Low-latency Spark Queries on Updatable Data

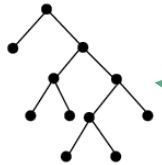
Alexandru Uta, Bogdan Ghit, Ankur Dave, Peter Boncz

Highly Dynamic Workloads on Updatable data

- Nowadays, many datasets are constantly updated
- Need to perform (near-)real-time queries on these data while being updated
- E.g.:
  - large twitter dataset which is continuously expanding
  - Interactively analyzing this (graph-like) dataset means mostly applying joins
- Shuffling and creating hash tables for each operation is expensive

Problem: Apache Spark unfit for dynamic workloads

- Such workloads are currently run in large-scale distributed setups
- But Spark does not:
  - **Store indexes**
  - **Support fine-grained updates/appends**
  - **Support fast point-lookups**
  - **Use such lookups for joins**



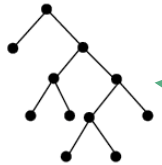
Col1	Col2	Col3
a	b	c
d	e	f
g	h	i

JOIN on Col1

Col1	Col2
d	x

Solution: Indexed DataFrame

- Indexed DataFrame supports:
  - **Equality indexes**
  - **Fine-grained appends**
  - **Fast point-lookups**
  - **Index-based joins**



Col1	Col2	Col3
a	b	c
d	e	f
g	h	i

JOIN on Col1

Col1	Col2
d	x

Indexed DataFrame API – extends DataFrame API

1) **Index Creation:**

indexedDF = regularDF.createIndex(columnNumber: Int)

2) Append:

newIndexedDF = indexedDF.appendRows(regularDF: Dataframe)

3) Lookups:

regularDF = indexedDF.getRows(key: AnyVal)

4) **Inner Equi-Joins:**

a) SELECT * from indexedDF JOIN regularDF ON indexedDF.col1 = regularDF.col2
b) indexedDF.join(regularDF, Seq("col"))

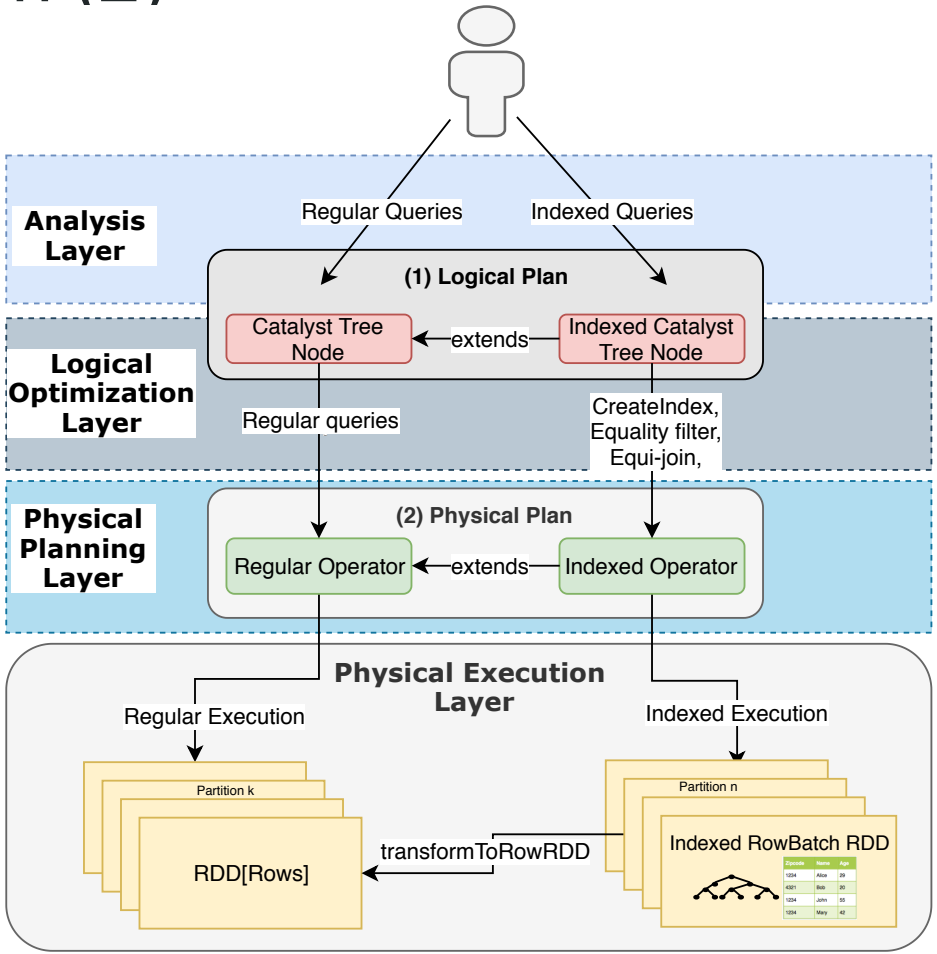
Design & Implementation (1)

