of code can be reduced by 1-2 orders of magnitude for adding a new DBMS.

Thirdly, what are the issues of client-side query partitioning and how to implement server-side query partitioning? Query partitioning is a simple but highly effective approach to accelerate `read_sql`. Like Dask and Modin, ConnectorX uses client-side query partitioning. For example, "`SELECT * FROM lineitem`" can be split into: `SELECT * FROM lineitem WHERE l_orderkey < 1,5000,000` and `SELECT * FROM lineitem WHERE l_orderkey ≥ 1,500,000`. ConnectorX will start a thread per subquery. Each thread will issue the subquery to the DBMS, then deserialize the subquery result, and finally convert it to the dataframe. Client-side query partitioning is widely adopted because it does not need to make any modification to the DBMS. However, it also has several issues, such as extra user burden, uneven partitioning, wasted resource, and data inconsistency. This inspires us to explore server-side query partitioning: A DBMS directly generates multiple partitions of the query result and transfers them to the Pandas client in parallel. We build a system prototype using PostgesSQL and demonstrate its superiority. We hope that database vendors can consider adding the support of server-side query partitioning in the future.

We conduct extensive experiments using both synthetic and real-world datasets. The results show that ConnectorX significantly outperforms existing libraries: Pandas, Dask, Modin, and Turbodbc, when loading large data from Databases to DataFrames. In particular, it achieves 13× speed-up and 3× memory reduction compared to Pandas.

Practical Impact. Since its first release in April 2021, ConnectorX has been widely adopted by real users, with a total of 100K+ downloads and 530+ 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 other popular open source projects, such as Polars [15] and DataPrep [12]. 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].

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 leverage Macro to simplify the type mapping from Databases to DataFrames.

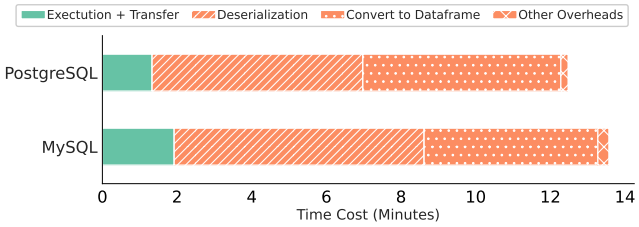

Figure 2: Time break down of Pandas.read_sql. (Orange parts happen on the client side.)

(4) We identify the issues of client-side query partitioning, and propose server-side query 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 perform an in-depth empirical analysis of `read_sql` in Section 2, and propose ConnectorX in Section 3. Section 4 dives into the topic of query partitioning. We present evaluation results in Section 5, review related work in Section 6, and conclude our paper in Section 7.

2 AN ANATOMY OF PANDAS.READ_SQL

In this section, we take an in-depth look at `Pandas.read_sql` [43]. There are other libraries that also provide the `read_sql` functionality and we will discuss and compare with them in Section 5.We observe that `read_sql` typically takes significant longer time than the actual query time and more memory than the actual query result size. It is natural to ask i) where does the time go? ii) where does the memory go?

To answer these questions, 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 in 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.

2.1 Where Does Time Go?

Under the hood, `read_sql` relies on driver libraries following the Python DB-API [29] 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 return the query result in the form of Python objects.
(3) *Convert to DataFrame*: Client converts the Python objects into NumPy [32] arrays and construct the final dataframe.

Figure 2 shows the time break down on PostgresSQL 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 [47] is less effective in speeding up `read_sql` in this case. For example, on PostgresSQL, 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

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

running time). This result suggests that we should focus on optimizing the client side, which is dominated by two data conversions: *deserialization* and *convert 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 [22][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 wait 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 wait until all Python objects are available. This kind of implementation will lead to two issues. First, all intermediate results will be temporarily kept in memory, which needs to allocate much more memory than needed as we will show in Section 2.2. Second, single thread execution cannot make full use of both network and computational resources.

2.2 Where Does 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 serialized result client received, the intermediate Python object, and the final dataframe, respectively.

