In-Memory Indexed Caching for Distributed Data Processing

Alexandru Uta
LIACS, Leiden University
a.uta@liacs.leidenuniv.nl

Bogdan Ghit
Databricks
bogdan.ghit@databricks.com

Ankur Dave
UC Berkeley
ankurd@eecs.berkeley.edu

Jan Rellermeyer
TU Delft
j.s.rellermeyer@tudelft.nl

Peter Boncz
CWI
p.boncz@cwi.nl

***Abstract*—Powerful abstractions such as dataframes are only as efficient as their underlying runtime system. The de-facto distributed data processing framework, Apache Spark, is poorly suited for the modern cloud-based data-science workloads due to its outdated assumptions: static datasets analyzed using coarse-grained transformations. In this paper, we introduce the `Indexed DataFrame`, an in-memory cache that supports a dataframe abstraction which incorporates indexing capabilities to support fast lookup and join operations. Moreover, it supports appends with multi-version concurrency control. We implement the Indexed DataFrame as a lightweight, standalone library which can be integrated with minimum effort in existing Spark programs. We analyze the performance of the Indexed DataFrame in cluster and cloud deployments with real-world datasets and benchmarks using both Apache Spark and Databricks Runtime. In our evaluation, we show that the Indexed DataFrame significantly speeds-up query execution when compared to a non-indexed dataframe, incurring modest memory overhead.**

I. Introduction

The advent of data science has fundamentally changed our perception of how to gain insights, namely by adding what Jim Gray called a *fourth paradigm* of science driven by data [1]. However, while pioneers like Gray envisioned relational database systems to become the engines of this new branch of science [2], the tremendous momentum of data science has called for new systems to be developed, most importantly systems that are optimized for processing large amounts of unstructured or semi-structured data using clusters of machines. Due to the more agile and iterative nature of data science, those systems have departed from the idea of forcing data into a fixed schema for the purpose of giving the database system the chance to optimize common operations through query optimization and the use of indexes.

Dataframes [3], [4], for instance, are modern-day data science abstractions similar to relational tables that enable users to express computations through SQL-like interfaces [5] and execute those computations on distributed processing frameworks such as Dask [6] or Apache Spark [7]. The high-level interfaces enabled by dataframes and SQL are very attractive to users as their programs can automatically trigger query optimization [5] without manual tuning.

Even though dataframes are widely adopted by data scientists, they are only as efficient as their underlying runtime system can be. In practice, the performance can be underwhelming because systems like Spark have been designed around assumptions like the static nature of data and justifies the reliance on coarse-grained transformations as the main processing paradigm but are, unfortunately, increasingly obsolete [8]. In the last decade new use cases for data science workloads emerged in which data can be processed through streaming interfaces [9], [10] and data-lakes [11], [12], which makes existing data processing pipelines to not necessarily run on static read-only files. The net result is an inefficient setup which is bottlenecked by network and IO bandwidth due to the reliance on shuffle and broadcast operations [13].

Traditionally, data indexing has been a very effective way of minimizing the network overhead as it can significantly reduce the amount of data transferred by pre-filtering [14], [15], [16], [17], [18]. However, supporting indexes on non-static datasets is difficult since write operations may cause consistency issues when scheduling tasks. This is particularly the case when using an external index as it as been suggested by prior work [19].

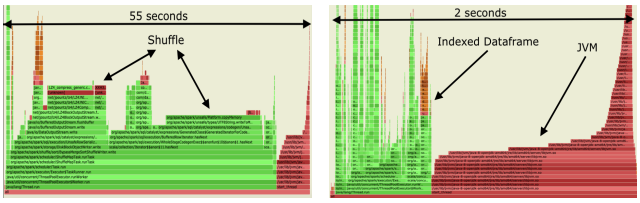
In this paper, we present a novel approach that combines the best of the two worlds of relational database technology and data science systems. We show how embedding a write-enabled in-memory indexed cache into structures like Dataframes unlocks better performance for modern applications while seamlessly integrating into processing frameworks like Spark. The result of this effort, the Indexed DataFrame, enables low-latency joins and point look-ups in interactive workloads on data that is continuously changing and increasing in size. The Indexed DataFrame extends the space of use cases for Spark by efficiently supporting applications such as on-line threat detection and response [20], or real-time social network monitoring and dashboarding [21].

We show empirically that the Indexed DataFrame can be seamlessly integrated in Apache Spark frameworks deployed in clusters, but also in production-ready cloud-based environments such as the Databricks Runtime [22]. The main benefit of the integration with existing frameworks is the automatic access to the framework's scheduling and fault-tolerance mechanisms.

The contributions of this work are:

1) We motivate the need of the Indexed DataFrame by showing the inefficiencies encountered in Spark when running typical data science queries (Section II).
2) We present the design and implementation of the novel Indexed DataFrame, including the API we support to index Spark dataframes, the integration with Spark's Catalyst optimizer, and the underlying indexing data structure (Section III).
3) We evaluate the performance of our reference implementation of the Indexed DataFrame which is available open-source[1]. We demonstrate the scalability of the Indexed

[1]https://github.com/alexandru-uta/IndexedDF



(a) Vanilla Spark. (b) Indexed DataFrame.

Fig. 1. Flame-graphs for 5 consecutive runs of a *join* operation on a Spark worker: Vanilla Spark (left) and the `Indexed DataFrame` (right) running on the Databricks Runtime. The workload is 5 consecutive join operations. The red part at the right of each subfigure represents JVM load, while the green parts represent actual Spark computations.

DataFrame, as well as the performance improvement of the Indexed DataFrame of up to 20X in production environments using several real-world workloads and datasets (Section IV).

II. A Case for Indexed DataFrames

Indexing is a well-known technique employed in traditional database systems to speed-up the access to the data source. In this section, we identify the main reasons why indexing is attractive for modern data science workloads and we explain why adopting them in data analytics engines such as Spark is challenging. In social-network graph processing [21] and cyber-security threat detection [20] many similar operations are performed continuously, while the datasets are constantly growing, often through fine-grained (individual records or small sets of) appends. Data processing frameworks such as Spark are not designed for offering sufficient performance to support such application patterns.

These applications make extensive use of *point lookups* and *join* operations. The former relate to quickly finding certain elements in a large data collection. The latter relate to combining information in multiple columns of two or more dataframes. Furthermore, both aforementioned applications could benefit from fine-grained *appends*: in the graph processing use-case, new links in social network graphs are formed continuously, while in real-time threat detection and security monitoring, network connections are incoming in high-volumes, and need to be analyzed in interactive time.

Hybrid transaction/analytical processing (HTAP) systems such as Druid [23], Splice Machine [24], or SnappyData [19] try to solve such problems by reconciling both OLAP and OLTP workloads. Designing such capabilities require a re-thinking of the underlying runtime system such that it efficiently supports both types of workloads, or augmenting it with another system, leading to maintenance difficulties.

