Zero-Cost, Arrow-Enabled Data Interface for Apache Spark

Sebastiaan Alvarez Rodriguez*, Jayjeet Chackrabroty‡,
Aaron Chu‡, Ivo Jimenez‡, Jeff LeFevre‡, Carlos Maltzahn‡ and Alexandru Uta*
*LIACS, Leiden University
Email: sebastiaanalva@gmail.com, a.uta@liacs.leidenuniv.nl
‡UC Santa Cruz
Email: {jchakra1,xweichu,ivotron,jlefevre,carlosm}@ucsc.edu

Abstract—Distributed data processing ecosystems are widespread and their components are highly specialized, such that efficient interoperability is urgent. Recently, Apache Arrow was chosen by the community to serve as a format mediator, providing efficient in-memory data representation. Arrow enables efficient data movement between data processing and storage engines, significantly improving interoperability and overall performance. In this work, we design a new zero-cost data interoperability layer between Apache Spark and Arrow-based data sources through the Arrow Dataset API. Our novel data interface helps separate the computation (Spark) and data (Arrow) layers. This enables practitioners to seamlessly use Spark to access data from all Arrow Dataset API-enabled data sources and frameworks. To benefit our community, we open-source our work and show that consuming data through Apache Arrow is zero-cost: our novel data interface is either on-par or more performant than native Spark.

I. INTRODUCTION

Distributed data processing frameworks, like Apache Spark [1], Hadoop [2], and Snowflake [3] have become pervasive, being used in most domains of science and technology. The distributed data processing ecosystems are extensive and touch many application domains such as stream and event processing [4], [5], [6], [7], distributed machine learning [8], or graph processing [9]. With data volumes increasing constantly, these applications are in urgent need of efficient interoperation through a common data layer format. In the absence of a common data interface, we identify two major problems: (1) data processing systems need to convert data, which is a very expensive operation; (2) data processing systems require new adapters or readers for each new data type to support and for each new system to integrate with.

A common example where these two issues occur is the de-facto standard data processing engine, Apache Spark. In Spark, the common data representation passed between operators is row-based [10]. Connecting Spark to other systems such as MongoDB [11], Azure SQL [12], Snowflake [3], or data sources such as Parquet [13] or ORC [14], entails building connectors and converting data. Although Spark was initially designed as a computation engine, this data adapter ecosystem was necessary to enable new types of workloads. However, we believe that using a universal interoperability layer instead enables better and more efficient data processing, and more data-format related optimizations.

The Arrow data format is available for many languages and is already adopted by many projects, including pySpark [1], Dask [15], Matlab [16], pandas [17], Tensorflow [18]. Moreover, it is already used to exchange data between computation devices, such as CPUs and GPUs [19]. However, the Apache Arrow Dataset API [20], not to be confused with the main Arrow [21] library, emerged as a platform-independent data consumption API, which enables data processing frameworks to exchange columnar data efficiently, and without unnecessary conversions. The Arrow Dataset API supports reading many kinds of datasources, both file formats and (remote) cloud storage. Exploring the benefits of the Arrow Dataset API on building storage connectors is currently an understudied topic.

In this paper, we therefore leverage the power of the Apache Arrow Dataset API and separate the computation offered by Spark from the data (ingestion) layers, which are more efficiently handled by Arrow. We design a novel connector between Spark and The Apache Arrow Dataset API, to which Spark can offload its I/O. Using the Arrow Dataset API, we enable Spark access to Arrow-enabled data formats and sources. The increasing adoption of Arrow will make many more data types and sources available in the future, without adding any additional integration effort for our connector and, by extension, for Spark.

In this work, we lay the foundation of integrating Spark with all Arrow-enabled datasources and show that the performance achieved by our connector is promising, exceeding in many situations the performance achieved by Spark-implemented data connectors. We experiment with several design points, such as batch sizes, compression, data types (e.g., Parquet or CSV), and the scaling behavior of our connector. Our analysis shows that our proposed solution scales well, both with increasing data sizes and Spark cluster sizes, and we provide advice for practitioners on how to tune Arrow batch sizes and which compression algorithms to choose. Finally, practitioners can integrate our connector in existing programs without modification, since it is implemented as a drop-in, zero-cost replacement for Spark reading mechanisms. The contribution of this work is the following:

