Albis: High-Performance File Format for Big Data Systems

Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schuepbach, and Bernard Metzler, *IBM Research, Zurich*

https://www.usenix.org/conference/atc18/presentation/trivedi

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

Albis: High-Performance File Format for Big Data Systems

Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schuepbach, and Bernard Metzler
IBM Research, Zurich

Abstract

Over the last decade, a variety of external file formats such as Parquet, ORC, Arrow, etc., have been developed to store large volumes of relational data in the cloud. As high-performance networking and storage devices are used pervasively to process this data in frameworks like Spark and Hadoop, we observe that none of the popular file formats are capable of delivering data access rates close to the hardware. Our analysis suggests that multiple antiquated notions about the nature of I/O in a distributed setting, and the preference for the "storage efficiency" over performance is the key reason for this gap.

In this paper we present Albis, a high-performance file format for storing relational data on modern hardware. Albis is built upon two key principles: (i) reduce the CPU cost by keeping the data/metadata storage format simple; (ii) use a binary API for an efficient object management to avoid unnecessary object materialization. In our evaluation, we demonstrate that in micro-benchmarks Albis delivers $1.9-21.4\times$ faster bandwidths than other formats. At the workload-level, Albis in Spark/SQL reduces the runtimes of TPC-DS queries up to a margin of $3\times$.

1 Introduction

Relational data management and analysis is one of the most popular data processing paradigms. Over the last decade, many distributed relational data *processing* systems (RDPS) have been proposed [15, 53, 29, 38, 35, 24]. These systems routinely process vast quantities of (semi-)structured relational data to generate valuable insights [33]. As the volume and velocity of the data increase, these systems are under constant pressure to deliver ever higher performance. One key factor that determines the performance is the data access rate. However, unlike the classic relational database management systems (RDBMS) which are jointly designed for optimal data storage and processing, modern cloud-based

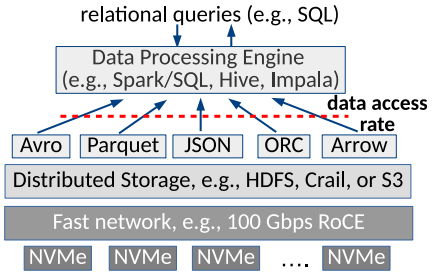


Figure 1: Relational data processing stack in the cloud.

RDPS systems typically do not manage their storage. They leverage a variety of external file formats to store and access data. Figure 1 shows a typical RDPS stack in the cloud. This modularity enables RDPS systems to access data from a variety of sources in a diverse set of deployments. Examples of these formats are Parquet [10], ORC [9], Avro [6], Arrow [5], etc. These formats are now even supported by the RDBMS solutions which add Hadoop support [49, 41, 31]. Inevitably, the performance of a file format plays an important role.

Historically, file formats have put the top priority as the "storage efficiency", and aim to reduce the amount of I/O as much as possible because I/O operations are considered slow. However, with the recent performance advancements in storage and network devices, the fundamental notion of *"a fast CPU and slow I/O devices"* is now antiquated [44, 40, 54]. Consequently, many assumptions about the nature of storage in a distributed setting are in need of revision (see Table 1). Yet, file formats continue to build upon these antiquated assumptions without a systematic consideration for the performance. As a result, only a fraction of raw hardware performance is reflected in the performance of a file format.

In this paper, we re-visit the basic question of storage and file formats for modern, cloud-scale relational data processing systems. We first start by quantifying the impact of modern networking and storage hardware

Assumption	Implications	Still valid in a modern setup?
1. I/O operations are orders of magnitude slower than the CPU	Use compression and encoding to reduce the amount of I/O required	No, with high-performance devices, the CPU is the new performance bottleneck [54, 20]
2. Random, small I/O accesses are slow	Use large block sizes to make large sequential scans [30, 29]	No, modern NVMe devices have high-performance for random, small accesses
3. Avoid remote data access	Preserve locality by packing data in discrete storage blocks [25]	No, fast I/O devices with network protocols (e.g., NVMeF) make remote storage as fast as local [26, 34]
4. Metadata lookups are slow	Decrease the number of files and blocks needed to store data [30]	No, high-performance distributed storage systems can do millions of lookups per second [50]
5. The final row representation is not known	Provide object/data-oriented API for row and column data types	No, the final row/column representation is often known (e.g., Spark `UnsafeRow`) and a binary API can be used

Table 1: Assumptions (1–2 are local, and 3–5 are distributed) and their implications on storing relational data.

on the performance of file formats. Our experiments lead to three key findings. First, no popular file format we test can deliver data access bandwidths close to what is possible on modern hardware. On our 100 Gbps setup, the best performer delivers about $⅓^{rd}$ of the bandwidth of the networked storage. Secondly, the CPU is the new bottleneck in the data access pipeline. Even in the presence of high-performance I/O hardware, file format developers continue to trade the CPU performance for "efficient" I/O patterns. Although this decision made sense for disks and 1 Gbps networks, today, this leads to the CPU being kept busy with (de)compressing, en/decoding, copying data, managing objects, etc., while trying to keep up with incoming data rates. Lastly, at the distributed systems level, strict adherence to locality, preference for large sequential scans, penchant to decrease the number of files/blocks, and poor object management in a managed run-time result in a complex implementation with a very high CPU cost and a poor "COST" score [37].

Based upon these findings, in this paper, we propose Albis, a simple, high-performance file format for RDPS systems. Albis is developed upon two key principles: (i) reduce the CPU cost by keeping the data/metadata storage format simple; (ii) use a binary API for an efficient object management to avoid unnecessary object materialization. These principles then also simplify the data/file management in a distributed environment where Albis stores schema, metadata, and data in separate files for an easy evolution, and does not enforce a store like HDFS to use local blocks. In essence, Albis's top priority is to deliver performance of the storage and network hardware without too much intrusion from the software layer. Our specific contributions in this paper are:

- Quantification of the performance of popular file formats on modern hardware. To the best of our knowledge, this is the first systematic performance evaluation of file formats on 100 Gbps network and NVMe devices. Often, such evaluations are muddied in the description of the accompanying relational processing system, which makes it hard to understand where the performance bottlenecks in the system are.
- Revision of the long held CPU-I/O performance assumptions in a distributed setting. Based upon these revisions, we propose Albis, a high-performance file format for relational data storage systems.
- Evaluation of Albis on modern hardware where we demonstrate that it can deliver performance within 15% (85.5 Gbps) of the hardware. Beyond micro-benchmarks, we also integrate Albis in Spark/SQL and demonstrate its effectiveness with TPC-DS workload acceleration where queries observe gains up to 3×.

2 File Formats in the Age of Fast I/O

The choice of a file format dictates how multi-dimensional relational tables are stored in flat, one-dimensional files. The initial influential work of column-oriented databases have demonstrated the effectivness of column storage for disks [51, 12]. This has led to the development of a series of columnar file formats. The most prominent of them are Apache Parquet [10], Optimized Row Columnar (ORC) [9], and Arrow [5]. All of these columnar formats differ in their column/storage encoding, storage efficiency, and granularity of indexing. Apart from providing a row-at-a-time API, all of these formats also have a high-performance vector API for column data. In this paper, we use the vector API for evaluation. In contrast to column-storage, we also consider two popular row-oriented formats. JSON [11] is a simple data representation that encodes schema and data together. JSON is widely supported due to its simplicity. The other format is Avro [6] that decouples schema and data presentation where both can evolve independently.

