Column Cache: Buffer Cache for Columnar Storage on HDFS

Takeshi Yoshimura
IBM Research - Tokyo
Tokyo, Japan
tyos@jp.ibm.com

Tatsuhiro Chiba
IBM Research - Tokyo
Tokyo, Japan
chiba@jp.ibm.com

Hiroshi Horii
IBM Research - Tokyo
Tokyo, Japan
horii@jp.ibm.com

***Abstract*—Columnar storage is a data source for data analytics in distributed computing frameworks. For portability and scalability, columnar storage is built on top of existing distributed file systems with columnar data representations such as Parquet, RCFile, and ORC. However, these representations fail to utilize high-level information (e.g., columnar formats) for low-level disk buffer management in operating systems. As a result, data analytics workloads suffer from redundant memory buffers with expensive garbage collections, unnecessary disk readahead, and cache pollution in the operating system buffer cache.**

We propose *column cache*, which unifies and re-structures the buffers and caches of multiple software layers from columnar storage to operating systems. Column cache leverages high-level information such as file formats and query plans for enabling adaptive disk reads and cache eviction policies. We have developed a column cache prototype for Apache Parquet and observed that our prototype reduced redundant resource utilization in Apache Spark. Specifically, with our prototype, Spark showed a maximum speedup of 1.28x in TPC-DS workloads while increasing Linux page cache size by 18%, reducing total disk reads by 43%, and reducing garbage collection time in a Java virtual machine by 76%.

I. Introduction

As data volumes increase, large-scale distributed data processing systems have been progressively more used in many data-centric applications. Hadoop is a de facto standard of scale-out architecture in the massive data processing world, and these days, various workloads including batch processing and interactive analytics such as machine learning and relational query processing are running on top of Hadoop ecosystems. Due to changes in the characteristics of these applications, it is increasingly important to consider how data is managed in the system and what data format is optimal for applications in order to swiftly deal with massive data.

In terms of data storage format for analytical read-mostly query processing, columnar data representation has significant advantages over the row-wise format in many aspects as studied in RDBMS [1]. For example, columnar storage can filter unused columns when loading large data into memory. This feature increases the chance to parallelize data processing on in-memory computing. Dremel [2] opens up columnar storage in massive data processing systems, and successors such as Parquet [3], RCFile [4], and ORC [5] are utilized in many systems today.

Since data processing systems consist of multiple software layers (e.g., operating systems (OSs), frameworks, runtime, and many libraries), data itself is also abstracted in multiple layers. We call this phenomenon *nested data abstractions* in this paper. While multi-layered data management systems try to achieve good performance with their own policies, this optimization often has a negative impact on the performance of the application. In the case of the Parquet example, shown in Fig. 1, applications must go through multiple data abstraction layers to retrieve underlying data stored in local files. In this process, OSs generally retain a page cache for the files, the Java virtual machines (JVMs) preserve a heap for the buffered input stream, the Hadoop distributed file system (HDFS) [6] may keep the data in memory for repeated access, and so on. These inefficient and non-coordinated data buffering and caching logics cause the following three problems.

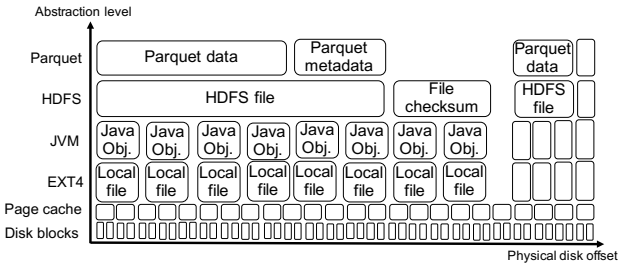

Fig. 1. **Nested data abstractions in Parquet.**

First, the multiple software layers for data management use redundant buffers. In big data processing, increased memory usage by redundant buffers can cause performance issues. Moreover, on columnar storage written in Java, redundant buffers also lead to frequent JVM heap memory allocations, which increases the CPU time of garbage collections (GCs) [7], [8].

Second, common heuristics in OSs causes inefficient disk accesses for columnar storage. Although the columnar storage has high-level information such as file formats, the underlying OS does not leverage the information for disk reads. We observed that heuristics in block I/O management degrades the disk I/O throughput by splitting block I/O requests into the configured length of disk readahead, which is too small for column reads.

Third, columnar storage can pollute the system-wide cache in OSs. The primary users of columnar storage on HDFS such

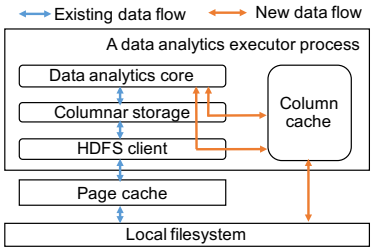

Fig. 2. **Column cache data flows in a data analytics.** Data flows of a column read after the data analytics obtains a file handler from HDFS servers is shown.

as Spark [9] and Tez [10] utilize page cache in OSs for their shuffle processing. Columnar storage also has two different access patterns in its data: main data is infrequently accessed, while metadata is frequently accessed. Even so, page cache in OSs manages them with the same cache eviction policy. In particular, main data accesses in columnar storage pollute the page cache and then slow down other file reads in the shuffle processing of data analytics. Data analytics provides rich information on the order of data reads in advance as query plans, but the cache eviction policy cannot leverage such information.

In this paper, we present a new buffer cache, *column cache*, to solve the major drawbacks of nested data abstractions. As shown in Fig. 2, column cache unifies and re-structures redundant buffers and caches in the multiple layers into a single software module. Regardless of the unified architecture, column cache does not need to modify HDFS *servers* and OSs. Column cache uses direct I/O [11] to read local files without page cache interleaving. It preserves the fault tolerance that HDFS servers and the underlying OS offer.

Column cache stores main data and metadata in different ways on the basis of high-level information such as file formats and query plans. The column cache reads the exact size of main data in accordance with the file formats. With the information of query plans, column cache estimates the number of scans for each main data in advance. Then, it can release the cached main data immediately after finishing the number of main data scans. Column cache monitors system memory and keeps the cached data in memory as long as possible if the system memory has enough space for page cache in the OS.