1) The design and implementation of a novel, zero-cost data interoperability layer between Apache Spark and the Apache Arrow Dataset API. Our connector separates the
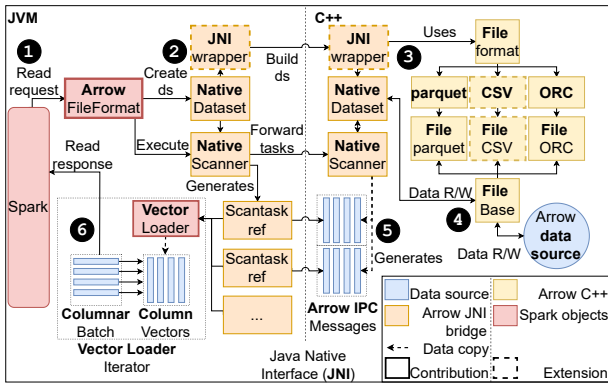
Fig. 1: Arrow-Spark design overview, integrating Arrow (right) and Spark (left) using the Java Native Interface (JNI).

computation (Spark) from the data (ingestion) layer (Arrow) and enables Spark interoperability with all Arrow-enabled formats and data sources. We open-source [22] our implementation for the benefit of our community.

2) The performance evaluation of the data interoperability layer. We show that Arrow-enabled Spark performs on-par or better than native-Spark and provide advice on how to tune Arrow parameters.

II. DESIGN AND IMPLEMENTATION

Our framework, called Arrow-Spark, provides an efficient interface between Apache Spark and all Arrow-enabled data sources and formats. Spark is in charge of execution, and Arrow provides the data, using its in-memory columnar formats. In Figure 1, we give a more detailed overview of how we access data through Arrow. Spark is a JVM-based distributed data processing system, whereas the Arrow Dataset API [20] is only implemented in C++ and Python, but not Java. To enable communication, we created a bridge implementation.

❶ **Reads.** Data transmission is initiated by a read request coming from Spark. Read requests arrive at the *Arrow File-Format* datasource.

❷ **JVM Dataset.** The *Arrow FileFormat* constructs a *Dataset* interface to read data through JNI, using the Arrow Dataset C++ API. The JVM *Dataset* interface forwards all calls through the *JNI wrapper* to C++. The Arrow *Dataset* API interface is constructed on the C++-side, and a reference UUID is passed to the JVM interface counterpart. Through this JVM interface, the *Arrow FileFormat* initiates data scanning (reading) using a *scanner*.

❸ **Arrow C++ Dataset API.** On the C++ side, a native *Dataset* instance is created. On creation, it picks a *FileFormat*, depending on the type of data to be read.

❹ **Data transmission.** The C++ Arrow Dataset API reads/writes the data in batches, using given FileFormat.

❺ **Arrow IPC.** Each data batch is placed in memory as an Arrow IPC message, which is a columnar data format. The address to access each message is forwarded to the JVM, and stored in a *Scantask* reference. Notice that here we make only one additional data copy.

```
def getAges: Unit {
    val conf: ArrowRDDReadConfig =
            ArrowRDDReadConfig.builder()
                .withPartitioner(...)
                .withNumPartitions(100)
                .withDataSourceURI(path)
                .build()
    val context = session.sparkContext
    val peopleRDD: ArrowRDD = ArrowSpark.load(context, conf)
    peopleRDD.filter(row => row.getShort(1) > 42)
    // ...
}
```

Fig. 2: Example usage of our connector for Spark RDDs.

❻ **Conversion.** Each Arrow IPC message is converted to an array of Spark-readable *column vectors*. Because Spark operators exchange row-wise data, we convert the *column vectors* to a row-wise representation by wrapping the vectors in a *ColumnarBatch*, which wraps columns and allows row-wise data access on them. This batch is returned to Spark and incurs data copying which cannot be avoided due to Spark operators only working on row-based data.

When reading data in Arrow, one uses a Dataset object. Datasets contain required information about datasources, such as the location of the data. On the JVM side (e.g., in Spark), a Dataset object holds a references to a C++ Arrow Dataset. To obtain data, we scan the dataset using a Scanner. We execute the scanner, which provides us with a list of scan tasks. Each task loads the next batch of results into memory, in a columnar representation, as an Arrow Inter-Process Communication (IPC) message. To acquire the data as arrays of Spark-readable column vectors, we use a *VectorSchemaRoot*. Once all necessary Arrow Vectors (columns) are extracted, we wrap the vectors in a *ColumnarBatch*. As we tried to display in Figure 1, a ColumnarBatch provides an interface to read a group of columns in a row-by-row fashion. This is required, because Spark can only process row-wise data. Now Spark is able to iterate over rows of data, and this concludes the data provisioning through Arrow.

