[Demo] Low-latency Spark Queries on Updatable Data

Alexandru Uta*
VU Amsterdam
A.Uta@vu.nl

Bogdan Ghit
Databricks
bogdan.ghit@databricks.com

Ankur Dave
UC Berkeley
ankurd@eecs.berkeley.edu

Peter Boncz
CWI Amsterdam
P.Boncz@cwi.nl

ABSTRACT

As data science gets deployed more and more into operational applications, it becomes important for data science frameworks to be able to perform computations in interactive, sub-second time. Indexing and caching are two key techniques that can make interactive query processing on large datasets possible. In this demo, we show the design, implementation and performance of a new indexing abstraction in Apache Spark, called the `Indexed DataFrame`. This is a cached DataFrame that incorporates an index to support fast lookup and join operations, and supports updates with multi-version concurrency. We demonstrate the Indexed Dataframe on a social network dataset using microbenchmarks and real-world graph processing queries, in datasets that are continuously growing.

ACM Reference Format:
Alexandru Uta, Bogdan Ghit, Ankur Dave, and Peter Boncz. 2019. [Demo] Low-latency Spark Queries on Updatable Data. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3299869.3320227

1 INTRODUCTION

Data science is pervading organizations that seek to exploit the power of data in a wide range of use cases. Interactive query processing on constantly updated, large datasets is encountered in many applications, ranging from graph processing algorithms [5], to geo-spatial analytics, to threat detection and response [4]. In this demo, we showcase the power of indexing and caching for achieveing interactive, sub-second response time for such applications.

Apache Spark has become widely adopted in data science applications, as it is a versatile framework, which allows users to run their own User Defined Functions (UDFs), quickly, on datasets of very large scale. In the absence of an on-premise cluster, good performance is also achieved elastically and flexibly in the cloud, for instance using the Databricks Spark service. Beyond running UDFs at scale, Spark also allows SQL access to datasets, empowering analytics on so-called data lakes. Data pipelines in Spark can be tightly coupled with machine-learning algorithms; where Spark takes care of data management for both training and deployment of learned models. As such, Spark is the linking component in a Unified Analytics framework, that encompasses data engineering (cleaning, transforming), data analytics (SQL), as well as machine learning and model deployment.

The many different characteristics of these *data science* workloads imply that Spark is getting deployed in much more diverse usage scenarios than when it was originally conceived, when RDDs were assumed to be static and read-only, and where queries or jobs were batch-oriented. In contrast, most Spark programs now use higher level interfaces, such as DataFrames and SQL that get the benefit of automatic query optimization, and rather than operating on static read-only files, data enters the system increasingly in streaming fashion, typically via Kafka [6] and/or the Spark structured streaming interface [3]. Databricks also introduced Delta Tables [2], a layer on top of Parquet-stored Dataframes, which provides transactional query integrity under continuous batch-updates.

We demonstrate a new kind of Spark data structure called the `Indexed DataFrame`. It is aimed at supporting low-latency joins and point lookups in interactive workloads on data that is moving all the time, also using relatively fine-grained updates. As such, it enables Spark in new use cases, such as on-line threat detection and response [4], and real-time social network monitoring and dashboarding [5]. Notice that online analytics on changing graphs is a challenging use case for Spark as graph navigation is very join-intensive, while

*Work performed while the author was an intern at Databricks.

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3320227

at the same time updates to the graph invalidate caching of Dataframes that is needed to keep such joins fast.

The `Indexed DataFrame` is an *updatable* Dataframe that remains *cached* even when data is added, and has a built-in concurrent cTrie[7] *index* that allows for sub-linear lookup for non-unique keys; that is exploited in low-latency filter and join operations. Whereas the `Indexed DataFrame` has a relatively low memory overhead in addition to the original data, it improves the query performance in multiple types of scenarios: queries on updatable graphs [5], standard database operations, and threat detection [4], where using indexes minimizes the amount of data that is materialized and processed. The `Indexed DataFrame` can be deployed as a lightweight library imported into Spark programs and sessions, even though it deeply integrates into internal Spark components such as the RDD cache as well as the Catalyst optimizer.

The contributions of our work are as follows:

(1) We demonstrate the design and implementation of the `Indexed DataFrame` in Spark. We explain the API we support to index Spark dataframes, we show how we integrated the index in Spark's Catalyst optimizer, and we present the underlying data structure. (Section 2).
(2) We showcase an initial performance evaluation of the `Indexed DataFrame`. We show the performance improvement of the `Indexed DataFrame` on both standard SQL workloads and real-world graph processing workloads (Section 3) and we present the details of our demonstration (Section 4).

2 INDEXED DATAFRAME DESIGN

In this section, we propose the `Indexed DataFrame`, a data abstraction that we can use to manipulate indexed datasets. We present the API and the main design and implementation details in Spark. The `Indexed DataFrame` is built on top of the Dataframe API supported by Spark SQL [1] and operates as a lightweight library, which can be added to any existing Spark program.