We observe that the peak memory is approximately 4× as large as the size of the final dataframe, which indicates that if the final dataframe is 25 GB, the system has to provide at least 100GB of memory in order to ensure the successful running of `read_sql`. 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. During the running of `read_sql`, there are three copies of the entire data kept 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 Chunk Loading Help?. Chunking [42, 53] 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 using `read_sql`. Figure 3 shows the results. For a fair comparison, we concatenate all the intermediate dataframes to a large one in the end to represent the entire query result. "No Chunk" represents that chunking loading is not used.

We can see that chunk loading is indeed very effective in reducing memory usage. This is because that it does not hold all the intermediate results in memory anymore. The peak memory usage can become almost equal to the final dataframe size when we set the

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

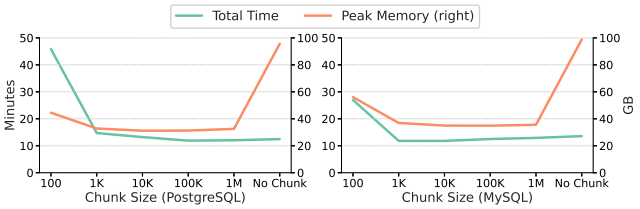

Figure 3: Time and memory change by varying chunk size.

chunk size within a certain range (e.g. 1K to 1M). However, chunk loading 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 chunk loading and handle a stream of dataframes.

2.3 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.

3 CONNECTORX

We develop ConnectorX, an open-source library to speed up `read_sql`. In Section 3.1, we present system workflow and discuss how we leverage the above opportunities to improve the performance of `read_sql`. 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 3.2.

3.1 How to Speed Up?

Overall Workflow. Like chunk loading, ConnectorX adopts a streaming workflow, where the client side 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 $q_1, q_2, \cdots, q_n$

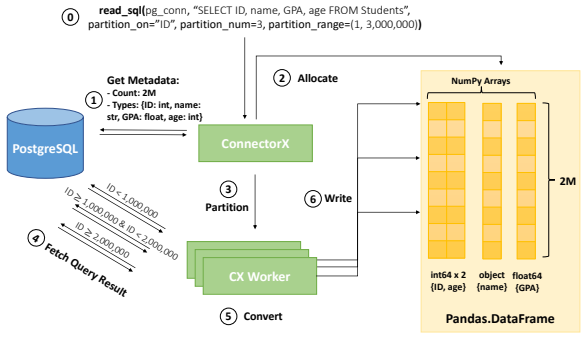

Figure 4: Workflow of ConnectorX.

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 ∈ [1,000,000, 2,000,000); ID ≥ 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 [45] and Dask [49]. We will discuss it further in Section 4.

String Allocation Optimization. ConnectorX pre-allocates the NumPy arrays in advance to avoid extra data copy. However, the buffers that the string object 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, which makes its allocation a potential bottleneck, especially when the degree of parallelism is large. To alleviate this overhead, ConnectorX constructs a batch of strings at a time while acquiring the Python Global Interpreter Lock (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.

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 batch. 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) due to the fact that 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 2.2 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,

[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`.

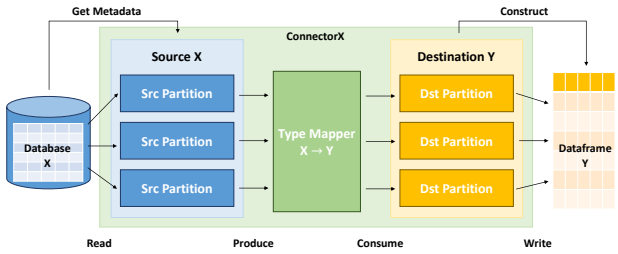

Figure 5: Overall architecture of ConnectorX.

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`.

3.2 How to Extend?

In this section, we discuss how to architect and design ConnectorX to support various databases 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 handles the job of reading and parsing 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, with each of them assigned a subquery. The Destination module is used to implement the logic of generating 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 or Destination to ConnectorX is straight forward. Due to the space limit, here we introduce how to add a new Source (adding a new Destination is similar). The non-trivial part of the code includes the following logic:

- Connecting to the new Source and supporting functionalities required by ConnectorX.
- Type mapping of the new Source to each Destination.