2.1 The Mismatch Assumptions

In this section, we re-visit the basic assumptions made about the nature of I/O and what impact they have on the

	1 GbE	100 GbE	Disk	NVMe
Bandwidth	117 MB/s	12.5 GB/s	140 MB/s	3.1 GB/s
`cycles/unit`	38,400	360	10,957	495

Table 2: Bandwidths and `cycles/unit` margins for networking and storage devices. A unit for network is a 1,500 bytes packet, whereas for storage it is a 512 bytes sector. `Cycles/unit` roughly denote the number of free CPU cycles for every data unit for a 3 GHz core. As a reference, a DRAM-access would be around 100 cycles.

file format design in the presence of modern hardware. This discussion is summarized in Table 1.

1. I/O operations are orders of magnitude slower than the CPU: During the last decade, we have witnessed the rise of high-performance storage and networking devices like 40 and 100 Gbps Ethernet, and NVMe storage. Once the staple of high-performance computing clusters, these devices and associated APIs can now be found in commodity cloud offerings from multiple vendors [3, 2, 1]. At the same time, the CPU performance improvements have stalled due to various thermal and manufacturing limits. Hence, the CPU's margin for processing incoming bytes has shrunk considerably [45, 21, 16, 54]. In Table 2 we summarize the bandwidths for state-of-the-art I/O devices from a decade ago and now. We also show the `cycles/unit` metric as an estimate of the CPU budget for every incoming unit of data. For the network, the unit is a 1,500 bytes packet, and for storage it is a 512 bytes sector. For a 3 GHz core (ignoring the micro-architectural artifacts), the number of cycles per second is around 3×10^9. The table shows that in comparison to a decade ago, CPU cycle margins have shrunk by two orders of magnitude.

2. Random, small I/O accesses are slow: Disk seeks are slow and take $\sim$10s of milliseconds, a cost that cannot be amortized easily for small accesses. Hence, disk-based file formats advocate using large I/O segments, typically a multiple of the underlying storage block, e.g., 128 or 256 MB [30]. However, NVMe devices can deliver high bandwidth for random, small I/O patterns. In our investigation (discussed in the next section), we find that the continuing use of large I/O buffers is detrimental to the cache behavior and performance. For example, on a 16 core machine with a 128 MB buffer for each task, the memory footprint of a workload would be 2 GB, a much larger quantity than the modern cache sizes.

3. Avoid remote data access: Modern NVMe devices can do 2-3 GB/s reads and 1-2 GB/s writes. At the same time, the availability of high-performance networks (40 and 100 Gbps) and efficient network protocols like NMVe-over-Fabrics, means that the performance gap between a local flash and remote flash is negligible [26, 34]. Hence, various proposed modifications to block placement strategies in Hadoop [25], and design decisions to pack schema, data, and metadata in the same block to avoid remote storage, can be relaxed.

4. Metadata lookups are slow: In any distributed storage, the number of metadata lookups is directly proportional to the number of blocks in a file. A lookup is an RPC that took 100-1,000 μs over 1 Gbps networks. This high cost has led to the decision to reduce the number of files (or blocks) by using complex block management strategies, type-specific encoding, packing data and schema in packed blocks, which in essence trades CPU for the I/O. However, modern storage solutions like Apache Crail [8] and RAMCloud [42] can do millions of metadata lookups/sec [50].

5. The final data representation is not known: File formats often assume that the final data representation in an RDPS engine is not known, and hence, a format must materialize the raw objects when reading and writing data. This decision leads to unnecessary serialization and object allocation, which hampers the performance in a managed run-time environment like Java.

2.2 Putting it Together: Performance

In this section, we quantify the cumulative effect of the aforementioned assumptions on the read performance of file formats on modern hardware. For this experiment, we read and materialize values from the `store_sales` table (the largest table) from the TPC-DS (scale=100) dataset. The table contains 23 columns, which consist of 10 integers, 1 long, and 12 decimal values. The input table is stored in the HDFS file system (`v2.7`) in Parquet (`v1.8`), ORC (`v1.4`), Arrow (`v0.8`), Avro (`v1.7`), and JSON formats. The goal of the experiment is to measure how fast we can materialize the values from a remote storage. The experiment is run between 2 machines (with dual Xeon E5-2690, 16 cores) connected via a 100 Gbps network. One machine is used to run the HDFS namenode and a datanode. This machine also contains 4 enterprise-grade NVMe cards, with a cumulative bandwidth of 12.4 GB/sec. The other machine runs the benchmarking code[1] on all 16 cores in parallel.

Figure 2 shows our findings. Here, the y-axis shows the effective *goodput* calculated by dividing the total incoming data size by the runtime. Notice that the incoming *data size* is different from the *file size*, which depends upon the file format used. We cannot use the file size for the bandwidth calculation because formats such as JSON use text encoding with interleaved schemas, thus making their file sizes up to 10$\times$ larger than the actual data size. In order to measure the actual data content, we count how

[1]The benchmarking code is open-sourced at `https://github.com/animeshtrivedi/fileformat-benchmarks`.

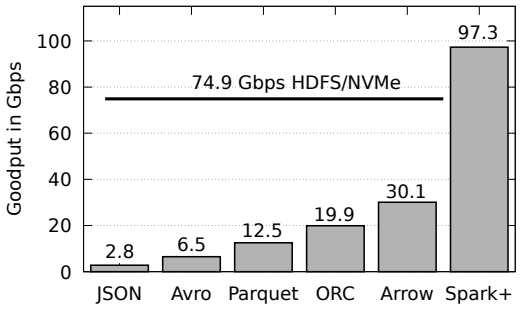


Figure 2: Reading bandwidths for various file formats.

		512M	256M	128M	64M	32M
Goodput (in Gbps)	Parq.	7.3	9.5	12.5	12.8	12.1
	ORC	13.6	17.5	19.9	20.2	20.1
Instructions/ row	Parq.	6.6K	6.7K	6.6K	6.6K	6.6K
	ORC	5.0K	4.9K	4.9K	4.8K	4.8K
Cache misses/ row	Parq.	11.0	10.6	9.2	7.1	6.5
	ORC	7.8	5.5	4.6	4.4	4.1

Table 3: Goodputs, instructions/row, and cache misses/row for Parquet (Parq.) and ORC with varying block sizes on a 16-core Xeon machine.

many integers, longs, and doubles the table contains, excluding the null values. We use this as the reference data point for the goodput measurements. We also show the data ingestion bandwidth of Spark/SQL [15] (as `spark+`) when the data is generated on the fly. The solid line represents the HDFS read bandwidth (74.9 Gbps) from the remote NVMe devices[2].

There are three key results from our experiment. First, as shown in Figure 2, none of the file formats that we test delivers bandwidth close to the storage performance. There is almost an order of magnitude performance gap between the HDFS (74.9 Gbps) and JSON (2.8 Gbps) and Avro (6.5 Gbps) performances. Columnar formats with their optimized vector API perform better. The best performer is Arrow, which delivers 40.2% of the HDFS bandwidth. Interestingly, Arrow is not for disks, but for in-memory columnar data presentation. Its performance only supports our case that with modern storage and networking hardware, file formats need to take a more "In-Memory"-ish approach to storage. In the same figure, we also show that in isolation from the storage/file formats, Spark/SQL can materialize `store_sales` rows from raw integers, longs, and doubles at the rate of 97.3 Gbps. Hence, we conclude that file formats are a major performance bottleneck for accessing data at high rates.

Secondly, the performance of these file formats are CPU limited, an observation also made by others [20]. When reading the data, all 16 cores are 100% occupied executing the thick software stack that consists of kernel code, HDFS-client code, data copies, encoding, (de)serialization, and object management routines. In Table 3 we present further breakdown of the performance (1^{st} row) with required instructions per row (2^{nd} row) and cache misses per row (3^{rd} row) for Parquet and ORC file formats when varying their block sizes. As shown, the use of large blocks (256 MB and 512 MB) always leads to poor performance. The key reason for the performance loss is the increased number of cache misses that leads to stalled CPU cycles. As we decrease the block size from 512 to 32 MB, the performance increases up to 128 MB, though the number of cache misses continues to decrease. At smaller block sizes (128 – 32 MB), the performance does not further improve because it is bottlenecked by the large number of instructions/row (remains almost constant as shown in the 2^{nd} row) that a CPU needs to execute. In further experiments, we use a 128 MB block size as recommended in the literature.