Spark Programming Interface. To address the requirements of a large spectrum of big data (analytics) applications, we have designed the `Indexed DataFrame` to support the following operations: `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, strings, and datetime).

As we want to store the `Indexed DataFrame` in the memory of the Spark executors, instantiating the index should be immediately followed by a caching operation. Furthermore,

Listing 1: The Indexed Data Frame API

```
// creating an index
var indexedDF = regularDF.createIndex(colNo)
// caching the indexed data frame
var indexedDF = indexedDF.cache()
// looking up keys returns a data frame containing
    all rows
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"))
```

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.

Integration with Catalyst. In Figure 1 we present 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 [1] query optimizer. Our library includes *optimization rules* that make regular Spark SQL queries aware of our custom indexed operations.

In Spark SQL, a query has an abstract representation called query plan which is converted through a sequence of transformations into an optimized query plan that is finally executed on the cluster. The Catalyst optimizer translates the query into a logical plan that provides a high-level representation of each operation without defining how to perform the actual computation. Catalyst optimization rules transform the logical plan into a physical plan with specific instructions on how to execute the query.

We develop index-aware optimization rules in Catalyst that 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.

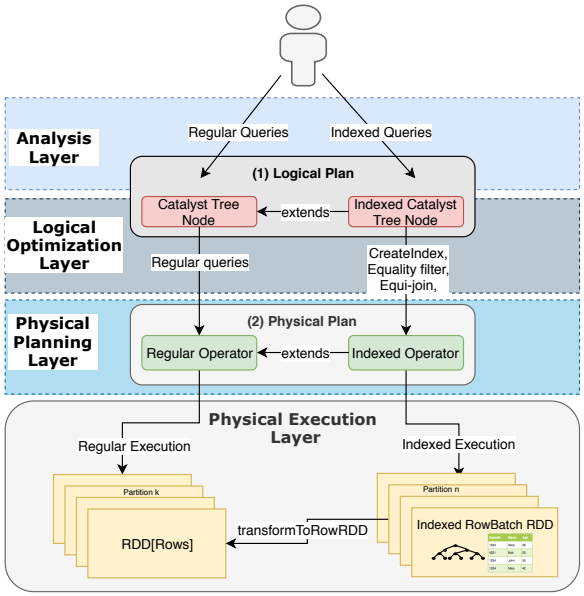

Figure 1: `Indexed DataFrame` logical flow. Users write SQL queries or use the Dataframe API. Catalyst rules determine whether the queries are regular or indexed. If regular, they follow the regular Spark Catalyst execution. If indexed, special rules, and optimization strategies are applied such that indexed execution is triggered. Moreover, an indexed RowBatch RDD can always fall back to a regular Spark Row RDD to trigger regular execution on top of `Indexed DataFrame`.

Our rules ensure that the `Indexed DataFrame` operations are always triggered when executing queries on indexed data. Similarly, for queries on basic Spark Dataframes that do not employ indexing, our implementation falls back to the default Spark behavior.

The Indexed Row-Batch RDD. Spark datasets are typically partitioned across multiple worker nodes so that the framework can divide jobs into multiple tasks that can be executed in parallel on multiple *executor* nodes. The Dataframe API can perform relational operations on Spark's built-in distributed collections, i.e., the resilient distributed datasets widely known as RDDs [8]. The `Indexed DataFrame` operates in a similar way by partitioning data across multiple executor nodes, but it requires a custom implementation of the *RDD* abstraction: the indexed *Row-Batch RDD*.

Our custom *RDD* implementation stores data *in-memory*. Each RDD partition is composed of three data structures: (1) a *cTrie* [7], which represents the index, (2) a set of *row batches*, which stores 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.

The cTrie stores a pointer to the latest appended row associated with a given key. If there are multiple rows associated with a key, the backward pointer data structure consists of a set of linked lists, one per unique key. This backward pointer 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, dense 64-bit numbers, each containing the row batch number, the offset within a row batch, and the size of the previous row indexed on the given key.

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 $4x2^{31}$ MB data per core. Spark 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 [1]. Both the batch and row sizes are configurable parameters.

The operation of the `Indexed DataFrame` on a single partition is similar both for index creation and for appends. First, each row is inserted in the row batch. 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. An `Indexed DataFrame` lookup consists of a lookup in the cTrie, followed by a traversal of the backward pointers in case multiple rows are associated with the same key.

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. The `Indexed DataFrame` is *hash* partitioned on the indexed column. This ensures a better load balancing when the key ranges are not known apriori. 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. For each partition, we create a cTrie and we insert the rows in their batches as described in Section 2.

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. When the Dataframe size is small enough to be broadcasted efficiently, our implementation falls back to a broadcast-join instead of a shuffle.