All interaction with Arrow is performed lazily, meaning they will only be executed on demand from Spark. This ensures we do not have to load in all data at once. This is good for performance, because external datasources may be larger than the memory available on Spark clusters.

A. Usage, available APIs and extensions

When designing our connector, we specifically searched for options to easily control Arrow from Scala, whether users need RDDs or the more modern Dataframes or Spark Datasets. We show a simple example for loading RDDs, and another example for Dataframes. In Figure 2, we show how we can obtain a RDD. We implemented a custom configuration object due to the many options that can be adjusted. Users can configure simple options, like how many partitions are to be generated, but also extend the connector by providing a custom partitioner to be used when splitting the data, or even a custom object to load Arrow datasets. After an ArrowRDD is loaded, we can use it like any other RDD. Providing the file format

2401

```
def getAges: Unit {
    val df: DataFrame = session.read.arrow(path)
    df.createOrReplaceTempView("People")
    session.sql("SELECT name FROM People WHERE age > 42")
    // ...
}
```

Fig. 3: Example usage of our connector for Spark DataFrames, Spark Datasets

TABLE I: Description of the experiments in this work.

Experiment	Dataset Size (GB)	Batch Size (KB)	Query	Row Size (Bytes)
E1	1,144	32-32,768	Scan: select *	4×8
E2	71-4,500	256	Scan: select *	4×8
E3	71-1,144	256	Scan: select *	4×8
E4	71-1,144	256	Scan: select *	4×8
E5	715	25,600	Projection: select c1,c2,...	100×8

is optional. If not specified, the connector will automatically determine what kind of data is requested. The example shows a filter operation on an RDD of individuals older than 42 years.

To show how interacting with our modern connector works, we provide an example in Figure 3. This example should be familiar to Spark users, as the common API is not changed. In "session.read", we obtain a DataFrameReader. The part right after, ".arrow(path)" calls an implicitly defined function, which loads the "ArrowFileFormat" object. This is all that is needed from a user-perspective to use this connector. The supplied path can point to either parquet- or CSV files. The ArrowFileFormat will automatically determine what kind of data is requested.

B. Arrow-Spark JNI Bridge

Apache Spark (core) is implemented in Scala, and there does not exist an Arrow Dataset implementation written in any JVM language. The Arrow Dataset API [20] is not to be confused with main Arrow library, for which a Java stub implementation exists [23].

Practitioners using Spark with Arrow are currently bound to a very small set of features. To use the pyarrow-dataset (Python wrappers around the C++ Arrow dataset API) implementation with PySpark (Python wrapper over Spark), one needs to implement these explicitly through the PySpark program, unlike our approach which is transparent to the programmer. Then, the Python bridge between Spark (JVM) and Arrow (C++) adds a highly inefficient link in applications, and a large functionality limitation. PySpark requires to convert the pyarrow dataset tables to *pandas* data, a PySpark-readable format. This conversion cancels Spark's lazy reading, and requires materializing the entire dataset into memory. We experimented with a PySpark+pyarrow setup, and found it was consistently 30 to 50 times slower than our connector, with a growing performance difference when increasing dataset sizes. Additionally, due to eager materialization dataset size is limited by RAM capacity.

Our work in this paper has a much wider scope, aiming to make core Spark read data using core Arrow, providing universal data access to core Arrow from Core Spark and all its wrapper implementations at once.

Implementing our connector in core Spark (JVM) circumvents all aforementioned inefficiencies and shortcomings. We can therefore use our connector with all programming languages that Spark supports.

The core Arrow Dataset implementation is in native C++. To access it, we use a Java Native Interface (JNI) im-

plementation [24], based on the Intel Optimized Analytics Platform project [25]. Even though we could have chosen other languages for which there exists an Arrow Dataset implementation, we decided to use C++, because of its native-execution performance. Moreover, the Arrow Dataset API core is programmed in C++. Using any other language with Arrow bindings would add additional overhead. Even though the bridge between the JVM and native code brings a slight time penalty, we were able to minimize it by limiting data copies (see Figure 1 for the data copies that our interface implements). Additionally, C and C++ are the best-supported languages for interfacing with the JVM, through JNI.