Thirdly, these inefficiencies are scattered throughout the software stack of file formats, and require a fresh and holistic approach in order to be fixed.

Summary: We have demonstrated that despite orders of magnitude performance improvements in networked-storage performance, modern file formats fail to deliver this performance to data processing systems. The key reason for this inefficiency is the belief in the legacy assumptions where CPU cycles are still traded off for I/O performance, which is not necessary anymore. Having shown the motivation for our choices, in the next section we present Albis, a high-performance file format.

3 Design of Albis File Format

Albis is a file format to store and process relational tabular data for read-heavy analytic workloads in a distributed setting. It supports all the primitive fixed (`int`, `timestamp`, `decimal`, `double`, etc.) and variable (`varchar` or `byte[]`) data types with simple and nested schemas. A nested schema is flattened over the column names and data is stored in the schema leaves. Albis's design is based upon the following choices:

- **No compression or encoding:** Albis decreases the CPU pressure by storing data in simple binary blobs without any encoding or compression. This decision trades storage space for better performance.
- **Remove unnecessary object materialization by providing a binary API:** The data remains in the binary format unless explicitly called for materialization. A

[2]Even though this is not 100 Gbps, it is the same bandwidth that HDFS can serve locally from NVMe devices. Hence, the assumption about the equality of local and remote performance holds.

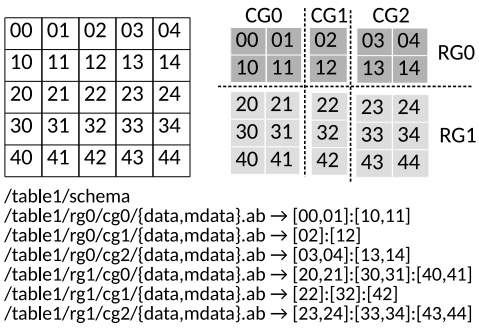


Figure 3: Table partitioning logic of Albis and corresponding file and directory layout on the file system.

reader can access the binary data blob for data transformation from Albis to another format (such as Spark's `UnsafeRow` representation) without materializing the data. This design choice helps to reduce the number of objects. A binary API is possible because the row data is not group compressed or encoded which requires the complete group to be decoded to materialize values.

- **Keep the metadata/data management simple:** Albis stores schema, data, and metadata in separate files with certain conventions on file and directory names. This setup helps to avoid complex inter-weaving of data, schema, and metadata found in other file formats. Due to the simple data/metadata management logic, Albis's I/O path is light-weight and fast.

In order to distribute storage and processing, Albis partitions a table horizontally and vertically as shown in Figure 3. The vertical partitioning (configurable) splits columns into *column-groups (CGs)*. At the one extreme, if each column is stored in a separate column group, Albis mimics a column store. On the other hand, if all columns are stored together in a single column group, it resembles a row store. In essence, the format is inspired by the DSM [23] model without mirroring columns in column groups. Horizontal partitioning is the same as sharding the table along the rows and storing them into multiple *row-groups (RGs)*. A typical row group is configurable either based on the number of rows or the size (typically a few GBs). The number and ordering of the row and column groups are inferred from their names as shown in Figure 3. Albis does not maintain any explicit indexes. Row data in various column groups are matched together implicitly by their position in the data file. The data, metadata, and file management and naming convention of Albis is similar to BigTable [22]. In the following sections, we discuss the storage format, read/write paths, support for data/schema evolution, and concerns with distributed data processing systems in detail. Table 4 shows the abridged Albis API.

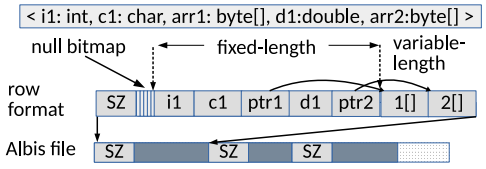


Figure 4: Albis row storage format.

3.1 Row Storage Format

After splitting the table along multiple CGs, each CG can be thought of as a table with its own schema and data. Row data from a column group is stored continuously, one after another, in a file. The Albis row format consists of four sections: a size counter, a null bitmap, a fixed-length and a variable-length section as shown in Figure 4. For a given schema, the number of fields determines the bitmap size in bytes. For example, a 23 columns schema (like TPC-DS `store_sales`) takes 3 bytes for the null bitmap. The fixed-length area is where data for fixed-length columns are stored in situ. A variable-length column data is stored in the variable-length area, and its offset and size is encoded as an 8-byte long and stored in the fixed area. With this setting, for a given schema, Albis calculates the fixed-length section size (that stays fixed, hence the name) by summing up the size of the fixed-type fields and 8×number of variable fields. For example, a schema of `<int, char, byte[], double, byte[]>` (as shown in the figure) takes one byte for bitmap, and 29 (= 4 + 1 + 8 + 8 + 8) bytes for the fixed segment. The row encoding is then prefixed by the total size of the row, including its variable segment. For a fixed-length schema (contains only fixed-length fields), Albis optimizes the row format by eschewing the size prefix as all rows are of the fixed, same size.

3.2 Writing Row Data

A writer application defines a schema and the column grouping configuration by allocating `AlbisSchema` and `AlbisColumn` objects. In the default case, all columns are stored together in a row-major format. The writer application then allocates an `AlbisWriter` object from the schema object. The writer object is responsible for buffering and formatting row data according to the storage format as described previously. Internally, the writer object manages parallel write streams to multiple CG locations, while counting the size. Once a RG size is reached, the current writers are closed, and a new set of writers in a new RG directory are allocated. Data is written and read in the multiple of *segments*. A segment is a unit of I/O buffering and metadata generation (default: 1 MB). The segment metadata includes the minimum and maximum values (if applicable), distribution

Class	Functions	Action
AlbisFileFormat	reGroup(Schema, Path, Schema, Path) reBalance(Path)	re-groups the schema in the given path location re-balances the data in the given path location
AlbisColumn	make(String, Type, ...)	makes a column with a name and type
AlbisColGroup	make(AlbisColumn[])	makes a column group from a given list of columns
AlbisSchema	getReader(Path, Filter[], AlbisColumn[]) getWriter(Path) makeSchema(ColumnGroup[])	gets a reader for a given location, projection, and filter gets a writer to a given location builds a schema with a given list of CGs
AlbisWriter	setValue(Int, Value) nextRow()	sets a value (can be null) for a given column index marks the current row done, and moves the pointer
AlbisReader	hasMore() and next() getValue(Int) getBinaryRow()	implements the Scala Iterator abstraction for rows gets the value (can be null) for a given column index returns a byte[] with the encoded row

Table 4: Abridged Albis API. Apart from row-by-row, Albis also supports a vector reading interface.

of data (e.g., sorted or not), number of rows, padding information, and offset in the data file, etc. The segment metadata is used for implementing filters.

3.3 Reading Row Data

A reader application first reads the schema from the top-level directory and scans the directory paths to identify row and column groups. The reader then allocates an AlbisReader object from the schema, which internally reads in parallel from all column groups to construct the complete row data. AlbisReader implements the Iterator abstraction where the reader application can check if there are more rows in the reader and extract values. The reader object reads and processes a segment's worth of data from all column groups in parallel, and keeps the row index between them in sync. An AlbisReader object also supports a binary API where row-encoded data can be returned as a byte[] to the application.