Being designed with immutability in mind, Spark only supports coarse-grained data transformation and not fine-grained updates/appends. To support updates, Spark needs to be integrated with external storage such as Cassandra [25] or Delta Tables [11] or the Azure implementation of Data Lakes [12], [26]. For running queries on fresh data, *even if only few records have been added*, Spark requires reloading the complete dataset from the external data store after a write. Without supporting fine-grained appends in-place, reloading data from external data sources is an expensive operation, *which highly limits interactive response times*.

Listing 1. The Indexed Data Frame API.

```
// creating an index
var indexedDF = regularDF.createIndex(colNo)
// caching the indexed dataframe
var indexedDF = indexedDF.cache()
// key lookup returns a dataframe
val lookupKey = 1234
val resultDataFrame = indexedDF.getRows(lookupKey)
// appending all the rows of a regular dataframe
val newIndexedDF = indexedDF.appendRows(aRegularDF)
// index-powered, efficient join
val result = indexedDF.join(regularDF,
     indexedDF.col("c1") === regularDF.col("c2"))
```

Spark is generally inefficient for point lookups and joins. Without additional data structures or partitioning, point lookups in Spark are linear in time to the number of entries. Joins are even more complex due to Spark's distributed nature: data is either sorted and then merged (i.e., Sort-Merge Join [27]), or hash-tables are being built for one of the dataframes, these are then broadcast and probed locally against all entries of the other dataframe (i.e., BroadcastHash Join [28]). In data science operational pipelines, these operations are not run only once, but continuously. Therefore, performing $O(n)$ operations for lookups, building hash-tables and shuffling data around for every run is inefficient. Implementing an *index* next to the data helps both these operations: point lookups now become worst-case logarithmic time, while for joins the index acts as a pre-built hash-table.

To show evidence for this, we performed 5 joins operations in a sequence on a $7GB$ Broconn [20] table, joining it with a small random sampled subset of itself, of less than $10MB$. We ran these join operations on the Databricks Runtime on four `i3.xlarge` virtual machine instances. Figure 1 shows the performance breakdown for these two executions. The regular Spark implementation for join operations needs to perform the same networked operations and hash-table building for each join execution. For the Indexed DataFrame, the index is computed only once, and its overhead can be amortized over many executions of the indexed operations.

We have argued for and gave empirical evidence that supports the addition of indexes in Spark. In the following sections, we provide the in-depth design and performance evaluation of the Indexed DataFrame.

III. Indexed DataFrame Design

We propose the `Indexed DataFrame`, a data abstraction that we can use to manipulate cached indexed datasets. We present the API and the main design and implementation details. The Indexed DataFrame is built as an extension library which integrates with the Dataframe API supported by Spark SQL [5], and can be added to existing Spark programs.

A. Programming Interface

To address the requirements of a wide spectrum of large-scale data science applications, we designed the In-

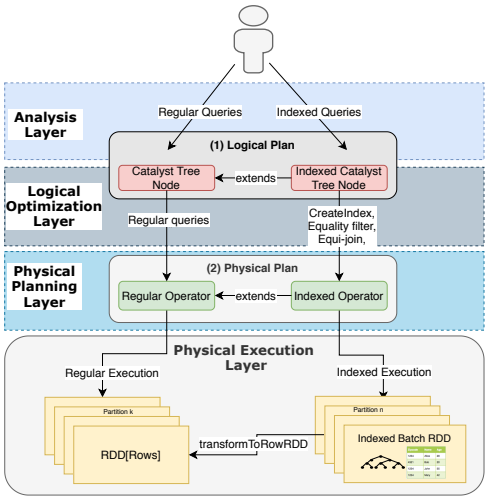


Fig. 2. Indexed DataFrame logical flow. Users write SQL queries or use the Dataframe API. Catalyst rules determine whether the queries are regular or indexed. If regular, they follow the regular execution. If indexed, special rules, and optimization strategies are applied such that indexed execution is triggered. An Indexed Batch RDD can always fall back to a regular Spark Row RDD to trigger regular execution on top of the Indexed DataFrame.

dexed DataFrame to support the following: `create index`, `cache index`, `point lookups`, `append rows`, and `indexed joins`. The corresponding Scala API is presented in Listing 1. In our current implementation the index supports any type of column, but for good performance, we recommend using only primitive column types (e.g., (un)signed 32/64-bit integers, floating point numbers).

As we want to store the Indexed DataFrame in-memory on the Spark executors, instantiating the index should be immediately followed by a caching operation. Furthermore, the *append rows* operation can be performed both in a fine-grained and a batch-oriented mode by organizing the rows we need to append as a regular Spark Dataframe. In this way, users can append with low latency small amounts of rows, or batch multiple updates in a larger Dataframe. When users need to lookup the rows associated with a certain key, our library returns a (smaller) Dataframe containing the required rows. In the case of *join* operations, if any of the sides of the relation are indexed, our implementation of the `Indexed DataFrame` triggers an indexed join operation. The result is a regular Spark dataframe. Evidently, in case of the *indexed join*, the indexed relation is always the build side (as it is actually pre-built due to the index), while the probe side is the non-indexed relation.

B. Integration with Catalyst

Figure 2 shows the architecture of the `Indexed DataFrame` and its integration with the Catalyst optimizer in Spark. To add indexed operations to the regular Spark SQL and the Dataframe API without modifying the Spark source code we employ Scala *implicit conversions*. In this way we can add our methods to the Dataframe class, while leveraging the full capabilities of the Catalyst [5] query optimizer. Our library includes *optimization rules* that make regular Spark SQL queries aware of our custom indexed operations.

In Spark SQL, queries have abstract representations called query plans. These are converted through a sequence of transformations into optimized plans that finally execute on the cluster. Catalyst translates queries into logical plans that provide high-level representations of each operator without defining how to perform the computation. Optimization rules transform the logical plan into a physical plan with specific instructions on how to execute the query.

Through our library, we use the extensibility of Catalyst to add index-aware optimization rules. These translate the indexed logical operators into physical operators. These rules ensure that the appropriate look-up functions are called for each indexed or basic logical operator and ensure that the Indexed DataFrame operations are always triggered when executing queries on indexed data. Similarly, for queries on non-indexed dataframes we fall back to the default Spark behavior.

C. The Indexed Batch RDD

Spark datasets are typically partitioned across multiple nodes so that the framework can divide jobs into multiple tasks that can be executed in parallel on multiple *executors*. The Dataframe API can perform relational operations on Spark's built-in distributed collections, i.e., the RDDs [7]. The `Indexed DataFrame` operates in a similar way by partitioning data across multiple executors, but requires a custom RDD implementation, the *Indexed Batch RDD*, to make use of indexed operations.