(1) Connection. 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

```rust
/* Source */
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);
}
pub trait SourcePartition {
    // generate result of next cell
    fn produce<T>(&mut self) -> T;
}
```

Figure 6: Simplified Source interfaces (*trait* [20] in Rust is similar to *interface* in other languages).

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

Figure 7: Example of defining type mapping in ConnectorX.

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 databases define their own type systems and physical representation for the types. 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 it is to define how to convert each type from each Source to each Destination manually by implementing the corresponding conversion function. However, this approach 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 database has 15 data types on average, and we want to support five databases 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 set of sophisticated but also intuitive *macros* to help the developers define the type mappings. Macro [19] can generate source code at compile time based on simple rules, which makes it perfectly suitable in creating highly repetitive type conversion functions. Figure 7 shows an example, in which the mapping relations are defined in `mappings`. Each line consists of three parts: the logical type and the underlying physical type in Source, the matched logical type and the physical type in 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 automatically supported, like str to str and i32 to i64, a developer could specify the conversion as `auto` and ConnectorX will automatically generate the corresponding conversion functions. `option` is used for non-trivial conversions, in which the developer is required to implement the corresponding type conversion function. To avoid repeated definitions, `none` indicates the case that the physical type pair is already handled. Using the macros makes the relation of type mapping as well as the corresponding logical and physical types clear and easy to maintain. Until now, these macros help us reduce the type mapping related code from more than 37K lines to only 1K lines.

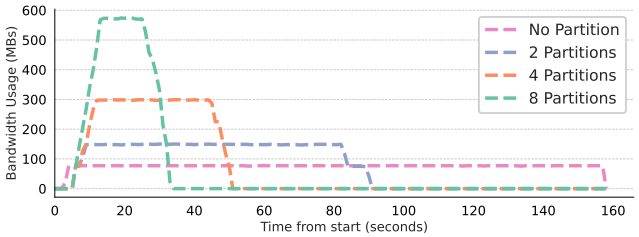

Figure 8: Network utilization by varying # partitions.

Flexibility. Existing tools require client driver provides certain API. For example, Pandas needs the input connection implements Python DB-API and Turbodbc only works on ODBC drivers. Unlike these approaches, ConnectorX does not have any requirement on the interface that the database driver provides, which makes it able to leverage any efficient driver and protocol. In fact, ConnectorX can support multiple accessing alternatives to the same Source so that users can select the one that benefits their specific scenario the most. For example, ConnectorX supports both Binary and CSV protocols for loading data from PostgreSQL. Binary protocol has a faster parsing speed while CSV protocol usually needs less bytes to transfer, which makes the later one a better choice when the network speed is slow.

4 QUERY PARTITIONING

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

4.1 Client-Side Query Partitioning

Client-side query partitioning partitions a query into multiple subqueries on the client side an query them independently in parallel. This approach can accelerate `read_sql` because it utilizes the high network bandwidth and CPU resource more sufficiently. Use bandwidth as an example. We show the network utilization of ConnectorX by varying the number of partitions in Figure 8. It is clear that No Partition (single endpoint) cannot saturate the network bandwidth at all. With more endpoints fetching data in parallel, bandwidth could be more sufficiently leveraged and thus lead to better end-to-end performance (`read_sql` finishes when bandwidth usage drop to 0).

Client-side query partitioning can directly work with an existing DBMS implementation. 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) *Uneven Partitioning.* If the query result table is not evenly partitioned, the parallelism will not be fully utilized, hurting the overall performance. (3) *Data Inconsistency.* Since multiple subqueries are sent to the server from independent sessions,

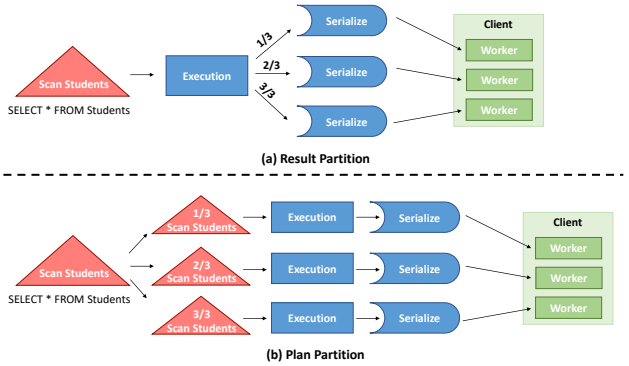

Figure 9: Illustration of server-side partition implementations on example "DECLARE example CURSOR FOR SELECT * FROM Students INTO 3".

their results may derive from different snapshots of the database. (4) *Wasted Resource.* Different subqueries may share the same costly sub-plan (e.g., scan the entire table). Since the DBMS processes each query independently, the sub-plan may be repeatedly executed for many times, thus wasting database resources.