Projection: AlbisReader takes a list of AlbisColumns that defines a projection. Internally, projection is implemented simply as re-ordering of the column indexes where certain column indexes are skipped. Naturally, the performance of the projection depends upon the column grouping. In the row-major configuration, Albis cannot avoid reading unwanted data. However, if the projected columns are grouped together, Albis only reads data files from the selected column group, thus skipping unwanted data. As we will show in Section 4.1, the implementation of projection is highly competitive with other formats.

Filtering: Albis implements two levels of filtering. The first-level of filtering happens at the segment granularity. If a filter condition is not met by the metadata, the segment reading is skipped immediately. For example, if a segment metadata contains the max value of 100 for an integer column, and a filter asks for values greater than 500, the segment is skipped. However, if the condition is not specific enough, then the rows are checked one-by-one, and only valid rows satisfying all filter conditions are returned. Currently, Albis supports null checks and ordinal filters (less than, greater than, equal to) with combinations of logical (NOT, AND and OR) operators.

3.4 Data and Schema Evolution in Albis

As described so far, the name and location of a data file plays an important role to support data and schema evolution in Albis. We now describe this in detail:

Adding Rows: Adding another row is trivial. A writer adds another row group in the directory and writes its data out. Appending to an existing row group is also possible on append-only file systems like HadoopFS. However, while adding another row, the writer cannot alter the column grouping configuration.

Deleting Rows: Deleting rows in-place is not supported as the underlying file system (HDFS) is append-only.

Adding Columns: Adding new columns is one of the most frequent operations in analytic. Adding a column at the end of the schema involves creating a column group directory (with associated data and metadata files). The ordering of row data in the newly added column is matched with the existing data, and missing row entries are marked null. Using this strategy, more than one column (as a CG) can be added at a time as well. The old schema file is read, and written again (after deleting the old one) with the updated schema.

Deleting Columns: A column delete operation in Albis falls in one of the two categories. The column(s) to be deleted is (are) either (i) stored as a separate CG; or (ii) stored with other columns. In the first case, the deletion operation is simple. The CG directory is deleted, and the schema is updated as mentioned previously. In the latter case, there are two ways Albis deletes columns. A light-weight deletion operation "marks" the column as deleted and updates the schema. The column is only marked as deleted, but the column is not removed from

the schema because the column data is still stored in the storage. In order to skip the marked column data, an `AlbisReader` needs to know the type(s) of the deleted column(s). In contrast, a heavy-weight delete operation emulates a read-modify-write cycle, where the CG is read, modified, and written out again.

Maintenance Operations: Apart from the aforementioned operations, Albis supports two maintenance operations: *re-grouping* and *re-balancing*. Re-grouping is for re-configuring the column grouping. For example, due to the evolution in the workload it might be necessary to split (or merge) a CG into multiple CGs for a faster filter or projection processing. Re-balancing refers to re-distributing the data between RGs. A RG is the unit of processing for Albis tables in a distributed setting. Adding and removing column and row data can lead to a situation where data between row-groups is not balanced. This data imbalance will lead to imbalanced compute jobs. The Re-balancing operation reads the current data and re-balances the data distribution in the row groups. While executing the re-balancing, it is possible to increase the number of row groups to increase the parallelism in the system. Re-grouping can be executed at the same time as well. These operations are slow and we expect them to be executed once-in-a-while. Even though column adding and deleting is one of the frequent operations, a complete re-balancing is only required if the added columns/rows contain highly uneven values.

3.5 Distributed Processing Concerns

How does an RDPS system process input Albis files? RDPS frameworks like Spark/SQL divide work among the workers using the size as a criteria. At a first glance, a segment seems to be a perfect fit for providing equal sized work items for workers. For a static table, segments *can* be used as the quantum of processing. However, as a new column is added, often as a result of distributed processing, it is critical in what order the rows in the new column are written because indexes are implicitly encoded with the data position in a file. For example, imagine a table with two segments in a single row-group. Now, another column is added to this table using two Spark/SQL workers, each processing one segment. As there are no ordering guarantees between tasks, and each task in the data-parallel paradigm gets its own file to write, it is possible that the first task gets the second segment, but the first new column file name. This mix-up destroys the row ordering when enumerating files based on their names. However, if the whole row-group is processed only by a single task, the newly added "single" column file is ensured to have the same ordering as the current row file (including all its segments). Thus, an Albis row-group is the unit of processing for a distributed RDPS system. A single task is responsible for processing, adding and deleting columns and rows within a single row group while maintaining the implicit ordering variant of the data. As previously discussed, if necessary, the data can be re-balanced to increase the number of row-groups, hence, parallelism in the system. The schema file updates are expected to take place on a centralized node like the Spark driver.

Which column grouping to use? The recommended column grouping setting depends upon the workload. Systems like H2O [14], etc., can change the storage format dynamically based upon the workload. We consider this work beyond the current scope. However, we expect that Albis can help in this process by providing meaningful insights about accesses and I/O patterns.

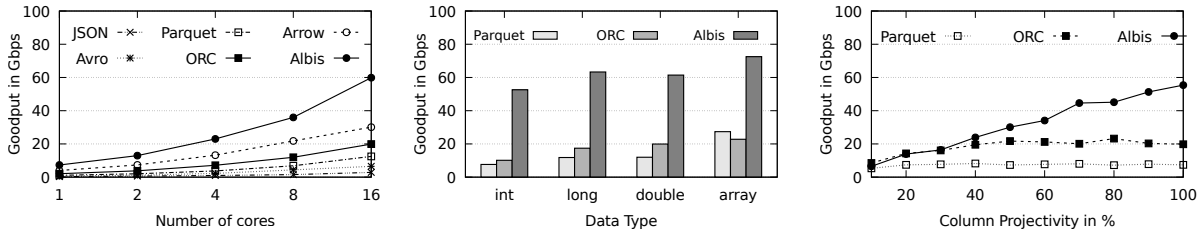
4 Evaluation

Albis is implemented in about 4k lines of Scala/Java code for Java 8. We evaluate the performance of Albis on a 4-node cluster each containing dual Xeon E5-2690 CPUs, 128GB of DDR3 DRAM, 4 enterprise-grade NVMe devices, connected via a 100 Gbps link, running Ubuntu 16.04. All numbers reported here are the average of 3 runs. We attempt to answer three fundamental questions:

- *First, does Albis deliver data access rates close to the modern networking and storage devices?* Using a set of micro-benchmarks over HDFS on 100 Gbps network and NVMe devices we demonstrate that Albis delivers read bandwidths up to 59.9 Gbps, showing a gain of $1.9-21.4\times$ over other file formats (Section 4.1). With Apache Crail (instead of HDFS), Albis delivers bandwidth of 85.5 Gbps from NVMe devices.
- *Secondly, does Albis accelerate the performance of data analytic workloads?* To demonstrate the effectiveness of Albis and its API, we have integrated it in Spark/SQL, and demonstrate an up to $3\times$ reduction in query runtimes. The overall TPC-DS runtime is also decreased by 25.4% (Section 4.2).
- *Lastly, what is the cost of design trade-offs of Albis, namely the cost of eschewing compression and increased look-up cost in a distributed system?* In our evaluation, Albis increases the storage cost by a margin of $1.3-2.7\times$ (based on the compression choice), but does not increase the load for extra block lookups. In exchange, it delivers performance improvements by a margin of $3.4-7.2\times$ (Section 4.3).

4.1 Micro-benchmarks

In this section, we evaluate the performance of Albis, through a series of micro-benchmarks. We focus on the



(a) (b) (c)

Figure 5: (a) Bandwidth vs. core scaling; (b) Effect of data type on performance; (c) Projectivity performance.

read because the write performance of all file formats are bottlenecked by the HDFS write bandwidth (∼20 Gbps).