In this work, we optimize Parquet file reads in Spark on HDFS. We modified 88 lines of code in Spark to call our custom Parquet reader and to pass query plans to column cache. We believe we can easily extend column cache to support other columnar formats such as ORC, which is often used with Tez. We do not assume any unique features of Spark and Parquet such as supports for nested data structures (originally introduced in Dremel [2]). Moreover, we require less or no effort for the backward compatibilities because HDFS and Parquet already have well-defined file formats and communication protocols to support older versions of them.

Our evaluation showed that column cache exhibited better usages of computing resources: it increased page cache size by 18%, reduced total disk reads by 43%, and reduced GC time by 76% in the best cases of multi-stream TPC-DS [12] with a scale factor of 1,000 GB. As a result of better resource utilization, we observed up to 1.28x speedups on six-node clusters with 32 CPU cores and 128 GB RAM.

The first contribution of this paper is to analyze the drawbacks of nested data abstractions in Parquet. Although the basis of our initial motivation is similar to HBase layered design issues that existing work has pointed out [13], we provide a more detailed study of the unique drawbacks due to losing high-level information in terms of the buffer and cache efficiency in columnar storage.

The second contribution of this paper is to provide a solution to inefficient buffer and cache in Parquet due to its nested data abstractions. We found that by using the rich information of columnar storage and data analytics workloads, column cache could predict most of the sizes and patterns of disk reads, unlike other buffer caches for general workloads. The major difference from the existing functionality that bypasses OS in database [14], [15], [16] is that column cache also avoids memory pressure in OS page cache that data processing other than reading columnar storage uses. As a result, column cache shows better performance in shuffle-heavy queries in Spark SQL.

The third contribution is to show and solve the issues specific to JVM runtime. Columnar storage frequently allocates JVM heap memory and incurs time-consuming GCs. Our prototype results imply that buffer and cache schemes in the columnar storage are also a cause of GCs in Spark SQL, known as a CPU bottleneck workload [7].

In Section II, we analyze and describe examples of concrete issues pertaining to nested data abstractions on Parquet as our initial motivation for this work. Then, Section III shows the design and concept of column cache for efficient columnar storage on distributed file systems. Section IV reports the implementation details of column cache for Parquet on top of HDFS. We evaluate column cache with Spark 2.2.0 on our microbenchmark and TPC-DS in Section V. Related work is presented in Section VI. We conclude with a brief summary in Section VII.

II. Analysis of Parquet Files

In this section, we analyze file access patterns and discuss page cache effects when Apache Spark runs TPC-DS with a 1 TB data set (a scale factor: 1,000 GB). We use Apache Parquet, a popular columnar format for Spark applications. In our analysis, five slave servers of 32 virtual cores store the data set with the format and process SQL with their 128 GB RAM and 1 TB disk.

A. Access Patterns

First, we identify typical patterns to access files formatted with Parquet. A Parquet format is a columnar format on top of HDFS, which stores an HDFS file as multiple OS files

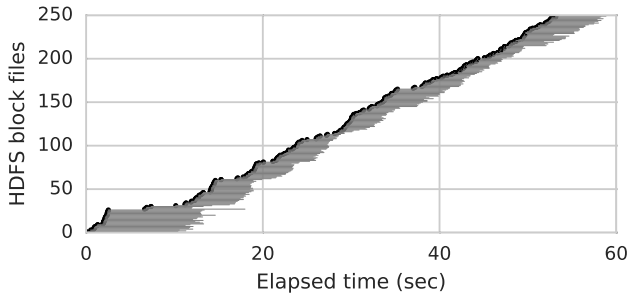

Fig. 3. **Access pattern of Parquet files.** The access patterns of 250 Parquet files are gathered and stacked. Gray line represents time for sequential accesses of a file and black circle represents non-sequential accesses.

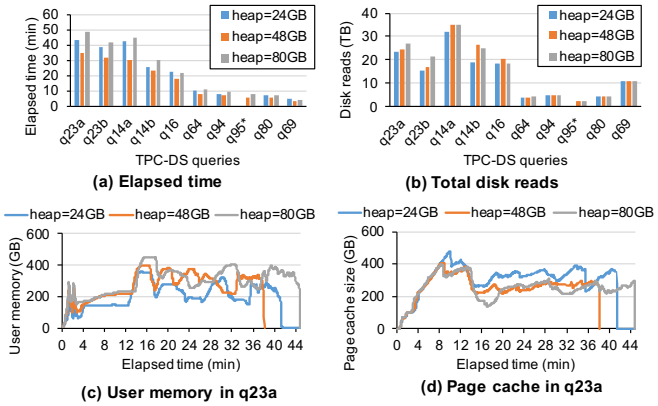

Fig. 4. **Performance effects of heap configurations.** (a) and (b) show the processing time and the total bytes of disk reads for ten queries. (c) and (d) show the time-series trace of user memory and page cache in query q23a. Query q95* caused out-of-memory errors in a 24-GB heap configuration.

in a directory with block numbers (e.g., blk_123456). We added instruments into the Linux kernel to record page cache behavior for a file descriptor and an offset in each file access and ran query q69 of TPC-DS a representative query that heavily loads table records stored with the Parquet format on HDFS.

Fig. 3 shows durations of file accesses for 250 files in an HDFS file formatted with Parquet while running the query q69. A gray line and a black circle respectively represent sequential and non-sequential access. We sorted the two types of accesses with block numbers of an HDFS file and access time. We observed three characteristics from this analysis.

First, Spark reads files with non-sequential access only at the start and with sequential access almost always. In the non-sequential access, we found that Spark scans the footer region in a Parquet file and determines its structure. In the Parquet format, records are stored in a file in columns rather than rows. The first stored data is the first column of the first record and the second stored data is the first column of the second record. The data of the second column follows all the data in the first column. The footer of a Parquet file informs offsets of the boundaries of columns. By using the offsets, Spark can avoid loading unnecessary data.

Even though Spark attempts to load only necessary data in a Parquet file, the OS may load unnecessary data in the file through speculative disk reads. For example, with readahead configuration, the OS reads more on-disk data than the application requests. Large readahead will improve the performance of sequential access, but increase redundancy of file reads.

Second, Spark reads a Parquet file only once in most cases. As shown in Fig. 3, once a sequential access finishes for a file, no more accesses happen for that file. We frequently observed this phenomenon not only in the query q69 but also in the other queries of TPC-DS.

Because Spark reads a Parquet file only once in processing a query, page cache for the file rarely work performance. In typical data analytics workloads, sizes of scanned tables in a query are larger than system memory. Therefore, cache pollution will occur by refreshing the page cache with Parquet files, which will not be reused.