4.2 Server-Side Query 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 result on the database server side would address the aforementioned issues. Specifically, once the DBMS receives a query, it partitions the query result into n equal partitions and provides n end points for the client to fetch them in parallel. Unlike client-side query partitioning, server-side query 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 it knows that there is only one query. In the following, we discuss the potential solution for this proposal.

SQL Syntax & Workflow. A key requirement of supporting server-side query 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 query 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 endpoints, the client first 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. Here we discuss how to modify the existing database in order to support query partitioning and multi-endpoint accessing to a single query result. We consider 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 endpoints 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 connected endpoints.

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 endpoint 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 5.4, 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 5.4, 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.

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 EVALUATION

We conduct extensive experiments to evaluate ConnectorX.

5.1 Experimental Setup

Datasets & Workloads. (1) *TPC-H [18].* 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 partition 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 [13].* This dataset contains 12.8M traffic flows and is 6.3GB in CSV format. 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 partition 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 [43] 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 [17] and SQLAlchemy [23]) that follow the Python DB-API [29] to connect and access to databases.

Dask [49] 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 3.1), and then invoke Pandas.`read_sql` for each partitioned query in parallel.

Modin [45] 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.

Hardware & Platform. Our experiments are conducted on two AWS EC2 r5.4xlarge instances (16 CPU cores, 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 server and client resides on the same r5.4xlarge instance (Local) and on two locally hosted machines (16 Intel Xeon E7-4830 v4 CPUs and 128GB memory) with network bandwidth of 200Mbit/s. We use three open-source databases (PostgreSQL, MySQL, SQLite) and one commercial database (MySQL).

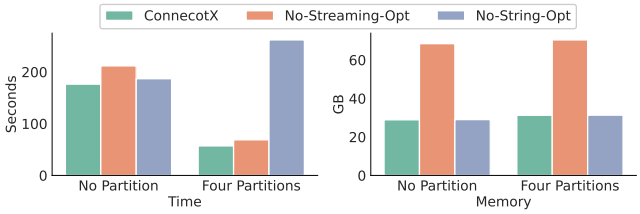

Figure 10: Ablation study.

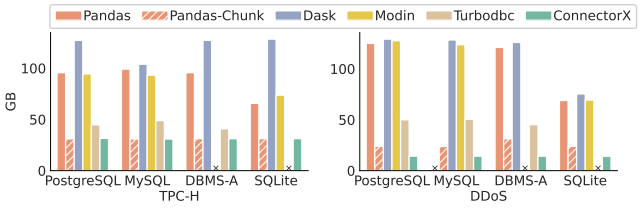

Figure 11: Memory comparison on four database systems. ("×" is placed if a method does not support the database or cannot handle the large query result.)

Implementation. We publish the scripts, datasets and workloads at https://github.com/sfu-db/connectorx-exp. We run each experiment five times and report the averaged result. We only provide the minimum information (partition number and partition column) to ConnectorX for query partitioning. Thus, ConnectorX needs to issue an extra query to get the min and max value of the partition column during the preparation phase.

5.2 Ablation Study

We conduct an ablation study to gain a deep understanding of ConnectorX's performance and verify the efficacy of the three design choices we made: 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 importance of the other two. We vary the implementation of ConnectorX and observe the performance change by loading the lineitem table from PostgreSQL to Pandas under no partition or four partitions settings. The results can be found in Figure 10. No-Streaming-Opt represents that the streaming workflow is not adopted; No-String-Opt represents that the string allocation optimization is not adopted.