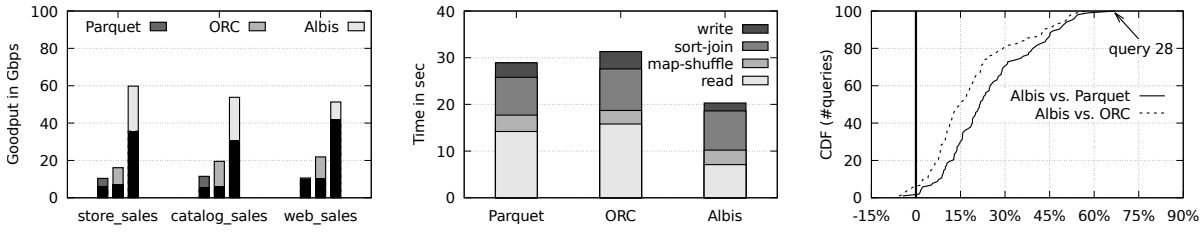
Read performance: We start by revisiting the key experiment from the beginning of the paper. Here, in Figure 5a we show the performance of Albis with respect to other file formats for reading the `store_sales` data. The x-axis shows the number of cores and the y-axis shows the data goodput performance. As shown, in comparison to other data formats, Albis delivers 1.9 (vs. Arrow) – 21.4× (vs. JSON) better read performance for reading the `store_sales` table. The performance gains of Albis can be traced down to its superior instruction utilization (due to its light-weight software stack), and cache profile. Table 5 shows the CPU profile of Albis against Parquet, ORC, and Arrow. As can be seen, Albis takes $1.2-4.1\times$ less instructions, and exhibits $1.5-3.0\times$ fewer cache misses per row. The peak data rate is at 59.9 Gbps, which is within 80% of the HDFS bandwidth. The gap between the HDFS performance and Albis is due to the parsing of schema and materialization of the data. For the sake of brevity, in the following sections we focus our effort on best performing formats, namely Parquet and ORC, for the performance evaluation. Arrow does not have native filter and projection capabilities.

Effect of schema and data types: We now focus our effort to quantify what effect a data type has on the read performance. We choose integers, longs, doubles, and arrays of byte types. For this benchmark, we store 10 billion items of each type. For the array, we randomly generate an array in the range of (1–1024) bytes. Figure 5b shows our results. The key result from this experiment is that Albis's performance is very close to the expected results. The expected result is calculated as what fraction of incoming data is TPC-DS data. As discussed in Section 3.1, each row contains an overhead of the bitmap. For a single column schema that we are testing, it takes 1 byte for the bitmap. Hence, for integers we expect $^4/_{5^{th}}$ of the HDFS read performance. In our experiments, the integer performance is 52.6 Gbps which is within 87.8% of the expected performance (59.9 Gbps). Other types also follow the same pattern. The double values are slower to materialize than longs. We are not sure about the cause of this behavior. The byte array schema delivers bandwidth very close to the HDFS read bandwidth as the bitmap overhead is amortized. With arrays, Albis delivers 72.5 Gbps bandwidth. We have only shown the performance of primitive types as higher-level types such as `timestamps`, `decimal`, or `date` are often encoded into the lower-level primitive types, and their materialization is highly implementation-dependent.

	Parquet	ORC	Arrow	Albis
Instructions/row	6.6K	4.9K	1.9K	1.6K
Cache-misses/row	9.2	4.6	5.1	3.0
Nanosecs/row	105.3	63.9	31.2	20.8

Table 5: Micro-architectual analysis for Parquet, ORC, Arrow, and Albis on a 16-core Xeon machine.

Projection performance: A columnar-format is a natural fit when performing a projection operation. A similar operation in a row-oriented format requires a level of redirection to materialize the values while leaving the unwanted data behind. However, a distinction must be made between a true column-oriented and PAX-alike format (e.g., Parquet). The PAX-encoding does not change the I/O pattern and amount of data read from a file system. It only helps to reduce the amount of work required (i.e., mini-joins) to materialize the projected columns. Albis's efficiency in projection depends upon the column grouping configuration. With a single column group, Albis is essentially a row store. Hence, the complete row data is read in, but only desired columns are materialized. To evaluate the projection performance, we generated a dataset with 100 integer columns and 100 million rows (total size: 40 GB). This dataset is then read in with a variable projectivity, choosing k out of 100 columns for k% projectivity. Figure 5c shows the projectivity (x-axis) and the goodput (y-axis). As shown, Albis (as a row store) always outperforms Parquet and ORC after 30% projectivity. It is only slower than ORC for 10% and 20% projectivity by a margin of 23.5% and 2.7%, respectively. We exercise caution with results as the superiority of row versus column format is a contentious topic, and gains of one over the other come from a variety of



(a) (b) (c)

Figure 6: Albis performance results: (a) read bandwidths (with Spark iterator in dark) of file formats for TPC-DS tables; (b) performance of the EquiJoin on two 32GB data sets; (c) CDF for performance gains for TPC-DS queries.

features that go beyond just micro-benchmarks [12].

Selectivity performance: In a real-world workload, RDPS systems often have predicates on column values. As all file formats maintain metadata about columns, they help the RDPS systems to pre-filter the input rows. However, it must be noted that often, filters are "*hints*" to file formats. A format is not expected strictly to return rows that satisfy the filters. Parquet, ORC, and Albis all maintain segment-level metadata that can be used to completely eliminate reading the segment. The performance saving depends upon the segment size. ORC also maintains metadata per 10k rows that allows it to do another level of filtering. In contrast, Albis supports strict filtering of values and hence, it avoids pushing unwanted rows into the processing pipeline. We evaluate selectivity performance on the same 100 integer column table used in the projection. The first column of the table contains integers between 0 and 100. We execute a simple SQL query "*select(*) from table where col0 <= k*", where k varies from 1 to 100, to select k% of input rows. Our results demonstrate similar gains (not shown) as the projection performance. Albis outperforms Parquet and ORC by a margin of $1.6-2.4\times$.

4.2 Workload-level Performance

For workload-level experiments, we have integrated Albis support into Spark/SQL [15] (`v2.2`) and evaluate its performance for an EquiJoin and the full TPC-DS query set. Naturally, input and output is only one aspect of a workload, and gains solely from a fast format are bounded by the CPU-I/O balance of the workload.

Spark/SQL data ingestion overheads: Integration into Spark/SQL entails converting incoming row data into Spark-specific format (the `Iterator[InternalRow]` abstraction). We start by quantifying what fraction of performance is lost due to framework-related overheads [54], which, of course, varies with the choice of the SQL framework. In Figure 6a, we show the read bandwidths for the three biggest tables in the TPC-DS dataset for ORC, Parquet, and Albis[3]. In each bar, we also show a dark bar that represents the performance observed at the Spark/SQL level. In general, from one third up to half of the performance can be lost due to framework-related overheads. The table `web_sales` performs the best with 89.2% of Parquet bandwidth delivered. However, it can be observed the other way around as well because the Parquet bandwidth is so low, hence, further overheads from the framework do not deteriorate it further.

Effect of the Binary API: While measuring the Spark ingestion rate, we also measure the number of live objects that the Java heap manages per second while running the experiment. For Albis we use two modes to materialize Spark's `UnsafeRow` either using the binary API or not. In our evaluation we find (not shown) that even without using the binary API, Albis (260.4K objs/sec) is better than Parquet (490.5K objs/sec) and ORC (266.4K objs/sec). The use of binary API futher improves Albis's performance by 4.3% to 249.2K objs/sec.