Third, Spark accesses multiple Parquet files in parallel. In this experiment, we observed that Spark allocates 32 tasks to scan 32 Parquet files at the same time. Spark provides the number of virtual cores as a configuration to specify its task concurrency and we configured the value 32 to be the same as the number of cores. This task concurrency is important for data processing in Spark. After data is loaded, Spark performs CPU-intensive operations in query processing, such as sorting and grouping records. Therefore, tuning guides of Spark recommend using the same number of physical cores as the number of virtual cores.

However, if Spark loads Parquet files in parallel, memory overheads increase. In the software stacks to load a Parquet file in Spark, there are several buffering mechanisms for the file: a Java class of a Parquet reader buffers to construct records, an HDFS client buffers to calculate a checksum for each block, and the OS buffers as page cache for future accesses. These memory overheads increase depending on the task concurrency of Spark and pollute the page cache as a result.

The above three observations indicate that cache pollution frequently happens when Spark reads Parquet files with unnecessary data loading, mostly meaningless page caching, and multi-threaded and multi-layered buffering. Spark and Parquet libraries reuse the existing mechanisms to load files, such as HDFS and OSs, which do not work efficiently to load large Parquet files.

B. Page Cache Effects on Spark SQL

As discussed in Section II-A, reading large Parquet files pollute the page cache in the underlying OS. The problem is that the cache pollution affects not only Parquet file reads but also other file operations in the entire system, such as Spark shuffle processing, which uses local files as temporal storage and relies on the page cache for its performance.

We analyze the performance effects of page cache on Spark by changing its heap memory configurations and comparing

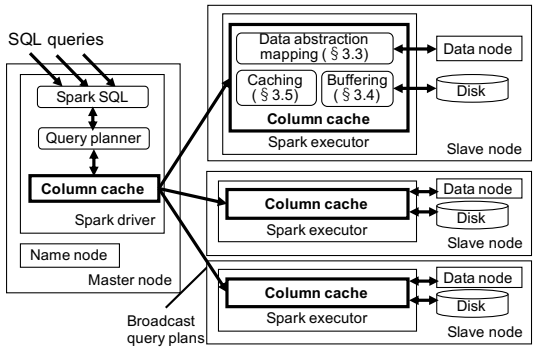

Fig. 5. **Overview of column cache.**

the performance and resource usage. We run ten TPC-DS queries with a 24, 48, and 80 GB JVM heap for Spark on the same cluster as the experiment in Section II-A. Spark can use raw heap memory outside of the JVM heap for shuffle processing, and we set 48 GB of the raw memory in all configurations. Using raw memory, a large portion of the JVM heap is used for buffering HDFS and Parquet data.

Fig. 4(a) shows how much these configurations affect the overall performance of our experiment. Fig. 4(b) reports the total bytes of disk reads in the six nodes. We found that disk intensive queries degraded performance by up to 14 minutes when changing the 48 GB heap to 80 GB. In the 80 GB heap, we often observed larger disk reads due to memory pressure in page cache. In contrast, the 24 GB heap caused runtime garbage collections due to memory pressure in JVM heap.

For ease of understanding, we also show the time-series trace of user memory and OS page cache in Fig. 4(c) and (d). Larger heap configuration caused higher memory pressure that evicted the page cache. In Spark, a symptom of shuffle processing appears as a spiky increase of user memory consumption (e.g., at 16 minutes in Fig. 4(c)). Fig. 4(d) shows the page cache decreases after the shuffle processing starts. The results of our experiments indicate that reduced memory footprint in Parquet files should lead to significant speedups in disk-intensive queries due to lower pressure in the page cache.

III. Column Cache

Column cache is a new buffer cache that overcomes the drawbacks of nested data abstractions in columnar storage on distributed file systems. We utilize it to solve the three problems described in Section II. Specifically, Parquet readers can reduce redundant buffers with data abstraction mapping (Section III-C), column cache simplifies and optimizes block I/O management (Section III-D), and cache pollution decreases as a result of our cache management (Section III-E). In this section, we focus on Parquet files as a primary target of column cache and Spark as the target of a parallel distributed computing framework.

A. Architecture

As shown in Fig. 5, column cache runs as a software component in user processes for a master and slaves. In the slaves, Spark executors call column cache to read Parquet files on HDFS instead of the default components. Column cache recognizes the formats of HDFS and Parquet files to read them without speculative disk reads in the OS. We create a buffering and caching mechanism for column cache without using the existing page cache of OSs. Column cache retrieves query plans in the master and uses the information in slaves as described in Section III-E. In data parallel computing, slave processes often scan different data splits that are not shared, and so, column cache does not share cached data among processes.

Column cache works only to read Parquet files and does not modify the others. As a result, Spark can use OS page cache to perform shuffle processing and read files of different formats as usual. In addition, column cache does not require any modification in HDFS *servers* such as Name and Data nodes. Therefore, column cache is available without any restarts of such storage servers. This design choice is essential since users hesitate to modify and restart existing storage servers to optimize workloads on running data clusters.

B. Overview of Buffer Cache Policy

Once column cache uses a memory buffer to read a file, it keeps it as a disk cache for requests to read the same file in the future. Unlike page cache in OS, column cache associates a range key between the beginning and the end of the offsets of cached file content to a memory buffer. Thus, a key has three elements: a file path, an offset in the file, and a length of a buffer. Every access to the offset within a cached range key returns cached data in column cache. If a Spark executor requests a disk region that overlaps with a cached memory, column cache issues disk reads only for the non-overlapped region and reuses the overlapped region in the cache.

Column cache manages disk cache with different strategies for two types of access patterns in Parquet files as described in Section II-A: for metadata regions and for the others. Column cache never evicts disk cache for metadata regions although it does evict cache for the others with a cache eviction policy described in Section III-E. A Spark executor accesses a metadata region of a Parquet file multiple times if the Parquet file is large. On the other hand, a metadata region is small because it contains only offsets for column data, data types, the length of metadata, and magic bytes. Therefore, we believe that memory of disk cache for metadata regions will not cause system memory pressure.

Column cache does not cache files for an HDFS checksum, which corresponds to an HDFS file as shown in Fig. 1. Column cache reads an HDFS file and then stores memory buffers for the file in the disk cache. Although verification with its checksum is necessary to read an HDFS file, this verification is redundant to read cached memory buffers. An advantage of user-level caching of Parquet files in column cache is that it reduces such redundant verification.