Figure 3 depicts the design of the *Indexed Batch RDD*. Our implementation stores data *in-memory*. This decision was made to optimize for performance but without loss of generality; the representation could easily extend to store data out-of-core, for example in SSD or NVMe devices for different tradeoffs. Each partition is composed of three data structures: (1) a *cTrie* [29], which represents the index, (2) a set of *row batches*[2], which store the tabular data, and (3) a set of *backward pointers*, which are used to crawl the partition for rows that are indexed on the same key.

Design. The cTrie is a concurrent hash-trie, which can employ thread-safe, lock-free inserts, deletes, and lookups. Furthermore, the cTrie can perform lock-free, atomic snapshotting in constant time. The cTrie snapshotting is similar to a *persistent data structure* [30]. Because a new snapshot version shares most of the state with the parent object, we only need to store the actual modifications resulted from appends. The cTrie requires minimum overhead when creating new Indexed DataFrames by taking snapshots on writes.

Non-unique Keys. The cTrie stores a pointer to the latest appended row associated with a key. If there are multiple rows associated with a key, the backward pointer data structure consists of linked lists, one per unique key. This data structure

[2]In our prototype we store data in row-wise format in the *Indexed Batch RDD*. However, this could seamlessly be changed to columnar formats. The decision is based on the type of workload the user needs to support.

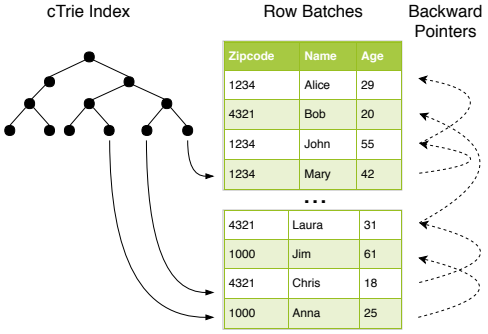


Fig. 3. Indexed DataFrame internal design: (1) cTrie for storing pointers to last row containing the key; (2) collection of row batches storing the data; (3) backward pointers for rows with equal keys.

can be used to traverse the list of rows associated with the key. The row batches are collections of binary, unsafe arrays (e.g., of 4MB in size), each storing a number of rows determined by the row and batch sizes. The pointers stored both in the cTrie and in the backward pointer data structure are packed in dense 64-bit integers, each containing the row batch number, an offset within a row batch, and the size of the previous row indexed on the same key.

Maximum Size. In our experiments we use indexed partitions with rows that may have up to 1 KB and 2^{31} row batches, each of which may have up to 4 MB. Thus, our setup enables 4×2^{31} MB data per core, sufficient for modern server configurations. Transformations within a partition are sequentially executed on a single core. As a rule-of-thumb, Spark deployments should be configured with 1 to 4 partitions per core[3]. Both the batch and row sizes are configurable.

Scheduling Physical Operators. To implement the *indexed* operations efficiently in Spark, we employ a *hash partitioning* scheme on the indexed key and shuffle operations to transfer the data to their indexed partitions. This section presents the main `Indexed DataFrame` operations.

Index Creation, Append. The operation of the Indexed DataFrame on a single partition is similar both for index creation and for appends. The Indexed DataFrame is *hash* partitioned on the indexed column. This ensures a better load balancing when the key ranges are not known a-priori. When an index is created on a regular Dataframe, its rows are shuffled based on the hash partitioning scheme to their respective Indexed DataFrame partitions. First, each row is inserted in the row batch responsible for its key. If a row with a similar key was already inserted in the partition, the cTrie entry for the key is updated to point to the newly added row, while the backward pointer of the newly added row is created to point to the previous row.

Lookup. A lookup operation is scheduled on the Spark partition responsible for holding that key based on the hash partitioning scheme. The lookup is then performed locally, and the set of resulting rows is returned to the user as a regular Dataframe. A local lookup consists of a search in the cTrie, followed by a traversal of the backward pointers if multiple rows share the same key.

[3]https://spark.apache.org/docs/latest/tuning.html

Listing 2. Indexed DataFrame divergence.

```
// dataframes A and B share the same parent
indexedDF_A = indexedDF.append(appendDF1)
indexedDF_B = indexedDF.append(appendDF2)
// divergent and materialized in reverse order
indexedDF_B.collect()
indexedDF_A.collect()
```

Indexed Join. To join a Indexed DataFrame and a (regular) Dataframe, the rows of the latter are shuffled according to the hash partitioning scheme of the former. As the *build* side is already created in the form of the index, the *probes* are made locally from the shuffled rows. If the Dataframe size is small enough to be broadcasted efficiently, we fall back to a broadcast-based join instead of a shuffle.

D. Fault-tolerance and Consistency

Spark fault-tolerance is achieved via task recomputation based on its lineage graph. To achieve fault-tolerance for the `Indexed DataFrame`, we use the recomputation mechanism provided by Spark in the following way. All Indexed DataFrame operations, except *append*, are able to be replayed using the lineage graph without any additional considerations. For the append operation to be re-generated properly, we rely, similarly to Spark Structured Streaming [10], on either a replayable data source, such as Apache Kafka [9] or a persistent (distributed) file system, such as HDFS [31].

Maintaining consistency when supporting data appends is non-trivial. We distinguish special use-cases: supporting straggler nodes, bad scheduler decisions, or even failed nodes. In either of these cases, a Spark task can be scheduled on a node that does not have locally the required indexed partition. Spark is optimized for achieving data locality, so every task is prioritized to run on a node that contains its input data. In case locality cannot be achieved within a configurable timeout, tasks can be scheduled on remote nodes. In such a case, in practice we end up with multiple copies of the same data. If this piece of data is obtained through multiple transformations, these are replayed on the new node.

In the case of Indexed DataFrame, this mechanism works similarly. This means that all former operations must be replayed locally, which is a safe operation but results in two *identical* copies of the same data. While in regular Spark operations this is a non-issue, since data is static, a new *append* operation means that the two partitions are not identical anymore. Hence, the older partition cannot be used for future tasks, because it is stale. To properly handle these situations, on each *append* on an Indexed DataFrame, the underlying custom RDD data structure (i.e., Indexed Batch RDD) increments a *version number*. The version number aids the scheduler not to send tasks to *stale* partitions, ensuring *consistent* operation.

TABLE I
THE HARDWARE CONFIGURATION USED IN OUR EXPERIMENTS.

Hardware	Type	#Cores	Memory	Network	Disk
Private Cluster	Intel E5-2630-v3	16	64 GB	FDR InfiniBand	SSD
Amazon EC2	i3.xlarge & i3.8xlarge	4 & 16	30 & 122 GB	10 Gbps	SSD

TABLE II
THE REAL-WORLD AND SYNTHETIC DATASETS AND THE QUERIES WE USE TO EVALUATE THE PERFORMANCE OF THE INDEXED DATAFRAME.