- **Goal: easy to use, easy integration with Spark**

- Standalone **sbt** project, does **not** modify Spark source code
- Included in any Spark program like a standalone library
- Works with **Apache Spark & Databricks Runtime**
- **Strategies** and **rules** to convert to and support Indexed operators

Design and Implementation (2)

- Additional rules determine whether query is indexed or not
- We extend catalyst logical and physical operators
- Indexed RowBatch RDD stores indexed data, is able to fall back to regular row RDD
- Indexed DataFrame can fall back to regular operation



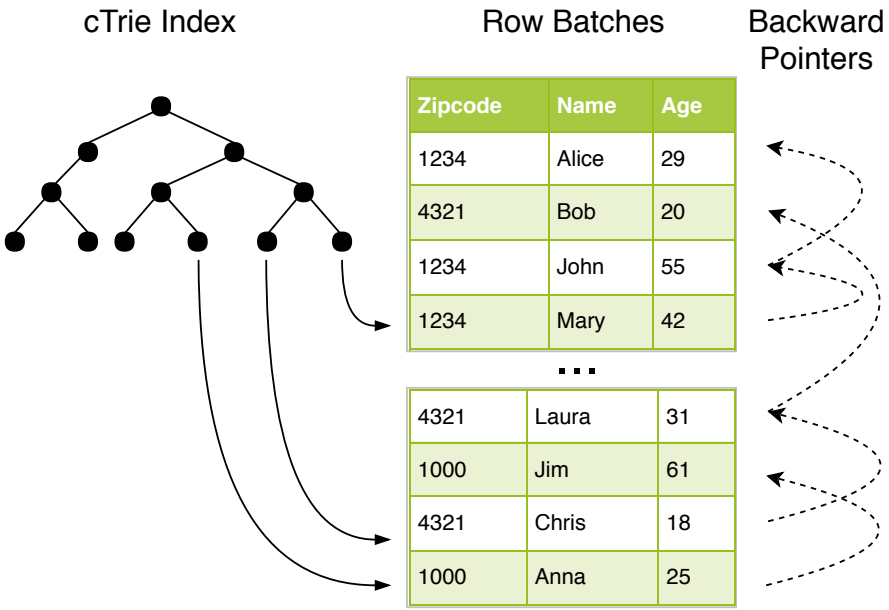
Design & Implementation (3)

- Partition the data by indexed key (by range or hash)

- cTrie (concurrent trie) to store the index
- RowBatches (4MB size) to store data

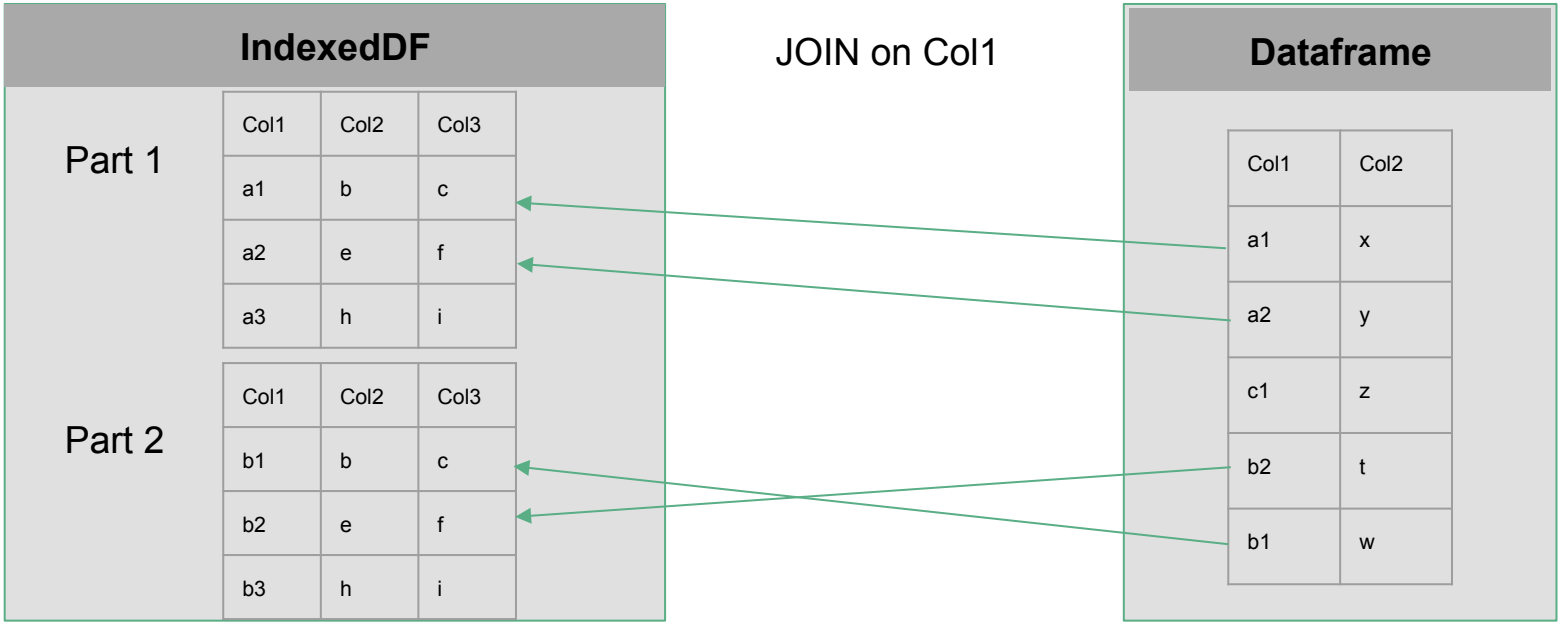
- Problem: Supporting graphs
  => duplicate keys (when storing edges)