Layer	API
Parquet	ParquetFileReader.readNextRowGroup
HDFS (data)	BlockReaderLocal.read
HDFS (csum)	BlockReaderLocal.fillBuffer
local FS	pread

Fig. 6. **Parquet/HDFS/Linux APIs to be tracked.**

C. Data Abstraction Mapping

Column cache allocates the exact size of memory for each column in a row group of a Parquet file. In general, the redundancy of disk buffering is derived such that memory must be allocated before disk reads even though disk reads are necessary to calculate the required size for the memory. In contrast, with the format information of a Parquet file, column cache can allocate the exact size of memory for each column before the actual disk read.

Offsets in a Parquet file are not offsets in an OS file. A Parquet file is an HDFS file that consists of multiple files. With only the offset ranges of a Parquet file, column cache cannot allocate memory before disk reads.

Column cache resolves a mapping from a column to local files by communicating with a Name and Data node of HDFS. In typical use-cases of Parquet, distributed computing frameworks calculate the offset ranges of a Parquet file to be scanned in advance of actual scans. Column cache acquires these offset ranges and sends them to an Data and Name node. They return the actual paths, offsets, and lengths of local files to be read, and then the column cache can allocate the exact size of memory for local file reads to read a column. Column cache keeps the data mapping in memory to avoid redundant server inquiries.

Fig. 6 shows example APIs that read local files of a Parquet file. When column cache calls the readNextRowGroup method of ParquetFileReader class to read records, the HDFS client module performs two reads of HDFS files, one for a block file (BlockReaderLocal.read) and one for a checksum file (BlockReaderLocal.fillBuffer). Thus, an HDFS client calls pread on the block and checksum file for an HDFS file read. Column cache allocates the exact size of memory for the two file reads beforehand.

D. Explicit Disk Buffering

Column cache uses direct I/O [11] to bypass page cache in an OS so as to avoid unnecessary disk readahead in the page cache. The biggest advantage of direct I/O is that it does not break file systems guarantees, such as file system journaling and access controls. Specifically, direct I/O does not merge or split block I/O requests from the user's applications. Therefore, we emulate a raw block I/O request within the file boundaries of each single read-variants system call invocation. Underlying block device drivers eventually digest the requests into a physical I/O that is as large as possible. Direct I/O requires programs to obey alignment rules in Linux and file descriptors with special open flags, but the limitations are moderate for implementing column cache. A widely used optimization in HDFS, short circuit local reads [17], allows us to retrieve the local file descriptors and offsets for HDFS block and checksum files. Note that column cache retrieves remote blocks as current HDFS does, but it also puts data in the cache as well as local data.

Column cache also leverages the information of file formats to aggregate multiple read system calls into one. Specifically, it aggregates physically consecutive columns to be scanned. The simple block I/O in direct I/O allows us to increase physical disk throughputs simply by aggregating the system calls.

Initial disk reads to obtain the structure of a Parquet file require a different buffering strategy, since the information of file formats is not available yet. At the initial read, column cache reads a disk block size (e.g., 4 KB) of data from a Parquet file to retrieve the offset and length of the metadata region. After the initial read, we perform a single read system call with the length and the offset of the metadata region.

One thing to note here is that we do not provide asynchronous and prefetch mechanisms for file reads, unlike page cache in the OS. The OS may aggregate (or split) multiple block I/O in different threads, but this is not necessary for typical use cases of Parquet files. We assume that column cache callers are data-parallel executors such as Spark, which attempts to read separated data regions in parallel. Therefore, asynchronous and prefetch mechanisms potentially cause redundant thread creations and latencies for waiting for irrelevant block I/O.

E. Cache Management

Column cache reduces cache pollution and keeps *system* memory less pressured by tracking states of each buffer cache entry. We use reference count, *weak* reference count, and last accessed time for tracking these states. Unlike existing work for the least-recently-used, least-frequently-used, and other variants (e.g., [18], [19]), our cache management is based on incoming data accesses within query plans.

The initial state is *Loading*, in which column cache fetches data from local or remote storage. Column cache determines the Loading state if the reference count is larger than zero. The reference count is incremented before the actual disk read starts and decremented after it finishes. Column cache does not release the memory at the Loading state.

The second state is *Reserved*, in which column cache keeps a column in memory until incoming queries consume it. We use weak reference counts for each column to manage the Reserved state. The weak reference count is calculated with query plans, which contain information about which order of tables is scanned in a query. Column cache prioritizes which columns should be kept in memory with the frequency of remaining scans each column has. Column cache first releases the least-frequently-used cache at the Reserved state if the system memory is pressured. We also use the least-recently-used policy with the column's last accessed time if two have the same priority.

The third state is *Cached*, in which column cache keeps column data in memory for potential iterative reuse during

API (parameters)	Return	Caller (Spark)
extractPlan(Root*)	Map*	Driver
broadcastPlan(Map*)	Void	Driver
receivePlan(Map*)	Void	Executor
parsePlan(Parquet Path)	Void	Task thread

Fig. 7. **APIs for ingesting query plans.** Map* represents the mapping of a Parquet file path and Parquet filters. Root* represents the root node of a query plan that is made as a direct acyclic graph. This figure also lists API callers when Spark uses column cache.

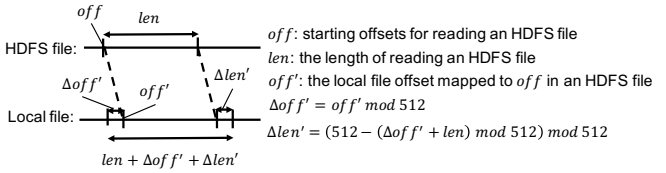

Fig. 8. **Converting an HDFS file read into a local file read.** The upper line shows a particular region that column cache requests to read from the view of HDFS. In this example, we handle 512-byte alignments in local file reads (the lower line) to use direct I/O.

a data analytics session. The Reserved cache becomes the Cached state if the weak reference count reaches zero, which mean that all the running queries do not scan the cache. Column cache monitors amounts of *free* system memory so that cache in the Reserved or Free state does not evict page cache in the OS. In our prototype, we set 80% of system memory usage as a threshold to start shrinking the column cache. Column cache also provides the upper limit of the Cached memory for use in specific environments such as those in which cluster schedulers exist (e.g., YARN).