Dataset	Experiment	Query	Query Desc.	Index Column
SNB (SF-1000)	§IV-B,IV-C,IV-D	Join	join edges with vertices ON edge_source	integer
SNB (SF-300)	§IV-E	SQ1-SQ7	online[4]	various
US Flights (120 GB)	§IV-E	Q1	join flights with planes on tailNum	string
	§IV-E	Q2	select * where tailNum = x	string
	§IV-E	Q3	join flights with selected flights table (flightNum <200)	integer
	§IV-E	Q4	join flights with selected flights table (flightNum <400)	integer
	§IV-E	Q5	point query (10 matches)	integer
	§IV-E	Q6	point query (100 matches)	integer
	§IV-E	Q7	point query (1000 matches)	integer
TPC-DS (SF-1,10,100,1000)	§IV-E	Join	store_sales JOIN date_dim ON ss_sold_date_sk	integer

E. Divergence and Multi-Versioning

Because the *append* operation returns a new version of the Indexed DataFrame, we can reach the situation depicted in Listing 2. Here, two successive appends on a parent data frame create two divergent children dataframes. However, the append is only triggered when the newly created Indexed DataFrame is materialized. Theoretically, we could make the parent dataframe read-only once an append is performed, to guard against such situations. In practice, in case the children dataframes are materialized in reverse order, it is not trivial to decide which append should be permitted.

To permit both appends, a pragmatic solution would be to employ a *copy-on-write* mechanism, such that divergent dataframes could co-exist. However, this incurs large performance penalties (i.e., full data copies) and storage overheads (i.e., keeping multiple copies of the same data). An efficient solution is to employ a *persistent data structure* scheme, where the children dataframes share the parent data and only store the *deltas*. This is achievable due to the cTrie index capability: whenever a snapshot is triggered, the newly created copy shares the initial state with no memory overhead and only stores differences to the previous version. In case of the row batches, this is achieved with a similar scheme: we use a secondary cTrie that stores pointers to the row batches of the Indexed DataFrame partitions. In this way, the Indexed DataFrame supports divergent appends with *minimum storage and performance overheads*.

F. Implementation

The Indexed DataFrame was implemented in Scala and Java and is available as a standalone open-source *sbt* project. Users can bundle the generated jar binary in their applications and simply have Spark make use of the Catalyst optimizations and strategies that trigger indexed operations. After a call to *createIndex* is made (i.e., the only modification a programmer has to make to a Spark program to use the Indexed DataFrame), all the functionality we implemented is triggered automatically if the optimizer rules we provided determine that the index can be used. These include Spark SQL code and dataframe code.

The Indexed DataFrame allocates memory for the rows it stores in *unsafe* off-heap memory that is not managed by the JVM, thus not triggering inefficient GC pauses. The only JVM-managed memory of the Indexed DataFrame is the cTrie, which is a scala-native implementation. However, as we show in Section IV, this is sufficiently effective in maintaining an index that achieves interactive query time.

TABLE III
THE SIZE OF EACH PROBE AND BUILD RELATION IN DIFFERENT JOIN OPERATIONS USED IN OUR EXPERIMENTS.

Join Scale	Probe Side (rows)	Indexed or Build Side (rows)	Result Size (rows)
S	10K	1B	1.5M
M	100K	1B	14M
L	1M	1B	110M
XL	10M	1B	1B

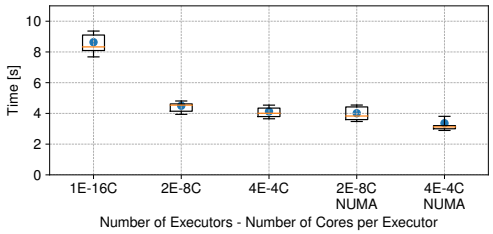


Fig. 4. Deployment of the `Indexed DataFrame` on a dual-socket NUMA machine. Number of Spark executors vs. cores per executor vs. pinning executors to NUMA domains. Data is presented as IQR boxplots, with whiskers representing minimum and maximum, while blue dot represents the mean performance.

IV. INDEXED DATAFRAME EVALUATION

We take an experimental approach to evaluate performance aspects of the Indexed DataFrame. To this end, we seek to answer the following questions:

Q1 *What (environment) parameters and variables is the Indexed DataFrame sensitive to?*

Q2 *How does the Indexed DataFrame scale with the number of cluster machines, numbers of cores, and problem size?*

Q3 *How does the Indexed DataFrame perform on typical Spark SQL operations (e.g., filter, projection, aggregation), and what are its overheads?*

Q4 *How does the Indexed DataFrame perform on real-world applications and how does it integrate into production environments?*

A. Experimental Setup

We evaluate the Indexed DataFrame through experiments on both a private non-virtualized cluster and a production system deployed on a public cloud. We present the cluster and cloud configurations and the SQL benchmarks that we use to assess the performance of the Indexed DataFrame.

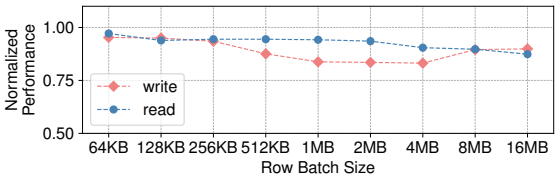


Fig. 5. Read and write performance for the Indexed DataFrame with varying row batch sizes. Results are normalized by the performance of a deployment where the batch size is 4KB (OS page size). Vertical axis does not start at 0 to improve visibility.

Cluster Setup. To run our experiments we have deployed Spark 2.3.0 with the Indexed DataFrame on a private non-virtualized cluster with the datasets stored in an HDFS instance with version 2.7.0. We co-located the HDFS instance with Spark on the same machines. Furthermore, we incorporated the Indexed DataFrame with clusters provisioned from the Amazon EC2 public cloud with the Databricks Runtime [22] system with the datasets stored on S3. For each experiment, we report averages of performance metrics over many runs, adhering to modern standards of performance reproducibility [13], [32]. The Indexed DataFrame is an in-memory table, thus our performance baseline is the default in-memory (columnar) caching mechanism provided by Spark.

Workloads and Datasets. We use two types of queries in our experiments: lookups and joins. We run these against synthetic and real-world datasets which we summarize in Table II. The Social Network Benchmark (SNB) [21] is designed to mimic typical social network structure and behavior. This benchmark was developed for evaluating data analytics applications on updatable graphs. The benchmark generates a social network with power-law structure, similar to Facebook. The benchmark also includes a number of queries to explore the graph. The dataset consists of *edge* and *vertex* tables (jointly over 33 GB), each of which stores various attributes of users in the network.

TPC-DS [33] is the de-facto dataset for assessing the performance of large-scale analytics frameworks. We use various scale factors from 1 to 1000 to assess the performance of indexed joins.

Finally, we use a real-world dataset released by the US Department of Transportation which tracks the performance of domestic flights operated by large air carriers [34]. We use two tables from this dataset, a flights table of 120 GB and a planes table of 420 KB. The `flights` table records the details of each flight that is identified by a unique number (`flightNum`) including the plane identifier, departure and arrival dates, delays, origin, and destination.

B. Sensitivity Analysis

We show an investigation on what parameters make a difference when deploying and running the Indexed DataFrame.