EquiJoin performance: Our first SQL benchmark is an EquiJoin operation, which is implemented as a Sort-Merge join in Spark/SQL. For this experiment, we generate two tables with a `<int,byte[]>` schema and 32 million rows each. The array size varies between 1 and 2kB. The total data set is around 64GB in two tables. The join operation joins on the `int` column, and then generates the checksum column for merged `byte[]` columns, which is written out. Figure 6b shows our results. The figure shows the runtime splits (y-axis) for the 4 stages of the join operation (reading in, mapping to partitions, sorting and joining partitions, and then the final write out) for Parquet, ORC, and Albis. As shown, Albis helps to reduce the read (7.1 sec for Albis) and write (1.7 sec for Albis) stages by more than $2\times$. Naturally, a file format does not improve the performance of the map and join stages, which remain constant for all three file formats. Overall, Albis improves the query run-time by 35.1% and 29.8% over ORC and Parquet, respectively.

TPC-DS performance: Lastly, we run the TPC-DS query set on the three file formats with the scaling factor

[3] Spark/SQL does not support the Arrow format yet (v2.2).

	None	Snappy	Gzip	zlib
Parquet	58.6 GB	44.3 GB	33.8 GB	N/A
	12.5 Gbps	9.4 Gbps	8.3 Gbps	-
ORC	72.0 GB	47.6 GB	N/A	36.8 GB
	19.1 Gbps	17.8 Gbps	-	13.0 Gbps
Albis	94.5 GB	N/A	N/A	N/A
	59.9 Gbps	-	-	-

Table 6: TPC-DS dataset sizes and performance.

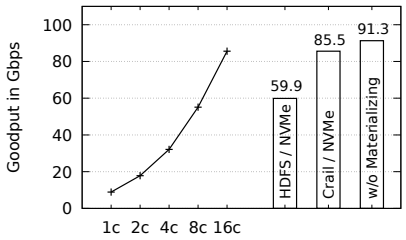


Figure 7: Albis performance on Crail with NVMeF.

of 100. We choose the factor 100 as it is the largest factor that we can run while keeping the shuffle data in the memory to avoid Spark's shuffle-related overheads. Figure 6c shows our results. On the y-axis, the figure shows the fraction of queries as CDF and on the x-axis it shows percentage performance gains for Albis in comparison to Parquet and ORC formats. There are two main observations here. First, for more than half of the queries, the performance gains are less than 13.8% (ORC) and 21.3% (Parquet). For 6 queries on ORC (only 1 on Parquet), the gains are even negative, however, the loss is small (-5.6%). Second, the last six queries on both file formats see more than a 50% improvement in run-times. The run-time of the query 28, which is the query with most gains, is improved by a margin of $2.3\times$ and $3.0\times$ for ORC and Parquet, respectively. Gains are not uniformly distributed among all queries because they depend upon what fraction of the query time is I/O bounded and what is CPU bounded (including framework related overheads). The run-times of the full TPC-DS suit with all queries is $1,850.1$ sec (Parquet), $1,723.4$ sec (ORC) and $1,379.2$ sec (Albis), representing a gain of 25.4% (over Parquet) and 19.9% (over ORC).

4.3 Efficiency of Albis

The cost of compression: With its sole focus on performance, Albis does not use any compression or encoding to reduce the data set size. This design choice means that Albis file sizes are larger than other file formats. In Table 6, we show the TPC-DS dataset (scale=100) sizes with compression options available on Parquet and ORC. As calculated from the table, due to highly efficient type-specific encoding (e.g. RLE for integers), even uncompressed Parquet and ORC datasets are $23.8-37.9\%$ smaller than Albis's. With compression, the gap widens to 64.2%. With the current market prices, the increased space costs 0.5$/GB on NVMe devices. However, the compressed dataset sizes also lead to significant performance loss when reading the dataset (also shown in the table). As we have shown in Section 2, the reading benchmarks are CPU bounded even without compression. Hence, adding additional routines to decompress incoming data only steals cycles from an already saturated CPU. In comparison to compressed bandwidths, Albis delivers $3.4-7.2\times$ higher bandwidths. One potential avenue to recover the lost storage efficiency is to leverage advanced hardware-accelerated features like deduplication and block compression further down the storage stack. However, we have not explored this avenue in detail yet.

Load on the distributed system: One concern with the increased number of files and the storage capacity of data sets is that they increase RPC pressure on the namenode. The number of RPCs to the namenode depends upon the number of files and blocks within a file. With an HDFS block size of 128 MB, the most efficient dataset from TPC-DS takes 271 blocks (33.8 GB with Parquet and gzip). In comparison, Albis's dataset takes 756 blocks. The lookup cost increase for hundreds of blocks is marginal for HDFS. Nonetheless, we are aware of the fact that these factors will change with the scale.

Delivering 100 Gbps bandwidth: For our final experiment, we try to answer the question what it would take to deliver 100 Gbps bandwidth for Albis. Certainly, the first bottleneck is to improve the base storage layer performance. The second factor is to improve the data density. For example, the `store_sales` table on Albis has the data density of 93.9% (out of 100 bytes read from the file system). On top of this, the effective bandwidth on 100 Gbps link is 98.8 Gbps, that gives us the upper bounds for the performance at 92.8 Gbps that we hope to achieve with the Albis `store_sales` table on a 100 Gbps link. To test the peak performance, we port Albis to Apache Crail with its NVMeF tier [8, 52]. Crail is an Apache project that integrates high-performance network (RDMA, DPDK) and storage (NVMeF, SPDK) stacks in the Apache data processing ecosystem. The peak bandwidth of Crail from NVMe devices is 97.8 Gbps [46]. Figure 7 shows our results. In the left half of the figure it shows the scaling performance of Albis on Crail from 1 core performance (8.9 Gbps) to 16 cores (85.5 Gbps). In comparison, the right half of the figure shows the performance of HDFS/NVMe at 59.9 Gbps and Crail/NVMe at 85.5 Gbps. The last bar shows the performance of Albis if the benchmark does not materialize Java object values. In this configuration, Albis on

Crail delivers 91.3 Gbps, which is within 98.4% of the peak expected performance of 92.7 Gbps.

5 Related Work

Storage Formats in Databases: N-ary Storage Model (NSM) stores data records contiguously in a disk page, and uses a record offset table at the end of the page [47]. Decomposition Storage Model (DSM) [23] proposes to split an n-column relation into n sub-relations that can be stored independently on a disk. NSM is considered good for transactions, whereas, DSM is considered suitable for selection and projection-heavy analytical workloads. A series of papers have discussed the effect of NSM and DSM formats on the CPU efficiency and query execution [27, 55, 12]. Naturally, there is no one size that fits all. Ailamaki et al. [13] propose the Partition Attributes Across (PAX) format that combines the benefits of these two for a superior cache performance. The Fractured Mirrors design proposes maintaining both NSM and DSM formats on different copies of data [48]. Jindal et al. propose the Trojan data layout that does workload-driven optimizations for data layouts within replicas [32]. The seminal work from Abadi et al. demonstrates that a column-storage must be augmented by a right query processing strategy with late materialization, batch processing, etc., for performance gains [12]. Various state of the art database systems have been proposed that take advantage of these strategies [51, 19]. In comparison so far, the focus of Albis has been on a *lightweight* file format that can faithfully reflect the performance of the storage and networking hardware.

File Formats in the Cloud: Many parts of the data format research from databases have found its way into commodity, data-parallel computing systems as well. Google has introduced SSTable, an on-disk binary file format to store simple immutable key-value strings [22]. It is one of the first external file formats used in a large-scale table storage system. RCFile [28] is an early attempt to build a columnar store on HDFS. RCFiles do not support schema evolution and have inefficient I/O patterns for MapReduce workloads. To overcome these limitations, Floratou et al. propose the binary columnar-storage CIF format for HDFS [25]. However, in order to maintain data locality, they require a new data placement policy in HDFS. Hadoop-specific column-storage issues like column placement and locality are discussed in detail by [30, 25]. The more popular file formats like Parquet [10] (uses Dremel's column encoding [38]) and ORC [9], etc., are based on the PAX format. Albis's column grouping and row-major storage format match closely with Yahoo's Zebra [4, 39]. However, Zebra does not support filter pushdown or statistics like Albis. Apache CarbonData is an indexed columnar data format for fast analytics on big data platforms [7]. It shares similarities with the Arrow/Parquet project. However, due to its intricate dependencies on Spark, we could not evaluate it independently. Historically, the priorities of these file formats have been I/O efficiency (by trading CPU cycles) and then performance, in that order. However, as we have demonstrated in this paper, the performance of these file formats are in dire need of revision.