III. ARROW-SPARK PERFORMANCE

We evaluate the performance of Arrow-Spark by comparing its performance with a standard Spark reading parquet and CSV data formats. We performed 5 separate experiments (E1-E5) to reveal various performance aspects of Arrow-Spark, as described in Table I. Each experiment draws appropriate conclusions and provides advice for practitioners.

A. Experiment Reproducibility

We used the specifications and parameters we give here as default values for every experiment, unless stated otherwise.
Experiment. We ran every experiment 31 times for every configuration. We discard the first execution for every experiment, because the JVM- and memory caches are still 'cold' during this run, producing an outlier. Between runs of the same experiment we did not close the Spark session as to keep JVM and caches warm. We adhered to the guidelines provided by Uta et al. [26], to ensure our performance results are reproducible and significant. Due to the stability of our cluster, the difference between the $1st$ and $99th$ percentiles for each experiment were below 10% (with one exception which we comment upon in Section III-B).
Hardware and Deployment. We ran all experiments on a cluster of 9 machines (8 Spark executor nodes and one driver), each equipped with 64 GB RAM, and a dual 8-core Intel E5-2630v3 CPU running at 2.4 GHz. Servers are interconnected with FDR InfiniBand connection (≈ 56 Gb/s). In the experiments, we always provide the number of nodes in terms of executor nodes. Spark is allowed to use up to 43 GB of the available 64 GB RAM in each node. Note that approximately 17 GB of RAM is already in use by input data as explained below. When reading parquet, we normally read uncompressed files unless stated otherwise.
Dataset. By default, we process 38.4×10^9 rows ($\approx 1,144$ GB) of synthetic data. The data we experiment with has rows of 4
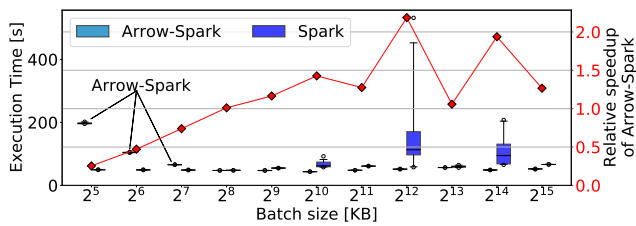
Fig. 4: *Batch size matters:* Execution time (boxplots, left y-axis) and relative speedup (curve, right y-axis) with varying batch sizes.

Fig. 5: *Arrow-Spark performance is good:* Execution time (bars) and relative speedup (curve) on increasing parquet data sets.

up to 100 columns which are 64-bit integers, split into files of 17 GB. During initial testing, we found that our experiments were influenced by local disk speed. With local disk reading, the bottleneck lies in disk speed, and we cannot uncover any framework inefficiencies. To mitigate this issue, we decided to deploy data as close as possible to the CPUs. We chose to deploy on RAMDisk, a memory-backed filesystem. Our machines have only 64 GB RAM, while we want to experiment with datasets of much larger sizes. To solve this new problem, we created X hardlinks for every file in a dataset of, e.g. 17 GB. Spark reads the hardlinks as if they were regular files, simulating a dataset size of $17 \cdot (X + 1)$ GB. This way, we were able to experiment with virtually infinitely large datasets, regardless of the RAM size constraints.

Performance Measurement. We are interested in I/O performance. To measure it we created queries (see Table I) which only read in all data, and count the number of rows as a way of triggering execution. This way, we can accurately measure the read performance. By default, we read this data into a Spark Dataframe (DF) to test with, reading 256 KB per batch. We replicated all experiments using Spark SQL, Datasets and directly RDDs, and the conclusions we drew were similar to what we present in this paper.

B. E1: Batch Size Tuning

To request Arrow to read data, we place (columnar) parquet data in memory buffers of limited sizes, i.e., *batch sizes*. We measured the execution time under varying amounts of batch sizes. The results are plotted in Figure 4. The performance of default Spark remains largely unchanged when changing the amount of rows a buffer may hold for the smaller batch sizes. Only when setting this number to a very large amount, Spark seems to suffer a more significant overhead. After investigating this further, we found that the overhead is because Spark uses up all available memory with larger batch sizes, causing many garbage collection calls and memory swapping. Arrow is much more memory efficient, due to its use of streaming principles to transmit data.