Modern Servers. Memory latency and bandwidth can be a significant bottleneck in large-scale data processing, especially when the execution platform is a complex NUMA architecture. Spark is JVM-based and thus agnostic to the underlying physical hardware of the machines. To deploy a

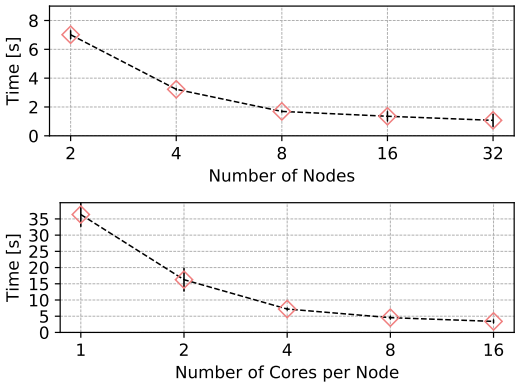


Fig. 6. The scalability properties of the `Indexed DataFrame`: horizontal scalability (top) and vertical scalability (bottom). The points are average runtimes over 10 repetitions and the whiskers standard deviations.

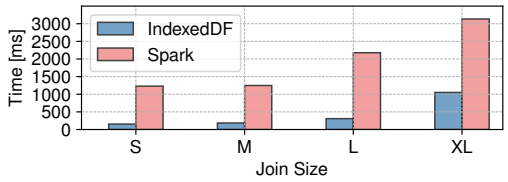


Fig. 7. The performance of the Indexed DataFrame versus vanilla Spark for different sizes of the join probe relation (see Table III). The bars represent the average runtimes over 10 repetitions.

Spark cluster, administrators need to decide how to slice the resources of the worker machines between multiple executors. Standard deployments are usually based on rules-of-thumb such as *one executor per machine* or n *cores per executor*. Optimal allocation is not investigated in prior work focused on performance [35], [36]. Recent research [37], [38] conducted on IBM Power8 clusters shows evidence that Spark is indeed sensitive to NUMA allocations.

Impact of NUMA. We assess the impact of NUMA machines on the performance of the `Indexed DataFrame`. We use `numactl` pinning to control on which socket the Spark executor can allocate threads and memory. Figure 4 summarizes our findings. We plot five combinations of executors and cores per executor, and NUMA pinning. For this experiment we ran a *join* operation between the 1B edge table of SNB SF-1000 and a 10M subset of it (XL join size in Table III). More fine-grained executors perform better, and NUMA pinning is able to further reduce the running time. In all subsequent experiments (with the exception of the ones performed in the cloud) we use the best performing configuration from Figure 4. On dual-socket machines, each of which having 8 cores per socket, we start 4 executors per machine – two per NUMA domain. Each executor is allocated 4 cores.

Row Batch Size. Another important low-level parameter for the Indexed DataFrame is the row batch *size* (i.e., the granularity of the buffers used to store data for each partition of the Indexed DataFrame). Larger allocations determine less batches per partition. We performed a similar experiment to the one described above to perform *reads* (i.e., joins) and measured append performance to determine *write performance*. Figure 5 summarizes the findings of the experiment. We normalize the

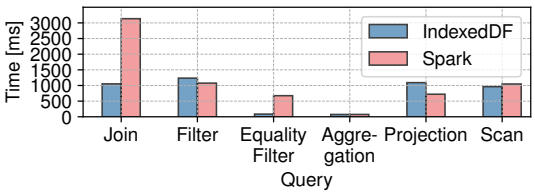


Fig. 8. The performance of the `Indexed DataFrame` versus vanilla Spark for different SQL operators. The bars depict the average runtimes over 10 repetitions.

performance for both reads and writes by the performance achieved using 4 KB batches, the default OS page size. We identify a sweet-spot for both read and write performance at batch sizes of 4 MB. We experimented also with larger batch sizes, of up to 128 MB, but these perform exceptionally poorly for writes and are excluded from the figure.

C. Scalability Analysis

Spark is designed to scale-out on compute clusters with many machines. To reduce the network communication [39] Spark employs delay scheduling, a technique that aims at co-locating the tasks with their data. Incorporating indexing in Spark may, however, impact the scalability of the applications. We evaluate the scalability properties of *join* operations on the Indexed DataFrame using the SNB benchmark by varying both the cluster size and the input data size.

Horizontal and Vertical. To showcase the scalability properties of the `Indexed DataFrame`, we use a join between the 1B edge table of SNB SF-1000 and a 10M subset of it (XL join size in Table III). We show that the `Indexed DataFrame` scales well both *horizontally* and *vertically* for such join queries in which our indexing structure requires a non-trivial amount of shuffle operations.

Figure 6a depicts the horizontal scalability behavior of the Indexed DataFrame. We increase the number of worker machines in our Spark cluster from 2 to 32 and we keep the input dataset size constant. We observe that the speed-up is sub-linear because increasing the cluster size results in more network communication. However, the same behavior applies for regular Spark operation. Furthermore, in Figure 6b we show how our Indexed DataFrame scales vertically. In this experiment, we setup a cluster with 4 worker machines and we vary the number of cores used by the Spark executors to run tasks from 1 to 16. For simplicity, in this experiment we configured a single executor per worker machine. We find that the Indexed DataFrame has close to linear scaling with the number of cores per executor.

Dataset Size. We compare the operation of a hash-join in Spark with our Indexed DataFrame. When two tables are hash-joined, Spark first creates a hash-table on the smaller side of the join which is called the *build* relation. When this relation is small (less than 10 MB), Spark broadcasts it across all worker machines in the cluster to co-locate it with the opposite table of the join – the probe relation. For each row in the *probe* relation, Spark finds corresponding rows from the build relation by lookups in the hash table.

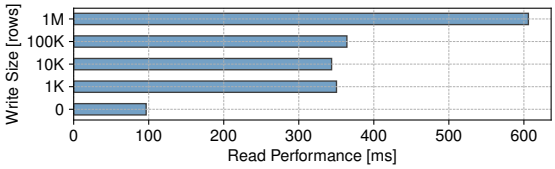


Fig. 9. Read performance latency increase when writing various amounts of rows at a time. Experiments are performed 200 times and results shown represent the mean.

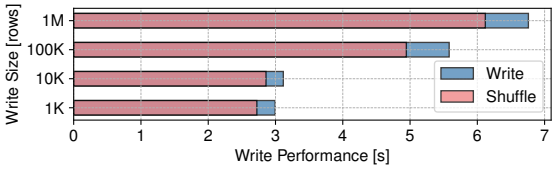


Fig. 10. Write performance throughput of the Indexed DataFrame for various amounts of rows written at a time. Experiments are performed 200 times and the performance results are cumulated. This result applies to both *appendRows* and *createIndex* since they are backed by the same writing mechanism.