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 number of partitions, it slows down the process by only 6% under the no partition setting. However, it becomes 4.6× slower when using 4 partitions. 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 the memory consumption. Without the streaming workflow optimization, No-Streaming-Opt needs 2.3× more memory due to the large intermediate results. On the other hand, 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 batch solution shown in Table 1. This validates the effectiveness of using Rust in terms of data representation.

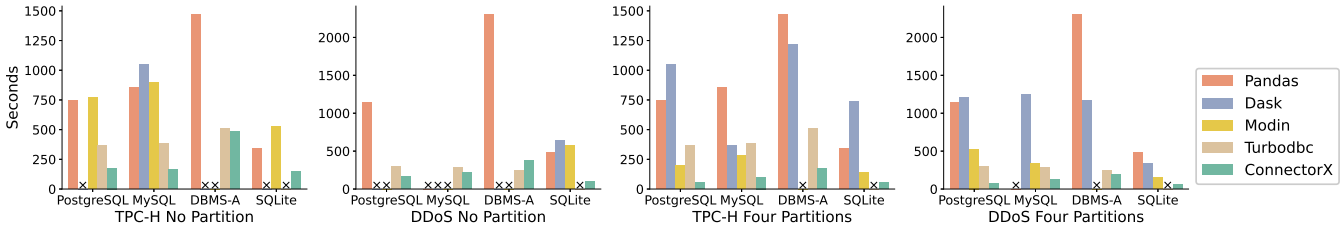

Figure 12: Speed comparison on four database systems. ("×" is placed if a method does not support the database or cannot handle the large query result.)

5.3 Performance Comparison

We compare ConnectorX with four state-of-the-art baselines when running `read_sql` to fetch the same query result into a Pandas dataframe. We first load the TPC-H lineitem and DDoS table, and then test how ConnectorX works under different network conditions and with more complex queries.

Memory Comparison. Figure 11 evaluates the peak memory usage of loading the entire TPC-H lineitem and DDoS tables. For the methods that support query partitioning (Modin, Dask, and ConnectorX), we show the result with four partitions. Pandas-Chunk enables chunk loading for Pandas (chunk size: 10K).

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. While Modin shows a similar result with Pandas, Dask suffers from the out-of-memory issue. 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 the majority DECIMAL data type, which is 13× larger in Python objects than in the final dataframe. Another interesting thing 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 of the 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 when network bandwidth is 10Gbit/s (except for SQLite, which only works on the client instance) in Figure 12. 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 four partitions to see how much speed up they could achieve when leveraging multiple cores on the client instance (the right two figures). Here we do not show the result of chunk loading since it 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 database'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.

Table 2: Speed compared to ConnectorX (four partitions) on PostgreSQL under different network bandwidth.

Bandwidth	Local	10 Gbit/s	200 Mbit/s
Pandas	14.26×	12.80×	3.05×
Dask	23.89×	17.94×	6.96×
Modin	3.88×	3.45×	1.58×
Turbodbc	6.64×	6.21×	1.90×
ConnectorX-NoPart	**3.16×**	**3.03×**	**1.08×**

(2) With Partitioning. When using four partitions, ConnectorX is the fastest one in all our 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 query partitioning to the same figure. With four partitions, 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 could barely finish with 128GB of memory, and since it needs to restart the workers when reaching to the memory limit, it is much slower than ConnectorX and Modin, and sometimes 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 four partitions as baseline and show its speedup w.r.t. each baseline. The speedups are shown in Table 2. ConnectorX-NoPart represents the result of ConnectorX when no partition is applied, and both Modin and Dask leverage four partitions. 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 may 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 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

Table 3: 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×

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 when using four partitions for ConnectorX.

For complex queries, getting metadata like the number of result rows becomes slower and may even need to execute the entire query. 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 3 shows the speedup of ConnectorX to Pandas. Green (Red) means that ConnectorX is faster (slower). We also list the cardinality of each query. It is clear that without query partitioning, ConnectorX is faster than Pandas by up to 3.8× or at least shows similar performance. Using four partitions could sometimes further speed up the process by up to 8.4×. Surprisingly, partitioning could 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 4 partitions by up to 3.3× especially when the result set is small. This finding further motivates the support of server-side query partitioning discussed in Section 4.2.