For Arrow-Spark, choosing a batch size is much more important for performance. Low batch sizes degrade performance significantly (see first two boxplots for 2^5 and 2^6 batch sizes). The root cause for this behaviour is the way Arrow-Spark works. For every batch, the Arrow Dataset API has to read data and transform it to IPC format. Further, the JVM-side reads
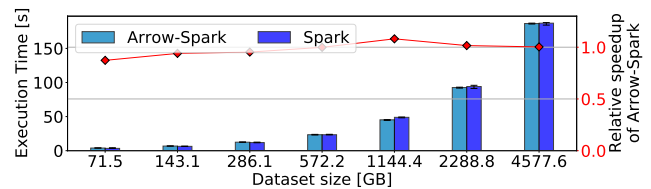
this data and converts it to a Spark-understandable format. With smaller batches, there are more conversions and memory copies on both sides. Batches of 8192 rows offer the best performance with both systems, due to hardware features. Each row in our sample data consists of four 64-bit integers. With 8192 rows, we get batches of exactly 256 KB. Our CPUs have a 256 KB L2 cache per core. (With different CPUs, the best batch size could be different.) By choosing a batch size of 8192 rows, we can exactly fit one batch inside the L2-cache. We verified the correctness of this hypothesis by experimenting with other data shapes.

Conclusion-1: Practitioners should always overestimate rather than underestimate batch sizes for Arrow-Spark. Underestimations cause strong performance degradation.

C. E2: Data Scalability

An important property of a distributed data processing system is its *data scalability* behavior, which evaluates system performance with increasing dataset sizes. We measured data scalability by reading parquet datasets with a wide range of different sizes. The results of this experiment are depicted in Figure 5. This figure plots the execution time with increasing dataset sizes (left vertical axis), as well as the relative speedup (or slowdown) between the two systems (right vertical axis). The execution time approximately doubles as the dataset size doubles. Both Spark and Arrow-Spark scale well.

Despite being slower than Spark on the smaller datasets we tested, Arrow-Spark gradually becomes relatively faster than Spark for larger datasets. We depict this effect in the lineplot of Figure 5, which shows the relative speedup of our framework, when compared to default Spark. This effect is because Arrow-Spark has several sources of constant overhead. Using the JNI bridge is the largest cause of overhead. The reading itself is slightly faster than with Spark, causing the difference between the measured systems to grow with the increase in dataset size. We found similar patterns when scaling the cluster size between 4 and 32 nodes.

Conclusion-2: Arrow-Spark scales well with dataset sizes. Its advantage over Spark is highest with 1-2 TB datasets. Practitioners can leverage Arrow-Spark at zero additional cost with larger dataset sizes.

D. E3: Row-wise Formats

Arrow-Spark supports multiple file formats through Arrow functionality. We performed an experiment comparing Arrow-Spark and default Spark on CSV data. The results are dis-
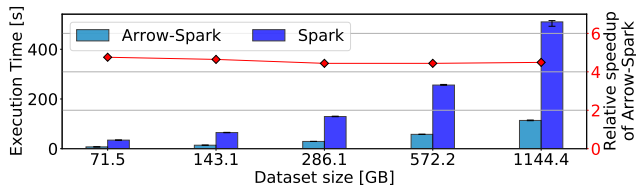
Fig. 6: *Arrow-Spark is faster than Spark with CSV files:* Execution time (bars) and relative speedup (curve) on increasing CSV data sets.
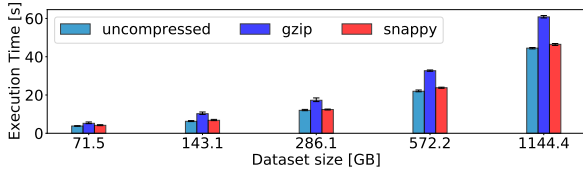

Fig. 7: *Arrow-Spark leverages compression:* Execution time for Arrow-Spark with different compression techniques for parquet data.

played in Figure 6. Our implementation is significantly faster than Spark, starting at a speedup factor of approximately 4.5. Larger datasets do not influence this speedup, which remains constant over all dataset sizes we tested.

This is due to Spark's inefficient CSV reader, the Java-based Univocity-CSV parser [27]. By contrast, our connector works with the highly efficient C++ Arrow Dataset API implementation. In cases where parsing is involved, using native code commonly is much more performant than other solutions. The cost of processing more data is higher for Spark, and much lower for Arrow-Spark due to the differences in parsers. This explains why using larger datasets results in a relatively bigger runtime increase for Spark.