The operation of the Indexed DataFrame is in contrast with the execution of a typical Spark hash-join. The index is always pre-built on the side of the join that remains in place, i.e., the larger table (the build side). The index is then used for locally probing the other table, whose partitions are shuffled over the network to co-locate them with the index. When the probing relation increases, the networked communication is prone to become a bottleneck. In Table III we depict the probe relation sizes used in our join queries. We find that irrespective the probe size, our Indexed DataFrame is faster than Spark with speed-ups in the range of 3 and 8 (Figure 7).

D. Indexed DataFrame Microbenchmarks

The Spark dataframe and SQL APIs contain a plethora of operations that programmers can employ to manipulate large-scale datasets. We want to ensure that the Indexed DataFrame delivers similar performance with vanilla Spark on such operations as well as negligible storage overhead.

Microbenchmarks. We compare the performance of the Indexed DataFrame versus vanilla Spark on a selection of SQL operators including join, filter, projection, aggregation, and scan using an in-memory input dataframe that incorporates the large 1 B rows edge table of the SNB benchmark (see Table III). Since the join and filtering operators naturally use the index their performance is significantly improved by the Indexed DataFrame as we show in Figure 8. We observe that the projection and non-equality filters are the only operators that suffer slowdowns because of our Indexed DataFrame. This is because our in-memory representation of the data is based on a row structure which is less efficient than the columnar format adopted by the Spark cache for projections or analyzing single columns.

Append & Index Creation. Writing data into the Indexed DataFrame has two components: the *latency* it adds to read operations (due to the extra row materialization and extra shuffles when actually writing the data), and the *throughput* of effectively appending the data. To measure these, we performed two experiments. The first entails running 200 S

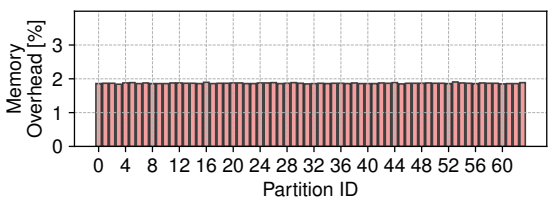


Fig. 11. The memory overhead of the Indexed DataFrame for each partition of a 30 GB table.

join operations (see Table III) in a sequence. Every 5 join operations we also issue an append. This models real-world behavior of users who need to query data sources that get written into regularly. The read performance is influenced by the size of the append. Figure 9 shows the results we achieved. The read performance is significantly influenced by the write size. Writes of at most 100K rows slow down reads by a factor of 3X, but larger writes double the latency to a factor of 6X. However, all this is still acceptable compared to regular Spark operation, where all join sizes take longer than 1 second (see Figure 7), without tolerating appends.

To measure effective writing throughput we performed 200 appends of various row sizes. Figure 10 presents the results. It is immediate that most of the *write* time is dominated by shuffles. This is expected, because the rows have to be sent to the Indexed DataFrame partitions that are responsible for storing them, as the data is hash-partitioned. Spark is known to not be efficient for networked operations [40]. All possible write implementations on Spark would suffer the same shuffle penalty. It is important to notice that the results are similar for both *append* and *createIndex*, as the two APIs perform the same internal operations. Note that the results presented in Figure 10 are cumulated performance results over 200 executions. In the topmost experiment we inserted 200 M rows in batches of 1 M rows, which took just below 7 seconds.

Memory Overhead. Storing a tree-based index next to the actual data may result in memory overhead. To assess how large this overhead is we investigated in detail our previous experiment. When creating the index, the 30 GB SNB edge table is split into partitions, each Spark node storing a subset of those partitions. In our design, each partition stores its own local index in the Indexed Batch RDD, as described in Section III. We measure the overhead of the cTrie index for each of these partitions. To this end, we instrument the Indexed DataFrame code with the JAMM memory meter [41]. Figure 11 plots the memory overhead for the 64 partitions of the SNB edge table. We find that the memory overhead for the Indexed DataFrame is consistently lower than 2% and therefore negligible in comparison with the performance benefits achieved by indexing.

Fault-Tolerance. In the Spark ecosystem, fault-tolerance is achieved via re-computation based on the lineage graph. Whenever a node is lost, all its data has to be re-created by re-running the operations that led to its creation. For the `Indexed DataFrame` this mechanism is no different, however, it entails more work (Section III-D). If an indexed partition is lost, then the index has to be re-created, and if

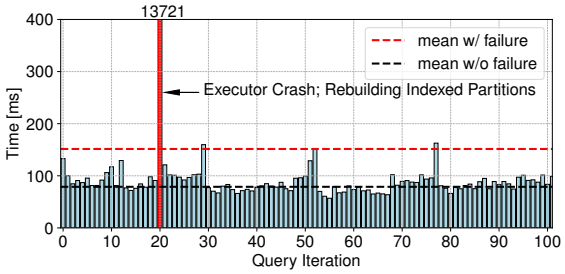


Fig. 12. Indexed DataFrame fault-tolerance overhead when one Spark executor fails during the execution of 200 join queries. The horizontal axis ends at 100 for visibility.

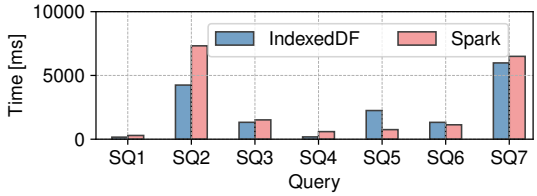


Fig. 13. The improvement of the SNB queries on `Indexed DataFrame` versus Spark on an input dataset of SF 300.

there were any appends on that particular partition, these have to be replayed as well. This process adds an overhead, but we show that once this overhead is paid for, the Indexed DataFrame performance falls back to normal. In a cluster of 8 nodes, we performed 200 S join operations (see Table III) and during the execution time we manually killed a Spark executor which was holding 4 indexed partitions. Figure 12 shows the results. We notice that the failure took place during the execution of the 20th query. Re-creating the index extends the execution time of this query to over 13s, but subsequent queries operate at regular speed and the average execution time is only increased marginally. In conclusion, the overhead incurred by re-creating the index can be tolerated in typical environments where failures occur but are infrequent.

E. Indexed DataFrame in the Real-World

In this section, we analyze the performance of the Indexed DataFrame using real-world workloads and datasets, as well as production deployments. We analyze the performance of a set of queries selected from the real-world SNB benchmark as well as the state-of-the-art TPC-DS and US Flights workloads.

Social-Network Benchmark. We run the complete set of short read SNB queries on an input dataset of scale factor 300. The results are presented in Figure 13. We observe that the `Indexed DataFrame` speeds up all queries, with the exception of SQ5 and SQ6, which are unable to use the index properly. This behavior is similar to the issue identified in Subsection IV-D in which we found that a *columnar* data representation performs better than a *row-based* representation for projections. The access patterns of these two queries trigger the inefficiencies of the row-based representation. However, data representation is orthogonal to the design of the Indexed DataFrame and column-based solutions may affect the scan performance because they are likely to produce more cache misses when materializing all rows [42].