Please note that ConnectorX targets on the large query result fetching scenario. It speeds up the process by optimizing the client-side execution as well as saturating both network and machine resources through parallelism. When query execution on the database server is the bottleneck (e.g. a complex query with a small result set), however, ConnectorX brings less benefit and sometimes it can be even slower due to the overhead in fetching metadata.

5.4 Server-Side Query Partitioning

We build a prototype on PostgreSQL for each server-side query partitioning approach as discussed in Section 4.2. 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. We collect statistics during selecting the TPC-H lineitem table, including the number of times each tuple in the table has been fetched (# Scan), the number of disk blocks read (# Disk Block Miss, subtracting the number of cache hit), from PostgreSQL statistics collector, and measure the end-to-end running time. The results are recorded in Table 4. The second row shows the performance of ConnectorX without query partitioning, while the lower part represents the result of applying client-side and server-side Plan Partition using different number of partitions.

[4]https://github.com/sfu-db/connectorx-exp/tree/main/tpch-spj-workload/spj

[5]https://arrow.apache.org/docs/python/generated/pyarrow.Table.html?pyarrow.Table.to_pandas

Table 4: Plan Partition analysis. 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**

When no partition is applied, PostgreSQL scans the entire table only once and gets the final dataframe in 156.1 seconds. Although client-side partitioning improves the efficiency of read_sql, it also puts heavier burden on the database. The number of scans required on the underlying data 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 partition number is small. With 16 partitions, it becomes 15.5× larger due to the 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 partitioned queries are issued. Furthermore, with more partitions, the resource saving on the server side can further help to improve the 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 resource, while does not apply extra overhead on the database server.

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* [52] and *perf* [44], 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 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 database to dataframe.

Data Management for ML. ML tasks may need to access and manage raw input or intermediate data in auxiliary format. Data

lake solutions [14, 25, 51] are usually adopted in this situation. Lakehouse [54] 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 [11, 16, 31, 33, 41], which are developed to standardize and manage model features and workflows.

On the other hand, accessing these data from external tools is notoriously slow [35, 38, 54]. Previous work [47] 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 [48] 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 [38] adopts Flight [28] 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 databases and client drivers, and achieves the maximum speed up within the current implementations.

SQL-Python Integration. ML tools usually adopt dataframes [27, 43, 45, 46, 49] as the abstraction for data manipulation. Data scientists are in general more familiar with dataframe operations thus 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 works [24, 34, 37, 39, 50] try to run ML code inside database engines. Ibis [1] aims to convert dataframe operations to a sequence of SQL queries and run them on a connected database. Declarative dataframe APIs [21, 40] are proposed to combine relational and procedural processing, which also allows the freedom in terms of cross-optimization between ML and database operators and has been studied in [26, 30, 36]. ConnectorX complements these solutions and it allows data scientist to move data out of DBMS to conduct sophistical analysis and build ML models using the Python data science ecosystem.

7 CONCLUSION

In this paper, we proposed ConnectorX, an open-source library for loading query result from database to dataframe in a fast and memory-saving way. We conducted a thorough analysis on the popular `Pandas.read_sql`, and identified optimization opportunities on client-side execution. We developed ConnectorX targeting on optimizing client-side execution of `read_sql` without modifying the existing implementation of database servers as well as client drivers, and provides modular interfaces for contributors to extend easily. We also identified the drawbacks of current client-side query partitioning approach that ConnectorX and other libraries are using, and proposed that database systems should support server-side partitioning instead 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: 2021-12-29.

[2] 2016. pandas read_sql is unusually slow. https://stackoverflow.com/questions/40045093/pandas-read-sql-is-unusually-slow. Accessed: 2022-01-12.

[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: 2022-01-12.

[4] 2017. Program (Time) Bottleneck is Database Interaction. https://stackoverflow.com/questions/44154430/program-time-bottleneck-is-database-interaction. Accessed: 2022-01-12.

[5] 2017. Use Turbodbc/Arrow for read_sql_table. https://github.com/pandas-dev/pandas/issues/17790. Accessed: 2022-01-12.