High-Performance Hardware: Recently, there has been a lot of interest in integrating high-performance hardware into data processing systems [17, 18]. Often, the potential performance gains from modern hardware are overshadowed by the thick software/CPU stack that is built while holding the decades old I/O assumptions [40, 54]. This pathology manifests itself as "system being CPU-bounded", even for many I/O-bound jobs [20, 43]. A natural response to this situation is to add more resources, which leads to a significant loss in efficiency [37]. In this work, we have shown that by re-evaluating the fundamental assumptions about the nature of I/O and CPU performances, we can build efficient and fast systems - a sentiment echoed by the OS designers as well [45]. Recently, Databricks has also designed its optimized caching format after finding out about the inadequate performance of file formats on NVMe devices [36]. However, details of the format are not public.

6 Conclusion

The availability of high-performance network and storage hardware has fundamentally altered the performance balance between the CPU and I/O devices. Yet, many assumptions about the nature of I/O are still rooted in the hardware of the 1990s. In this paper, we have investigated one manifestation of this situation in the performance of external file formats. Our investigation on 100 Gbps network and NVMe devices reveals that due to the excessive CPU and software involvement in the data access path, none of the file formats delivered performance close to what is possible on modern hardware. Often, CPU cycles are traded for storage efficiency. We have presented Albis, a light-weight, high-performance file format. The key insight in the design of Albis is that by foregoing the assumptions and re-evaluating the CPU-I/O work division in the file format, it is possible to build a high-performance *balanced* system. In the process of designing Albis, we have also presented an extensive evaluation of popular file formats on modern high-performance hardware. We demonstrate that Albis delivers performance gains in the order of $1.9 - 21.4\times$; superior cache and instruction profile; and its integration in Spark/SQL shows TPC-DS queries acceleration up to a margin of $3\times$. Encouraged by this result, we are exploring applicability of Albis with multiple frameworks.

References

[1] Amazon EC2 I3 Instances: Storage optimized for high transaction workloads. https://aws.amazon.com/ec2/instance-types/i3/. [Online; accessed Jan-2018].

[2] AWS News Blog, Elastic Network Adapter - High Performance Network Interface for Amazon EC2. https://aws.amazon.com/blogs/aws/elastic-network-adapter-high-performance-network-interface-for-amazon-ec2/. [Online; accessed Jan-2018].

[3] Microsoft Azure: High performance compute VM sizes, RDMA-capable instances. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-hpc#rdma-capable-instances. [Online; accessed Jan-2018].

[4] Pig 0.8 Documentation, Zebra Overview. https://pig.apache.org/docs/r0.8.1/zebra_overview.html. [Online; accessed Jan-2018].

[5] Apache Arrow - Powering Columnar In-Memory Analytics. https://arrow.apache.org/, 2017. [Online; accessed Jan-2018].

[6] Apache Avro. https://avro.apache.org/, 2017. [Online; accessed Jan-2018].

[7] Apache CarbonData. http://carbondata.apache.org/index.html, 2017. [Online; accessed Jan-2018].

[8] Apache Crail (Incubating) - High-Performance Distributed Data Store. http://crail.incubator.apache.org/, 2017. [Online; accessed Jan-2018].

[9] Apache ORC - High-Performance Columnar Storage for Hadoop. https://orc.apache.org/, 2017. [Online; accessed Jan-2018].

[10] Apache Parquet. https://parquet.apache.org/, 2017. [Online; accessed Jan-2018].

[11] JSON (JavaScript Object Notation). http://www.json.org/, 2017. [Online; accessed Jan-2018].

[12] ABADI, D. J., MADDEN, S. R., AND HACHEM, N. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada, 2008), SIGMOD '08, ACM, pp. 967–980.

[13] AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND SKOUNAKIS, M. Weaving Relations for Cache Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 2001), VLDB '01, Morgan Kaufmann Publishers Inc., pp. 169–180.

[14] ALAGIANNIS, I., IDREOS, S., AND AILAMAKI, A. H2o: A hands-free adaptive store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA, 2014), SIGMOD '14, ACM, pp. 1103–1114.

[15] ARMBRUST, M., XIN, R. S., LIAN, C., HUAI, Y., LIU, D., BRADLEY, J. K., MENG, X., KAFTAN, T., FRANKLIN, M. J., GHODSI, A., AND ZAHARIA, M. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia, 2015), SIGMOD '15, ACM, pp. 1383–1394.

[16] BARROSO, L., MARTY, M., PATTERSON, D., AND RANGANATHAN, P. Attack of the Killer Microseconds. *Commun. ACM 60*, 4 (Mar. 2017), 48–54.

[17] BARTHELS, C., LOESING, S., ALONSO, G., AND KOSSMANN, D. Rack-Scale In-Memory Join Processing Using RDMA. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia, 2015), SIGMOD '15, ACM, pp. 1463–1475.

[18] BINNIG, C., CROTTY, A., GALAKATOS, A., KRASKA, T., AND ZAMANIAN, E. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow. 9*, 7 (Mar. 2016), 528–539.

[19] BONCZ, P. A., MANEGOLD, S., AND KERSTEN, M. L. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK* (1999), pp. 54–65.

[20] CANALI, L. Performance Analysis of a CPU-Intensive Workload in Apache Spark. https://db-blog.web.cern.ch/blog/luca-canali/2017-09-performance-analysis-cpu-intensive-workload-apache-spark, 2017. [Online; accessed Jan-2018].

[21] CAULFIELD, A. M., COBURN, J., MOLLOV, T., DE, A., AKEL, A., HE, J., JAGATHEESAN, A., GUPTA, R. K., SNAVELY, A., AND SWANSON, S. Understanding the Impact of Emerging

Non-Volatile Memories on High-Performance, IO-Intensive Computing. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (Washington, DC, USA, 2010), SC '10, IEEE Computer Society, pp. 1–11.

[22] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington, 2006), OSDI '06, USENIX Association, pp. 205–218.

[23] COPELAND, G. P., AND KHOSHAFIAN, S. N. A Decomposition Storage Model. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data* (Austin, Texas, USA, 1985), SIGMOD '85, ACM, pp. 268–279.

[24] COSTEA, A., IONESCU, A., RĂDUCANU, B., SWITAKOWSKI, M., BÂRCA, C., SOMPOLSKI, J., ŁUSZCZAK, A., SZAFRAŃSKI, M., DE NIJS, G., AND BONCZ, P. VectorH: Taking SQL-on-Hadoop to the Next Level. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA, 2016), SIGMOD '16, ACM, pp. 1105–1117.

[25] FLORATOU, A., PATEL, J. M., SHEKITA, E. J., AND TATA, S. Column-oriented Storage Techniques for MapReduce. *Proc. VLDB Endow. 4*, 7 (Apr. 2011), 419–429.

[26] GUZ, Z., LI, H. H., SHAYESTEH, A., AND BALAKRISHNAN, V. NVMe-over-fabrics Performance Characterization and the Path to Low-overhead Flash Disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference* (Haifa, Israel, 2017), SYSTOR '17, ACM, pp. 16:1–16:9.