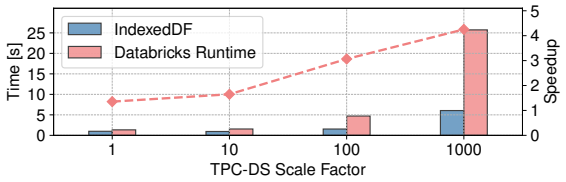


Fig. 14. Performance of the Indexed DataFrame relative to Databricks Runtime using TPC-DS dataset and a join query running on 16 i3.8xlarge Amazon EC2. Bars represent average results computed over 10 runs. Secondary vertical axis represents speedup.

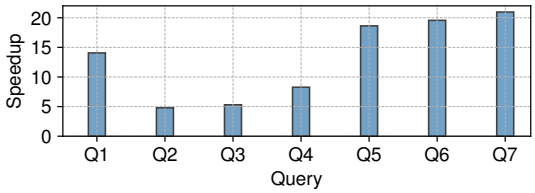


Fig. 15. Speedup of the Indexed DataFrame compared to the Databricks Runtime on 7 queries running on the US Flights dataset. Bars represent average speedup over 10 runs.

Integration in Production. The Indexed DataFrame is implemented as a lightweight library, which can be included in any existing Spark programs, with minimal modifications that basically require attaching the library to the Spark cluster and invoking a simple API to create the index. In this way, we integrated our library in the Databricks Runtime to speed-up the performance of join and look-up queries. In this benchmark we use the BroadcastHashJoin implementation, which is faster than the notoriously slow SortMerge Join.

TPC-DS. Figure 14 shows our results with TPC-DS datasets of increasing size when running a join query (see Table II) on an Amazon EC2 cluster of 16 `i3.8xlarge` running the Databricks Runtime. We observe that by having the Indexed DataFrame created upfront we are able to significantly improve the performance by large margins. We tested several orders of magnitude of the TPC-DS scale factor, from 1 to 1000. The trend here is clear: the larger the dataset, the larger the gap between the indexed version of the join compared to its non-indexed version. This is an effect of the large amount of data that can be filtered out by using the index: the larger the dataset size, the more data is filtered out by the index.

US Flights. Figure 15 shows our results on the Databricks Runtime when executing several queries analyzing the US Flights dataset (see Table II). We compare the performance of the Indexed DataFrame operations on both string and integer columns. Our main findings are as follows. As expected, Indexed DataFrame clearly outperforms the stock Databricks Runtime, by factors of 5-20X. The largest speedups are achieved for integer-key based filters (Q5–Q7). The string-based filter (Q2) is sped up only up to 5X because non-primitive type data incurs additional overhead to be used as a key. Strings need to be hashed into an 32-bit number which is then used as a key in the cTrie. For similar reasons, integer-key joins are sped up more than the string-key joins.

V. Discussion and Related Work

Indexing has been successfully and extensively used in database design for decades to accelerate many types of database operations, including joins [43], [44], [45]. These techniques were not only applicable in single-machine setups, but also in distributed databases [14], [15], [16], in both OLTP [17] and OLAP [18] systems. These database systems that support indexing have in common an ability to handle fine-grained writes, sometimes with strong transaction semantics.

Going forward, in the *big data era* the major technology switch was toward MapReduce-like systems [46], which are designed for achieving performance through scaling out and simplicity. In this category, we can include not only Hadoop [47], but also Spark [7], DryadLinq [48], or Naiad [49], which extend the simplistic MapReduce model to more complex dataflows. Such big data processing systems are thought to be complementary to parallel databases: Stonebraker et al. [50] consider the former to excel at complex analytics and ETL, while the latter at efficiently query-ing large datasets. One of the inefficiencies of MapReduce-like systems come from the fact that they are not designed to support indexing [51].

Significant research effort has been invested into adding indexes to improve the performance of data processing systems. Hadoop has been updated with adaptive HDFS indexes [52], indexes at split level [53]. Moreover, indexes have been used to speed up Hive queries [54] running on top of Hadoop. Indexes have also been built in Spark, for the purpose of quickly solving geospatial queries [55], [56], [57]. However, the downside of all such indexing techniques on current data processing systems is that they do not support any kind of fine-grained updates, either in place or appends, thus not being able to offer interactive performance for applications that do need updates.

SnappyData [58] supports finer-grained updates and indexes by integrating Spark with an external key-value store–Apache GemFire. Managing an external system next to Spark is an overhead that not many organizations can afford. Furthermore, existing Spark programs need be significantly modified to support SnappyData. In our experiments, we were unable to get consistent results from SnappyData when trying to run the benchmarks used in our evaluation. Koalas [4] is another indexed processing system that uses Spark as a backend. Due to this design, its performance is bottlenecked by the behavior of default Spark when it comes to fine-granuar updates as exhibited by our benchmarks. In our experiments, Koalas was orders of magnitude slower for both indexed joins and point queries. Dask is a python-based distributed framework that offers Pandas-like functionality. However, language overheads like global interpreter locks caused Dask in our experiments to be orders of magnitude slower than the Indexed DataFrame.

Another kind of large-scale storage systems for big data processing that provide indexes are the so-called cloud data lakes, such as Delta Lake [59] and Helios [12] or Hyperspace [26]. They offer a tightly-coupled integration between

analytics engines and large-scale cloud (object) storage systems. In contrast to these systems, which offer secondary storage indexing, our work could be seen as an in-memory indexed cache, which is easy to integrate even with systems such as Delta Lake or Helios.

The Timely Dataflow system [60] based on the Naiad [49] design promises to solve most sources of inefficiencies in current data processing systems. It is unclear at the moment whether such systems would even benefit from indexes, but their current performance promises to deliver beyond the inefficiencies of current data processing systems. However, their application in practice lags behind the popularity of Spark. Finally, using machine learning techniques to learn and optimize indexes [61] is a promising research direction, provided it can build upon robust indexing mechanisms. Indexed DataFrames could be such a mechanism for AI-driven indexes

VI. Conclusion

In this paper we argue for bridging the divide between traditional relational database systems and data analytics platforms like Spark to get the best of both worlds. Concretely, we showed that adding indexing to the main underlying abstractions of Spark can greatly improve performance in many cases. The presented Indexed DataFrame brings indexing to the Spark data processing engine, as well as fine-grained appends with minimum performance and memory overheads, and large-scale gains for workloads that make use of the index, such as joins or point lookups. The Indexed DataFrame is designed as a lightweight library that can be included in any Spark program, from any Spark distribution–be it Apache, or Databricks Runtime–to achieve a boost in performance for workloads that make use of indexes, of up to 20X.

Acknowledgements

Part of this work was conducted while the first author was an intern at Databricks. We would like to thank Herman van Hovell, Adrian Ionescu for their suggestions on the implementation of the project, as well as Matei Zaharia for his valuable comments on the manuscript of the paper. The work in this article was in part supported by The Dutch National Science Foundation NWO Veni grant VI.202.195.