[6] 2017-2021. Turbodbc - Turbocharged database access for data scientists. https://turbodbc.readthedocs.io/en/latest/. Accessed: 2021-12-22.

[7] 2021. ConnectorX for ETL workload. https://github.com/sfu-db/connector-x/discussions/133. Accessed: 2021-12-08.

[8] 2021. ConnectorX for ML feature fetching. https://github.com/sfu-db/connector-x/issues/140#issuecomment-948918848. Accessed: 2021-12-08.

[9] 2021. ConnectorX integrates with dataframe system. https://pola-rs.github.io/polars-book/user-guide/howcani/io/read_db.html. Accessed: 2021-12-08.

[10] 2021. ConnectorX plan for supporting data sources. https://github.com/sfu-db/connector-x/discussions/61. Accessed: 2021-12-08.

[11] 2021. Data Vesion Control (DVC). https://dvc.org/. Accessed: 2021-12-28.

[12] 2021. DataPrep: The easiest way to prepare data in Python. https://dataprep.ai/. Accessed: 2021-12-08.

[13] 2021. DDoS Dataset. https://www.kaggle.com/devendra416/ddos-datasets. Accessed: 2021-12-22.

[14] 2021. Google BigQuery. https://cloud.google.com/bigquery. Accessed: 2021-12-29.

[15] 2021. Polars: Fast multi-threaded DataFrame library in Rust and Python. https://github.com/pola-rs/polars. Accessed: 2021-12-08.

[16] 2021. Serve your features in production. https://feast.dev/. Accessed: 2021-12-28.

[17] 2021. sqlite3 — DB-API 2.0 interface for SQLite databases. https://docs.python.org/3/library/sqlite3.html. Accessed: 2021-12-22.

[18] 2021. TPC-H Homepage. http://www.tpc.org/tpch. Accessed: 2021-12-22.

[19] 2022. The Rust Programming Language - Macro. https://doc.rust-lang.org/book/ch19-06-macros.html. Accessed: 2022-01-14.

[20] 2022. The Rust Programming Language - Traits: Defining Shared Behavior. https://doc.rust-lang.org/book/ch10-02-traits.html. Accessed: 2022-01-15.

[21] 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

[22] 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: 2022-01-12.

[23] 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

[24] 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

[25] 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

[26] 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

[27] The Apache Software Foundation. 2016-2021. Apache Arrow. https://arrow.apache.org/. Accessed: 2021-11-18.

[28] The Apache Software Foundation. 2016-2021. Apache Arrow Flight. https://arrow.apache.org/docs/format/Flight.html. Accessed: 2021-11-18.

[29] The Python Software Foundation. 2001. PEP 249 – Python Database API Specification v2.0. https://www.python.org/dev/peps/pep-0249/. Accessed: 2021-10-18.

[30] 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

[31] 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: 2021-12-28.

[32] 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
[33] Jeremy Hermann and Mike Del Balso. 2017. Meet Michelangelo: Uber's Machine Learning Platform. https://eng.uber.com/michelangelo-machine-learning-platform/. Accessed: 2021-12-28.
[34] 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
[35] 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*.
[36] 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
[37] 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
[38] 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
[39] 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
[40] 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
[41] 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
[42] The pandas development team. 2008-2021. pandas.read_sql. https://pandas.pydata.org/docs/reference/api/pandas.read_sql.html. Accessed: 2022-01-22.
[43] The pandas development team. 2020. *pandas-dev/pandas: Pandas.* https://doi.org/10.5281/zenodo.3509134
[44] Linux perf framework. 2020. Linux perf framework. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed: 2021-12-06.
[45] 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
[46] R Core Team. 2021. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org/
[47] 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
[48] 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
[49] Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*. Citeseer.
[50] 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
[51] 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
[52] PostgreSQL top Project. 2020. PostgreSQL top Project. https://pg_top.gitlab.io/. Accessed: 2021-12-01.
[53] Itamar Turner-Trauring. 2021. Loading SQL data into Pandas without running out of memory. https://pythonspeed.com/articles/pandas-sql-chunking/. Accessed: 2022-01-21.
[54] 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