[27] HARIZOPOULOS, S., LIANG, V., ABADI, D. J., AND MADDEN, S. Performance Tradeoffs in Read-optimized Databases. In *Proceedings of the 32Nd International Conference on Very Large Data Bases* (Seoul, Korea, 2006), VLDB '06, VLDB Endowment, pp. 487–498.

[28] HE, Y., LEE, R., HUAI, Y., SHAO, Z., JAIN, N., ZHANG, X., AND XU, Z. RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering* (Washington, DC, USA, 2011), ICDE '11, IEEE Computer Society, pp. 1199–1208.

[29] HUAI, Y., CHAUHAN, A., GATES, A., HAGLEITNER, G., HANSON, E. N., O'MALLEY, O., PANDEY, J., YUAN, Y., LEE, R., AND ZHANG, X. Major Technical Advancements in Apache Hive. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA, 2014), SIGMOD '14, ACM, pp. 1235–1246.

[30] HUAI, Y., MA, S., LEE, R., O'MALLEY, O., AND ZHANG, X. Understanding Insights into the Basic Structure and Essential Issues of Table Placement Methods in Clusters. *Proc. VLDB Endow. 6*, 14 (Sept. 2013), 1750–1761.

[31] IBM. IBM InfoSphere BigInsights Version 3.0, File formats supported by Big SQL. `https://www.ibm.com/support/knowledgecenter/SSPT3X_3.0.0/com.ibm.swg.im.infosphere.biginsights.dev.doc/doc/biga_fileformats.html`, 2017. [Online; accessed Jan-2018].

[32] JINDAL, A., QUIANÉ-RUIZ, J.-A., AND DITTRICH, J. Trojan data layouts: Right shoes for a running elephant. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing* (Cascais, Portugal, 2011), SOCC '11, ACM, pp. 21:1–21:14.

[33] KEDIA, S., WANG, S., AND CHING, A. Apache Spark @Scale: A 60 TB+ production use case. `https://code.facebook.com/posts/1671373793181703/apache-spark-scale-a-60-tb-production-use-case/`, 2016. [Online; accessed Jan-2018].

[34] KLIMOVIC, A., LITZ, H., AND KOZYRAKIS, C. ReFlex: Remote Flash == Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China, 2017), ASPLOS '17, ACM, pp. 345–359.

[35] KORNACKER, M., BEHM, A., BITTORF, V., BOBROVYTSKY, T., CHING, C., CHOI, A., ERICKSON, J., GRUND, M., HECHT, D., JACOBS, M., JOSHI, I., KUFF, L., KUMAR, D., LEBLANG, A., LI, N., PANDIS, I., ROBINSON, H., RORKE, D., RUS, S., RUSSELL, J., TSIROGIANNIS, D., WANDERMAN-MILNE, S., AND YODER, M. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilo-*

mar, CA, USA, January 4-7, 2015, Online Proceedings (2015).

[36] LUSZCZAK, A., SZAFRANSKI, M., SWITAKOWSKI, M., AND XIN, R. Databricks Runtimes New DBIO Cache Boosts Apache Spark Performance: why NVMe SSDs improve caching performance by 10x. `https://databricks.com/blog/2018/01/09/databricks-runtimes-new-dbio-cache-boosts-apache-spark-performance.html`, 2018. [Online; accessed Jan-2018].

[37] MCSHERRY, F., ISARD, M., AND MURRAY, D. G. Scalability! but at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)* (Kartause Ittingen, Switzerland, 2015), USENIX Association.

[38] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: Interactive Analysis of Web-scale Datasets. *Proc. VLDB Endow. 3*, 1-2 (Sept. 2010), 330–339.

[39] MUNIR, R. F., ROMERO, O., ABELLÓ, A., BILALLI, B., THIELE, M., AND LEHNER, W. Resilientstore: A heuristic-based data format selector for intermediate results. In *Model and Data Engineering - 6th International Conference, MEDI 2016, Almería, Spain, September 21-23, 2016, Proceedings* (2016), pp. 42–56.

[40] NANAVATI, M., SCHWARZKOPF, M., WIRES, J., AND WARFIELD, A. Non-volatile Storage. *Queue 13*, 9 (Nov. 2015), 20:33–20:56.

[41] ORACLE. Oracle Big Data SQL Reference. `https://docs.oracle.com/cd/E55905_01/doc.40/e55814/bigsqlref.htm`, 2017. [Online; accessed Jan-2018].

[42] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., RUMBLE, S., STUTSMAN, R., AND YANG, S. The RAMCloud Storage System. *ACM Trans. Comput. Syst. 33*, 3 (Aug. 2015), 7:1–7:55.

[43] OUSTERHOUT, K., RASTI, R., RATNASAMY, S., SHENKER, S., AND CHUN, B.-G. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA, 2015), NSDI'15, USENIX Association, pp. 293–307.

[44] PETER, S., LI, J., WOOS, D., ZHANG, I., PORTS, D. R. K., ANDERSON, T., KRISHNAMURTHY, A., AND ZBIKOWSKI, M. Towards High-performance Application-level Storage Management. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems* (Philadelphia, PA, 2014), HotStorage'14, USENIX Association.

[45] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO, 2014), OSDI'14, USENIX Association, pp. 1–16.

[46] PFEFFERLE, J. Apache Crail Storage Performance – Part II: NVMf. `http://crail.incubator.apache.org/blog/2017/08/crail-nvme-fabrics-v1.html`, 2017. [Online; accessed Jan-2018].

[47] RAMAKRISHNAN, R., AND GEHRKE, J. *Database Management Systems, Chapter 9.6 Page Formats*, 3rd ed. McGraw-Hill, Inc., 2003.

[48] RAMAMURTHY, R., DEWITT, D. J., AND SU, Q. A Case for Fractured Mirrors. In *Proceedings of the 28th International Conference on Very Large Data Bases* (Hong Kong, China, 2002), VLDB '02, VLDB Endowment, pp. 430–441.

[49] RUDNYTSKIY, V. Loading sample data from different file formats in SAP Vora. `https://www.sap.com/developer/tutorials/vora-zeppelin-load-file-formats.html`, 2017. [Online; accessed Jan-2018].

[50] SCHUEPBACH, A., AND STUEDI, P. Apache Crail Storage Performance – Part III: Metadata. `http://crail.incubator.apache.org/blog/2017/11/crail-metadata.html`, 2017. [Online; accessed Jan-2018].

[51] STONEBRAKER, M., ABADI, D. J., BATKIN, A., CHEN, X., CHERNIACK, M., FERREIRA, M., LAU, E., LIN, A., MADDEN, S., O'NEIL, E., O'NEIL, P., RASIN, A., TRAN, N., AND ZDONIK, S. C-store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases* (Trondheim, Norway, 2005), VLDB '05, VLDB Endowment, pp. 553–564.

[52] STUEDI, P., TRIVEDI, A., PFEFFERLE, J., STOICA, R., METZLER, B., IOANNOU, N., AND KOLTSIDAS, I. Crail: A High-Performance I/O

Architecture for Distributed Data Processing. *IEEE Bulletin of the Technical Committee on Data Engineering 40*, 1 (March 2017), 40–52.

[53] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow. 2*, 2 (Aug. 2009), 1626–1629.

[54] TRIVEDI, A., STUEDI, P., PFEFFERLE, J., STOICA, R., METZLER, B., KOLTSIDAS, I., AND IOANNOU, N. On The [Ir]relevance of Network Performance for Data Processing. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (Denver, CO, 2016), USENIX Association.

[55] ZUKOWSKI, M., NES, N., AND BONCZ, P. DSM vs. NSM: CPU Performance Tradeoffs in Block-oriented Query Processing. In *Proceedings of the 4th International Workshop on Data Management on New Hardware* (Vancouver, Canada, 2008), DaMoN '08, ACM, pp. 47–54.