References

[1] Hey *et al.*, *The fourth paradigm*. Microsoft Research, 2009, vol. 1.
[2] Barclay *et al.*, "Microsoft terraserver: a spatial data warehouse," in *SIGMOD*, 2000.
[3] McKinney, *Python for data analysis*. " O'Reilly Media, Inc.", 2012.
[4] "Koalas: Pandas API on Apache Spark," https://koalas.readthedocs.io/en/latest/.
[5] Armbrust *et al.*, "Spark SQL: relational data processing in spark," in *SIGMOD*, 2015.
[6] Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th python in science conference*, no. 130-136, 2015.
[7] Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012.
[8] Vuppalapati *et al.*, "Building an elastic query engine on disaggregated storage," in *NSDI*, 2020.
[9] Kreps *et al.*, "Kafka: A distributed messaging system for log processing," in *NetDB*, 2011.
[10] Armbrust *et al.*, "Structured streaming: A declarative API for real-time applications in apache spark," in *SIGMOD*, 2018.
[11] M. Armbrust *et al.*, "Databricks delta: A unified data management system for real-time big data," 2017.
[12] Potharaju *et al.*, "Helios: hyperscale indexing for the cloud & edge," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, 2020.
[13] Uta *et al.*, "Is big data performance reproducible in modern cloud networks?" in *NSDI*, 2020.
[14] Danzig *et al.*, "Distributed indexing: A scalable mechanism for distributed information retrieval," in *ACM SIGIR*, 1991.
[15] Nam and Sussman, "Spatial indexing of distributed multidimensional datasets," in *CCGrid*, 2005.
[16] Dehne *et al.*, "A distributed tree data structure for real-time OLAP on cloud architectures," in *IEEE Big Data*, 2013.
[17] Diaconu *et al.*, "Hekaton: SQL server's memory-optimized OLTP engine," in *SIGMOD*, 2013.
[18] Gupta *et al.*, "Index selection for OLAP," in *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997, Birmingham, UK*, 1997.
[19] Ramnarayan *et al.*, "Snappydata: A hybrid transactional analytical store built on spark," in *SIGMOD*, 2016.
[20] Brezinski and Armbrust, "Threat detection and response at scale," in *Spark Summit*, 2018.
[21] Erling *et al.*, "The LDBC social network benchmark: Interactive workload," in *SIGMOD*, 2015.
[22] "Databricks Runtime," https://www.databricks.com.
[23] "Apache druid," http://druid.apache.org.
[24] "Splicemachine," https://www.splicemachine.com.
[25] Lakshman and Malik, "Cassandra: a decentralized structured storage system," *EuroSys*, vol. 44, no. 2, 2010.
[26] R. Potharaju, T. Kim, E. Song, W. Wu, L. Novik, A. Dave, A. Fogarty, P. Pirzadeh, V. Acharya, G. Dhody *et al.*, "Hyperspace: The indexing subsystem of azure synapse."
[27] Graefe, "Sort-merge-join: An idea whose time has (h) passed?" in *IEEE ICDE*, 1994.
[28] DeWitt and Gerber, *Multiprocessor hash-based join algorithms*. University of Wisconsin-Madison, Computer Sciences Department, 1985.
[29] Prokopec *et al.*, "Concurrent tries with efficient non-blocking snapshots," in *PPoPP*, 2012.
[30] Driscoll *et al.*, "Making data structures persistent," *Journal of computer and system sciences*, 1989.
[31] Shvachko *et al.*, "The hadoop distr. file system," in *IEEE MSST*, 2010.
[32] Maricq *et al.*, "Taming performance variability," in *OSDI*, 2018.
[33] Nambiar and Poess, "The making of tpc-ds," in *VLDB*, 2006.
[34] "US Flights On-Time Airline Statistics," http://stat-computing.org/dataexpo/2009/the-data.html.
[35] Ousterhout *et al.*, "Making sense of performance in data analytics frameworks," in *USENIX NSDI*, 2015.
[36] Chaimov *et al.*, "Scaling spark on HPC systems," in *ACM HPDC*, 2016.
[37] Chiba and Onodera, "Workload characterization and optimization of TPC-H queries on apache spark," in *ISPASS*, 2016.
[38] Baig *et al.*, "Performance characterization of spark workloads on shared NUMA systems," in *BigDataService*, 2018.
[39] Zaharia *et al.*, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *EuroSys*, 2010.
[40] Zhang *et al.*, "Riffle: optimized shuffle service for large-scale data analytics," in *Proceedings of the Thirteenth EuroSys Conference*, 2018.
[41] "JAMM Memory Meter," https://github.com/jbellis/jamm.
[42] Wen *et al.*, "CORES: towards scan-optimized columnar storage for nested records," *TOS*, 2019.
[43] Roussopoulos, "View indexing in relational databases," *ACM Trans. Database Syst.*, vol. 7, no. 2, 1982.
[44] Valduriez, "Join indices," *ACM Trans. Database Syst.*, 1987.
[45] O'Neil and Graefe, "Multi-table joins through bitmapped join indices," *SIGMOD*, 1995.
[46] Dean and Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004.
[47] White, *Hadoop - The Definitive Guide*. O'Reilly, 2009.
[48] Yu *et al.*, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *OSDI*, 2008.
[49] Murray *et al.*, "Naiad: a timely dataflow system," in *SOSP*, 2013.
[50] Stonebraker *et al.*, "Mapreduce and parallel dbmss: friends or foes?" *Commun. ACM*, vol. 53, no. 1.

[51] DeWitt and Stonebraker, "Mapreduce: A major step backwards," *The Database Column*, vol. 1, 2008.
[52] Richter *et al.*, "Towards zero-overhead static and adaptive indexing in hadoop," *VLDB Journal*, 2014.
[53] Eltabakh *et al.*, "Eagle-eyed elephant: split-oriented indexing in hadoop," in *EDBT/ICDT*, 2013.
[54] Mofidpoor *et al.*, "Index-based join operations in hive," in *IEEE Big-Data*, 2013.
[55] Xie *et al.*, "Simba: Efficient in-memory spatial analytics," in *SIGMOD*, 2016.
[56] Cui *et al.*, "Indexing for large scale data querying based on spark sql," in *2017 IEEE 14th International Conference on e-Business Engineering (ICEBE)*, 2017.
[57] Tang *et al.*, "Locationspark: A distributed in-memory data management system for big spatial data," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, 2016.
[58] Mozafari *et al.*, "Snappydata: A unified cluster for streaming, transactions and interactice analytics," in *CIDR*, 2017.
[59] Armbrust *et al.*, "Delta lake: high-performance acid table storage over cloud object stores," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, 2020.
[60] "Timely Dataflow," https://github.com/TimelyDataflow.
[61] Kraska *et al.*, "The case for learned index structures," *CoRR*, vol. abs/1712.01208, 2017.