Conclusion-3: Overall, Arrow-Spark brings significantly higher performance for ingesting text-based file formats than default Spark and the performance difference is constant with respect to dataset size. Practitioners should always choose Arrow-Spark for such file formats.

E. E4: Parquet Compression

Arrow-Spark can leverage parquet compression. We compare the performance of Arrow-Spark under several compression options. The results can be found in Figure 7. Note that using compressed parquet files in practical situations produces vastly different results from the results we obtained in our experimental setup using RAMDisk. Usually, the bottleneck for reading data is I/O. Using compressed data trades I/O for increased CPU load. In our setup, we deploy data on RAMDisk, and we have no I/O bottleneck. Reading compressed data only increases CPU load in this case, decreasing performance. We compare reading *uncompressed* parquet with *snappy* and *gzip*. When reading from memory, without much I/O overhead, reading compressed data is slower due to the added CPU overhead of decompressing it.

Conclusion-4: Arrow-Spark is able to leverage various compression algorithms for parquet files. When leveraging slower
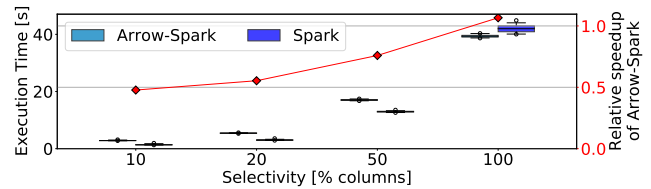

Fig. 8: *Arrow-Spark pushes down projections:* Execution time (boxplots) and relative speedup (curve) when reading 9.6 billion rows in batches of 32,768 rows, varied projection selectivity.

I/O media, these can help practitioners by trading I/O volume for increased CPU-load.

F. E5: Column Projection

One benefit of many columnar dataformats is the ability to project columns, i.e. to select a subset of the total number of columns for reading at little to no cost. Arrow-Spark pushes down projection operations from Spark to Arrow. The benefit of doing this is that Arrow becomes in charge of providing subsets of data. Practitioners need not change code, but use the Arrow-Spark interface we designed, because projections are directly controlled by the Spark query optimizer. The results can be found in Figure 8. When selecting very few columns, Spark is relatively quicker because it pushes down projections to its parquet reader, while Arrow-Spark pushes down to Arrow over the JNI. When selecting more columns, Arrow-Spark becomes relatively faster due to its advantage over larger data sizes. Overall, this behavior is consistent with findings from E2. Although Spark is seemingly faster at higher selectivities, this is relative: on real-world datasets, even 1% selectivity could leverage datasets of over 1TB, at which point Arrow-Spark's performance is superior.

Conclusion-5: Arrow-Spark is able to leverage projections for parquet files, effectively pushing down projection queries to the data layer, minimizing data movement.

IV. RELATED WORK

Several connector extensions allow users to read new file formats or datasource backends, such as HDF5 or netCDF [28], as well as extensions on Hadoop to allow efficiently reading of highly structured array-based data. Albis [29], proposes a new columnar-hybrid format to be used with modern hardware. Frameworks like MongoSpark [30] and Databricks Spark-RedShift [31] connect a specific backend to Spark. There also are extensions to improve existing reading implementations: Intel Optimized Analytics Package (OAP) [25], and the JVM Arrow dataset API [24]. Our connector has a different approach to memory allocation, backward-compatibility, and we support more file types at the moment.

Databricks [32] also provides several connectors separating execution from data layers, such as JDBC and ODBC connectors [33]. The difference between such projects and our work is that we provide a separation layer between execution and data

ingestion with high interoperability for any (Arrow-enabled) datasource. There is no official support for using Core (Scala) Spark together with Core Arrow (C++). Our system provides such support.

V. Conclusion

We investigated decoupling the computation and the data (ingestion) layers. This is needed for enabling interoperability and reducing the amount of data conversion. We built a prototype that leverages Arrow-enabled data sources to Apache Spark, effectively decoupling Spark's computation from Arrow's data ingestion. Through our experimentation, we concluded that using connectors like ours is zero-cost: Arrow-Spark not only retains overall performance, but in some cases even significantly improves it. Next to improved performance, we gain the ability to access any datasource that implements support for Arrow, allowing us to connect to many different data processing frameworks, data storage systems and file formats.