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

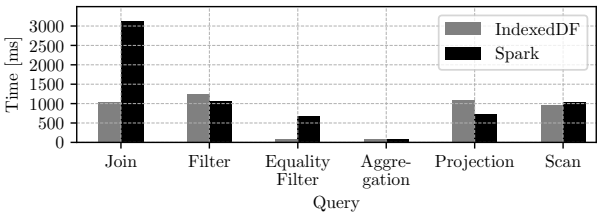

Figure 2: `Indexed DataFrame` vs. vanilla Spark.

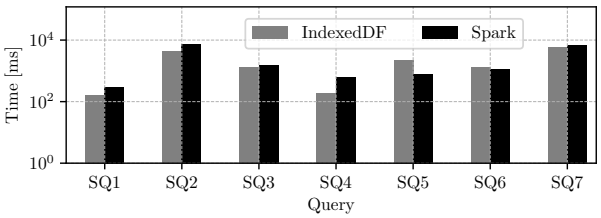

Figure 3: SNB SF300 Simple Read Queries on `Indexed DataFrame` vs. Spark. Logarithmic vertical axis.

3 EVALUATION

In this section we present the workloads and the configuration of our system for the experimental evaluation of the Indexed Dataframe in Spark. Our workloads consist of several join operations and queries on updatable graphs.

Dataset. We run our experiments on datasets generated using the Datagen tool provided by the SNB benchmark [5]. These datasets represent graph structures, which are represented as *edge* and *vertex* tables.

SQL Operators. We run a selection of SQL operators (i.e., join, filter, projection, aggregation, scan) on `Indexed DataFrame` and vanilla Spark in which all operations are applied on *cached* (e.g., in-memory) dataframes. All the operators were applied to the *person-knows-person* tables, while the join is computed between *person-knows-person*, and *person* tables.

Operators that use the index (i.e., join and equality filter) are significantly sped up compared to the others. The results are plotted in Figure 2. The only operator significantly slowed down by `Indexed DataFrame` is the projection. This is due to the fact that our in-memory represesentation follows a Row structure, while the Spark cache stores data in a columnar format, which is much more efficient to project.

SNB Queries. We run the 7 SNB defined simple read queries, on data SF300. The `Indexed DataFrame` speeds up all queries, with the exception of Q5 and Q6, which cannot make use of the index. The results are presented in Figure 3.

4 DEMONSTRATION DETAILS

We explain the demonstration setup we will employ in order to show the performance of our Indexed Dataframe in Spark SQL using real-world datasets and workloads. Processing dynamically changing graph structures and filtering large data volumes are very attractive applications for large audiences and organizations, with many practical implications. We focus on a graph processing use-case, where data scientists write queries for understanding human behavior and interaction, and preferences in social networks.

Demo Setup. We evaluate our Index Dataframe library on Spark clusters deployed on virtual machines in Amazon EC2. During the demonstration we will have access to a 10-node Spark cluster. Furthemore, we will use the Social Network Benchmark to generate a large-scale graph structure stored on Amazon S3, and the Apache Kafka [6] engine to handle the constant updating stream that is mutating the graph.

Demo Visualization. We will demonstrate a real-time graph-monitoring dashboard. The dashboard shows a visualization of the evolution of (a part of) the graph over time, but also the execution of queries using the `Indexed DataFrame`, and vanilla Spark. The demonstration audience will then experience first-hand the power of the indexed operation as the low-latency, or speedup, with which the `Indexed DataFrame` queries results are returned.

5 CONCLUSION

In this demo, we showcase a lightweight Spark library, called `Indexed DataFrame`, that enables interactive response times on join and lookup queries, even on large datasets that grow continuously; thanks to building in multi-versioning and indexing in Dataframes, integrating with the Spark caching infrastructure and its Catalyst optimizer. Our demo is a graph monitoring use case, concurrently handling the update workload of the Social Network Benchmark (SNB), and transparently running SNB queries both on vanilla Spark and Spark using `Indexed DataFrames`. Our initial results show that the `Indexed DataFrame` can achieve up to 8X speed-ups relatively to the vanilla Spark implementation.

REFERENCES

[1] Michael Armbrust et al. 2015. Spark sql: Relational data processing in spark. In *SIGMOD*.
[2] Michael Armbrust et al. 2017. Databricks Delta: A Unified Data Management System for Real-time Big Data. (2017).
[3] Michael Armbrust et al. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *SIGMOD*.
[4] Dominique Brezinski and Michael Armbrust. 2018. Threat Detection and Response at Scale. In *Spark Summit*.
[5] Orri Erling et al. 2015. The LDBC social network benchmark: Interactive workload. In *SIGMOD*.
[6] Jay Kreps et al. 2011. Kafka: A distributed messaging system for log processing. In *NetDB*.
[7] Aleksandar Prokopec et al. 2012. Concurrent tries with efficient non-blocking snapshots. In *Acm Sigplan Notices*.
[8] Matei Zaharia et al. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*.