IV. Implementation

We prototyped column cache in Parquet 1.9.0 on Hadoop and HDFS 2.7.3 on OpenJDK 1.8. We selected Java and Scala for our prototype development because Parquet, Hadoop, and Spark are written in these languages. We implemented column cache with 3920 lines of Java and 2033 lines of Scala. We modified 88 lines of Spark 2.2.0 to call column cache.

A. Column Cache APIs

Column cache provides APIs for Parquet file reads and for ingesting query plans. These APIs wrap existing APIs. Thus, existing users of Parquet files can use column cache simply by changing API calls to the column cache wrappers. We also define additional APIs in which data analytics frameworks can pass their query plans to column cache.

Fig. 7 shows four APIs that column cache uses to ingest query plans. A query planner in parallel distributed computing frameworks first calls extractPlan with the root of the tree that represents the query plan. Then, extractPlan extracts pairs of scanned HDFS files and filter operators for a Parquet read. The broadcastPlan and receivePlan APIs broadcast and receive the extracted map with (de)serialization. Scanner/filter task threads call parsePlan to request column cache to count the number of incoming scans of each region in the given HDFS file.

B. Explicit Memory Management

The cache management discussed in Section III-E requires explicit, raw (i.e., C language level) memory management because we use Java/Scala for the development of column cache. JVM does not return JVM heaps to the OS unless GCs occur. To this end, column cache directly calls anonymous mmap/munmap to allocate and free memory for buffer and cache. Column cache uses JNI to directly call mmap/munmap with an external library [20]. The library wraps every Libc call and contains native binaries for each CPU platform. We do not use other malloc variants such as a sun.misc.Unsafe memory allocator because column cache needs to allocate far larger memory than a single memory page. This kind of raw heap management in Java is now becoming common among data analytics such as Spark [21].

C. Direct I/O on HDFS files

An obstacle in using direct I/O in Linux is the rules for alignments [11] in the read system call parameters. Specifically, we need to align the memory address, file offset, and length to be read. In particular, the alignments for offset and length raise an implementation challenge because we need to convert offsets and lengths in an HDFS file to ones in local files. Fortunately, mmap returns a page-aligned address of the memory, which often align to the size of disk blocks (e.g., 512 bytes). Thus, we need to care about the alignments for file offset and length to be read.

Fig. 8 shows the conversion of an HDFS file read to a local file read with direct I/O. When column cache receives a request for reading a column, it first looks up the information of the Parquet metadata to retrieve the HDFS file path ($path$), the starting offset (off) of the column, and the length (len) of the column. Column cache also obtains local file mapping (a file descriptor for a local file path (FD), offset (off'), and length (len')) from the data abstraction mapping with the HDFS-level information. Then, we allocate raw heap memory ($addr$) with the size of $len + \Delta off' + \Delta len'$ in Fig. 8. Column cache calls the pread systemcall on the allocated memory with FD and registers $addr + \Delta off'$ as a cache entry with the key ($path$, off, len).

V. Evaluation

In this section, we report the performance of column cache that is integrated into Spark 2.2.0. Three types of workloads are evaluated: a microbenchmark, single-stream TPC-DS, and multi-stream TPC-DS. TPC-DS is a standard benchmark for decision support systems and covers four query classes that characterize most decision support queries: reporting, ad-hoc, iterative, and data mining. We use a scale factor of 1,000 GB for the TPC-DS input.

A. Experimental Setup

We use our computing cluster under IBM Cloud for these experiments. TPC-DS workloads are run under one master and five slave nodes, while our microbenchmark uses a single

node. In IBM Cloud, each node runs Ubuntu 16.04 LTS with 32 virtual CPU cores, 128 GB RAM, and two virtual disks. There are two types of disks: a 100-GB disk for general binaries and logs, and a 1-TB disk for HDFS and Spark. They are formatted and mounted as an XFS file system on Linux 4.13. The block I/O scheduler is Noop, which is the default scheduler in our environment, and the readahead size is 128 KB. Before every measurement, we drop page cache in Linux to ensure the input is read from the file system. The configuration of Spark for our experiment is a 48-GB JVM heap, 48 GB of raw memory, and 32 GB unused. The unused memory is reserved for page cache. The microbenchmark uses a 48-GB JVM heap as well. All the experiments use the same versions of OpenJDK (version 1.8), HDFS (version 2.7.3), and Spark (version 2.2). Our Spark runs as the standalone mode, where each node allocates a JVM process for Spark jobs. We set ten threads for the GCs in the Spark process. All the configurations are the same in the vanilla Spark and our customized Spark with column cache. For multi-stream runs of TPC-DS, we configure Spark as a fair scheduling mode, which enables us to run concurrent queries [22]. Other configurations are not mentioned here as they are the default.

B. Microbenchmark

We created a microbenchmark that scans Parquet files on a single node so as to easily understand the performance advantage of column cache. It emulates a typical scanning phase of Spark SQL, which assigns a series of Parquet files to threads. We used 256 Parquet files with SNAPPY compression. Each file contains three million records with flat (i.e., not-nested) columns that have eight different data types. The benchmark assigns the files to 32 tasks corresponding to the 32 CPU cores in our experiment.

Fig. 9(a) shows the throughput when the column cache is used and not used. We also changed the number of threads to verify scalability and see how the effectiveness of column cache changes by the number of concurrent scans. The results are averages of ten runs for each configuration. Column cache shows up to 1.2x speedup compared to the baseline. The throughput well scales up to 16 threads, and the effectiveness of column cache increases as the number of threads increases. This means column cache is well suited for big data analysis. This benchmark is an extremely disk-intensive workload, and thus, the throughput with 32 threads was worse than the one with 16 threads because the workload reaches the maximum bandwidth of the storage.

Fig. 9 also contains time-series trace of CPU, memory, and disk usages with or without column cache under the configuration of 32 threads. We focus on the results of 32 threads since other experiments with Spark also use 32 threads for data scans. Fig. 9(b) shows higher CPU usage in column cache, which means efficient processing of Parquet files. The CPU utilization becomes high while decoding complex formats of Parquet files and becomes low during disk read. Column cache achieves higher CPU usages because of efficient utilization of memory and disks.