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica *et al.*, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[2] Apache Software Foundation, "Hadoop." [Online]. Available: https://hadoop.apache.org

[3] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang *et al.*, "The snowflake elastic data warehouse," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 215–226.

[4] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[5] M. H. Iqbal and T. R. Soomro, "Big data analysis: Apache storm perspective," *International journal of computer trends and technology*, vol. 19, no. 1, pp. 9–14, 2015.

[6] R. Shree, T. Choudhury, S. C. Gupta, and P. Kumar, "Kafka: The modern platform for data management and analysis in big data domain," in *2017 2nd International Conference on Telecommunication and Networks (TEL-NET)*, 2017, pp. 1–5.

[7] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé *et al.*, "Ibm streams processing language: Analyzing big data in motion," *IBM Journal of Research and Development*, vol. 57, no. 3/4, pp. 7–1, 2013.

[8] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging ai applications," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 561–577.

[9] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *First international workshop on graph data management experiences and systems*, 2013, pp. 1–6.

[10] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, 2015, pp. 1383–1394.

[11] K. Chodorow, *MongoDB: the definitive guide: powerful and scalable data storage.* " O'Reilly Media, Inc.", 2013.

[12] D. Chappell *et al.*, "Introducing windows azure," *Microsoft, Inc, Tech. Rep*, 2009.

[13] D. Vohra, "Apache parquet," in *Practical Hadoop Ecosystem*. Springer, 2016, pp. 325–335.

[14] Apache, "Apache orc - high-performance columnar storage for hadoop," https://orc.apache.org/, Apache Software Foundation, accessed: 2020-11-06.

[15] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th python in science conference*, vol. 126. Citeseer, 2015.

[16] MATLAB, *version 7.10.0 (R2010a)*. Natick, Massachusetts: The MathWorks Inc., 2010.

[17] G. Van Rossum *et al.*, "Python," 1991.

[18] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[19] cuDF community, "cudf - gpu dataframes," https://github.com/rapidsai/cudf, RAPIDS, accessed: 2020-11-16.

[20] A. D. Team, https://arrow.apache.org/docs/python/dataset.html#reading-from-cloud-storage, Apache Software Foundation, accessed: 2020-11-08.

[21] ——, "Apache arrow," https://arrow.apache.org, 10 2018.

[22] Anonymous, "Arrow-spark," https://github.com/Sebastiaan-Alvarez-Rodriguez/arrow-spark-publication, 2021.

[23] A. D. Team, "Apache arrow java implementation," https://arrow.apache.org/docs/java/, 10 2018.

[24] H. Zhang, "Arrow-7808: [java][dataset] implement dataset java api by jni to c++," https://github.com/zhztheplayer/arrow-1/tree/ARROW-7808, Github, Apache Arrow community, accessed: 2020-09-14.

[25] I. Corporation, "Optimized analytics package for spark platform (oap)," https://github.com/Intel-bigdata/OAP , Github, Intel Corporation, accessed: 2021-01-15.

[26] A. Uta, A. Custura, D. Duplyakin, I. Jimenez, J. Rellermeyer, C. Maltzahn, R. Ricci, and A. Iosup, "Is big data performance reproducible in modern cloud networks?" in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 513–527.

[27] T. univocity team, "univocity-parsers," https://github.com/uniVocity/univocity-parsers , Github, uniVocity, accessed: 2021-02-01.

[28] J. Liu, E. Racah, Q. Koziol, R. S. Canon, and A. Gittens, "H5spark: bridging the i/o gap between spark and scientific data formats on hpc systems," *Cray user group*, 2016.

[29] A. Trivedi, P. Stuedi, J. Pfefferle, A. Schuepbach, and B. Metzler, "Albis: High-performance file format for big data systems," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 615–630.

[30] R. Lawley, "Apache spark connector for mongodb," https://www.slideshare.net/mongodb/how-to-connect-spark-to-your-own-datasource, https://databricks.com/blog/2015/03/20/using-mongodb-with-spark.html, MongoDB, accessed: 2020-11-06.

[31] Databricks, "Databricks," https://github.com/databricks/spark-redshift, Databricks, Github, accessed: 2021-02-01.

[32] ——, "Databricks," https://databricks.com/, Databricks, accessed: 2021-02-01.

[33] ——, "Databricks," https://docs.databricks.com/data/data-sources/sql-databases.html, Databricks, accessed: 2021-02-01.