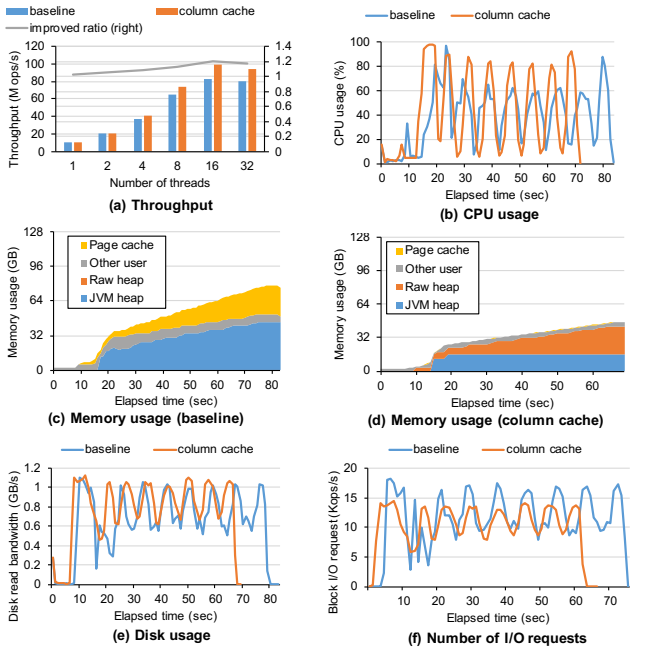

Fig. 9. **Microbenchmark results.** These figures show the throughput and resource usage of our microbenchmark that emulates a typical scanning phase of Spark SQL.

Fig. 9(c) and (d) shows memory usage in the baseline and column cache, respectively. We separated the usage of the Java heap and raw heap. The baseline stores in-memory data to the Java heap, while column cache uses raw heap with explicit memory management. "Other user" represents the memory usage of user processes other than the Java and the raw heap, calculated by subtracting the two areas from the usage of total user memory.

The memory usage of column cache is much smaller because the size of page cache is very small. This is because column cache bypasses page cache in OS using direct I/O. The sum of the Java and the raw heap is almost the same between the baseline and column cache because a large portion of these areas is used by the in-memory data. Note that the size of the page cache of the baseline and the size of the raw heap of column cache are almost the same. Since the size of the raw heap is the size of the in-memory data, this result implies that the in-memory data is redundantly stored in both the user memory buffers in the Java heap and the page cache in the case of the baseline.

We also measured the number of block I/O requests to show the advantage of direct I/O. Fig. 9(e) shows the bandwidth for disk reads. The column cache achieves higher bandwidth by explicit disk buffering with direct I/O. Fig. 9(f) shows the number of block I/O requests per second, which decreases in column cache compared to the baseline because direct I/O can issue larger chunks of block I/O than page cache, while page cache limits the block size to the readahead bytes. The default size of readahead in Ubuntu is 128 KB, but we observed that 94% of requests from column cache is 1 MB in our

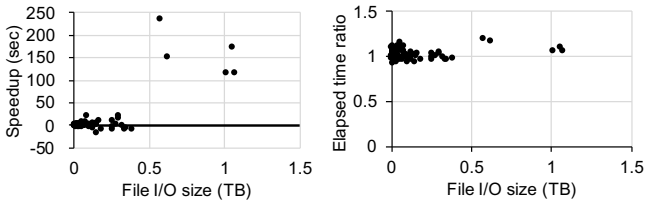

Fig. 10. **Elapsed time.** Each point shows the improvement of processing time for each query. The left figure shows the absolute decrease of elapsed time by column cache. The right figure shows the improvement ratio of the time (higher is better). Both figures show the result with file I/O that Spark reports.

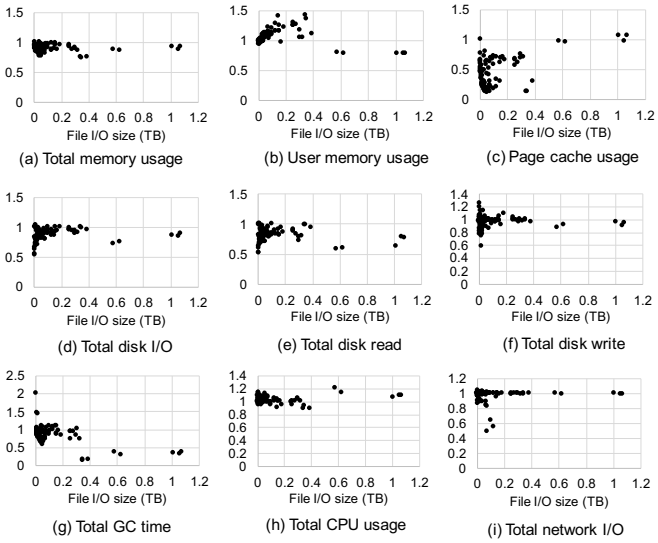

Fig. 11. **Ratio of resource usages.** Each point shows the decreased ratio of resource usage for a query. Lower y-axis values are better and less than one means improvement over the baseline. The x-axis shows the amounts of file I/O that Spark reports.

microbenchmark. The page cache does not allow adaptive disk reads even if there are various sizes of disk reads.

These results explain why we do not choose mmap-based I/O for column cache. Mmap reuses the page cache functionality including readahead. Generalized mechanisms of readahead in the page cache cause the slowdown in our workloads.

C. Single-Stream TPC-DS

We also evaluate column cache with all the 103 queries in TPC-DS. We measured the elapsed time for each query and the resource utilization in each workload. Each result is an average of ten runs.

Fig. 10 shows the speedups for the 103 queries of TPC-DS with column cache. The file I/O sizes shown were the total number of input and shuffle bytes that Spark reports. Column cache showed at most a 1.2x speedup compared to the baseline although Spark workloads involve other operations such as data compressions and shuffle processing. In particular, large improvements tended to happen in queries with large file I/O. A large file I/O often occurs in shuffle-heavy processing in Spark. Column cache monitored system memory to avoid memory pressure in the page cache, which speeds up Spark shuffle processing.

In contrast, 33 queries with less than 400-GB file I/O showed a slowdown by up to 15 seconds (7% slowdown for the query) due to the overhead of column cache. Column cache does not optimize workloads with small working sets, since the system can store the whole data in memory.

Fig. 11 shows the overall resource utilization in the 103 queries. We measure memory usage, disk usage, GC time, CPU usages, and network usages. We measure the GC time that Spark reported while we monitor system-level statistics for other resources.

Fig. 11(a) shows reduced memory usage of column cache (by up to 30%) while (b) shows up to a 40% increase of memory usage in small file I/O. Column cache cached data in user memory, and so, it simply used the unused memory for its cache. In large shuffle size, the cache was released and the total size of memory became less than the baseline. As shown in Fig. 11(c), column cache evicted its cache when the page cache dominated the memory and increased the ratio of page cache by 6%. The figure also shows that the direct I/O reduced the size of page cache in small file I/O.

Fig. 11(d) shows that column cache reduced disk reads by up to 48%. Column cache reduced cache misses in the page cache, which in turn reduces extra disk reads, as shown in Fig. 11(e). Some short queries also reduced disk writes with column cache, as shown in (f). Column cache reduced memory pressure in page cache and it also resulted in fewer disk writes.

Queries with large file I/O showed that GC time decreased with column cache, as shown in Fig. 11(g). We observed up to a 90% decrease of GC time but some short queries show a 100% GC time increase due to the overhead of column cache. Column cache shows up to a 20% increase of CPU time in Fig. 11(h).

Network I/O is also decreased in queries with small file I/O (as shown in Fig. 11(i)), since the queries sent only a few Giga-bytes to networks on each node. As a result, the cache for remote blocks in column cache reduced relatively large amounts of network I/O in these queries. Other queries did not show any improvements in amounts of network I/O because most of the queries sent and received network packets for shuffle processing, which we did not modify.

D. Analysis of a Representative Query

To provide more detailed views of resource utilization, we show the time-series metrics of TPC-DS query q14b, with the highlighted result of column cache speedup. Fig. 12 shows the time-series data of CPU utilization, disk read bandwidth, used memory, and page cache size with and without column cache for a single run of query q14b. We select the metrics in one of five slave nodes in our cluster.

As shown in Fig. 12(a), the user memory of query q14b can be divided into two phases: eventual memory increase by

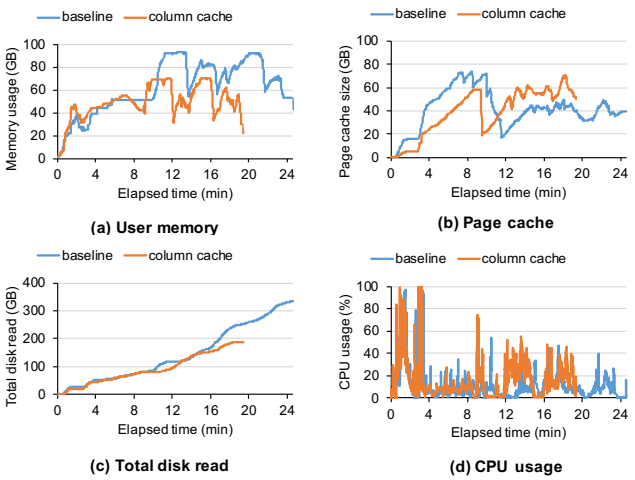

Fig. 12. **Summary of TPC-DS query q14b.** The figures show time-series traces of each resource utilization in a slave node.

table scan (until 10 minutes) and spiky increases by shuffle processing. Column cache directly affected the table scan phase, but it also influenced shuffle processing by our cache replacement policy. Column cache decreased total amounts of user memory in shuffle phase by evicting less frequently used data during the query processing.

In addition to the user memory reduction in the scan phase, column cache did not pollute the page cache by using direct I/O, as shown in Fig. 12(b). The OS preserved the page cache until the system memory was filled with it (at around 7 minutes). The page cache in column cache was derived from shuffle outputs.

Fig. 12(c) shows that column cache reduced disk reads by 142 GB. We decreased total disk reads by avoiding cache pollution in the page cache. Column cache increased CPU utilization, as shown in Fig. 12(d). Column cache achieves higher CPU usages due to its efficient utilization of memory and disks, as we discussed in Section V-B.

E. Multi-Stream TPC-DS

The third evaluation of column cache is under the existence of multiple queries, which represents a more realistic use case. To this end, we built 90 workloads with seven randomly selected queries from 103 queries in TPC-DS. These workloads concurrently submitted the selected queries to Spark SQL. Spark also processed the queries in parallel by using a fair scheduler [22]. We evaluate the elapsed time for each query and the overall resource utilization in each workload. Each result is an average of three runs.

As Fig. 13 shows, large improvements tended to happen in queries with large file I/O, as well as for results of a single query. The speed up by column cache became larger (up to 1.28x) than the single query run shown in Fig. 10.

Fig. 14 helps clarify how the 1.28x speedup happened. The figure shows the overall resource utilization for each workload. Multi-stream TPC-DS caused heavier shuffle processing and

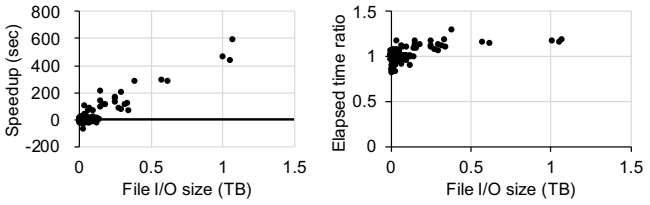

Fig. 13. **Elapsed time.** The figure shows the processing time of each query in multi-stream TPC-DS. Note that we use file I/O sizes that each query (not workload) requires, and so, the maximum amounts of file I/O are different from those in Fig. 14.

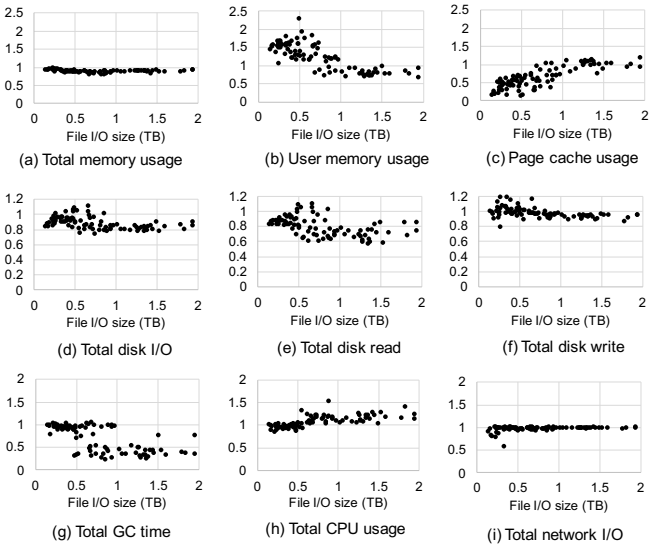

Fig. 14. **Ratio of resource usages.** Each point shows decreased resource utilization in a workload with seven concurrent queries. We use file I/O sizes that Spark reports for each workload, which consists of seven concurrent queries.

more memory pressure than single-stream TPC-DS. As a result, user memory usage in Fig. 14(b) showed a more decreased ratio than the results of single-stream TPC-DS because of the cache eviction in column cache. As shown in Fig. 14(c), we observed up to an 18% increase of page cache in workloads with large file I/O. As well as user memory, column cache reduced total disk reads by up to 43% in Fig. 14(e). We also observed a large ratio of user memory in workloads with small file I/O (up to 2.3x). In multi-stream TPC-DS, column cache needed to preserve a larger cache for processing seven concurrent queries. As a result, the user memory in multi-streams became larger than in single streams if there was no pressure on the system memory. The high concurrency should cause large overheads in user-level cache management, but we still observed up to a 76% decrease of GC time.

VI. Related Work

Application-level buffer cache management, which is a primary characteristic of column cache, has been explored mainly in database management systems. An article in 1981 [23]

stated that the motivation behind their OS bypass was poor performance in generic OS buffer management. Modern storage engines [14] and databases [15], [16] also provide OS-bypassing functionalities. We also show that Spark, which is a modern data analytics platform, also has a performance issue in buffer cache management. In particular, column cache solves issues deriving from layered software stacks around parallel distributed computing and distributed file systems.

A more drastic design of column cache is to re-design the entire storage stack to optimize device-level data management. The biggest benefit of the drastic design is to fully utilize modern high-performance storage such as NVMe SSD. It enables us to reduce the cost of context switches between the kernel and user processes [24], [25]. We could also modify the OS kernel to expose key kernel-level information to Spark so that it can modify buffer cache policies as done in [26], [27], [28]. Instead of these drastic approaches, we demonstrate the design of application-specific buffer cache using the limited interfaces of the existing APIs of the Linux kernel. Fadvice [29] is also able to change cache behavior in OSs, but unfortunately, the functionality is limited and not enough to satisfy the requirements of column cache.

VII. Conclusion

In this paper, we presented and solved issues of nested data abstraction with column cache. In column cache, the unified architecture in user space enables us to easily leverage query plans and format information for optimized buffer cache in HDFS. Column cache increased Linux page cache size by 18%, reduced total disk reads by 43%, and reduced GC time by 76% in multi-stream TPC-DS. As a result of improved resource utilization, Spark showed a maximum speedup of 1.28x. We achieved the speedup with small fixes of Spark and no restarts of HDFS servers.

References

[1] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: A column-oriented dbms," in *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*, 2005, pp. 553–564.

[2] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: Interactive analysis of web-scale datasets," *Proceedings of VLDB Endowment*, vol. 3, no. 1-2, pp. 330–339, September 2010.

[3] "Apache parquet," http://parquet.apache.org.

[4] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu, "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 (ICDE '11)*, 2011, pp. 1199–1208.

[5] "Apache orc," https://orc.apache.org.

[6] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST '10)*, 2010, pp. 1–10.

[7] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI '15)*, 2015, pp. 293–307.

[8] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingan, D. Murray, S. Hand, and M. Isard, "Broom: Sweeping out garbage collection from big data systems," in *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (HotOS '15)*, 2015, pp. 2–2.

[9] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, 2012, pp. 15–28.

[10] "Apache tez," https://tez.apache.org.

[11] "Clarifying direct io's semantics," https://ext4.wiki.kernel.org/index.php/Clarifying_Direct_IO%27s_Semantics.

[12] M. Poess, T. Rabl, and H.-A. Jacobsen, "Analysis of tpc-ds: The first standard benchmark for sql-based big data systems," in *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*, 2017, pp. 573–585.

[13] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis of hdfs under hbase: A facebook messages case study," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '14)*, 2014, pp. 199–212.

[14] "Wiredtiger: System buffer cache," http://source.wiredtiger.com/2.9.3/tune_system_buffer_cache.html.

[15] "Direct i/o - facebook/rocksdb wiki - github," https://github.com/facebook/rocksdb/wiki/Direct-IO.

[16] "The column-store pioneer — monetdb," https://www.monetdb.org.

[17] "Apache hadoop 2.7.3 - hdfs short-circuit local reads," https://hadoop.apache.org/docs/r2.7.3/hadoop-project-dist/hadoop-hdfs/ShortCircuitLocalReads.html.

[18] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "Pacman: Coordinated memory caching for parallel jobs," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*, 2012.

[19] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies," in *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '99)*, 1999, pp. 134–143.

[20] "Java native accesses," https://github.com/java-native-access/jna.

[21] R. Xin and J. Rosen, "Project tungsten: Bringing apache spark closer to bare metal," https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html, April 2015.

[22] "Job scheduling — spark 2.3.0 documentation," https://spark.apache.org/docs/2.3.0/job-scheduling.html#fair-scheduler-pools.

[23] M. Stonebraker, "Operating system support for database management," *Communications of the ACM*, vol. 24, no. 7, pp. 412–418, Jul. 1981.

[24] S. Peter, J. Li, D. Woos, I. Zhang, D. R. K. Ports, T. Anderson, A. Krishnamurthy, and M. Zbikowski, "Towards high-performance application-level storage management," in *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage '14)*, 2014.

[25] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 4, pp. 11:1–11:30, Nov. 2015.

[26] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. A. Nugent, and F. I. Popovici, "Transforming policies into mechanisms with infokernel," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, 2003, pp. 90–105.

[27] S. VanDeBogart, C. Frost, and E. Kohler, "Reducing seek overhead with application-directed prefetching," in *Proceedings of the 2009 Conference on USENIX Annual Technical Conference (USENIXATC '09)*, 2009.

[28] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling," *ACM Transactions on Computer Systems (TOCS)*, vol. 14, no. 4, pp. 311–343, Nov. 1996.

[29] D. Plonka, A. Gupta, and D. Carder, "Application buffer-cache management for performance: Running the world's largest mrtg," in *Proceedings of the 21st Large Installation System Administration Conference (LISA '07)*, 2007, pp. 63–78.