- Solution:
  - cTrie stores pointer to last row with same key
  - Backward pointers to previous rows

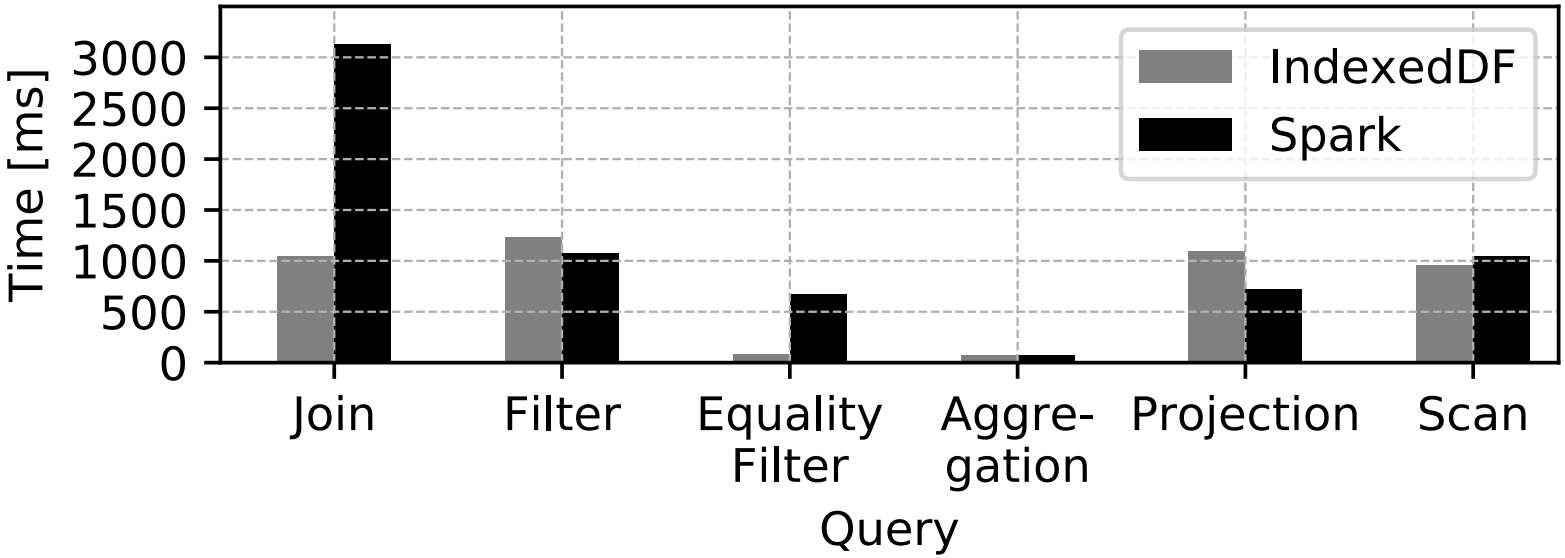


Design & Implementation (4)

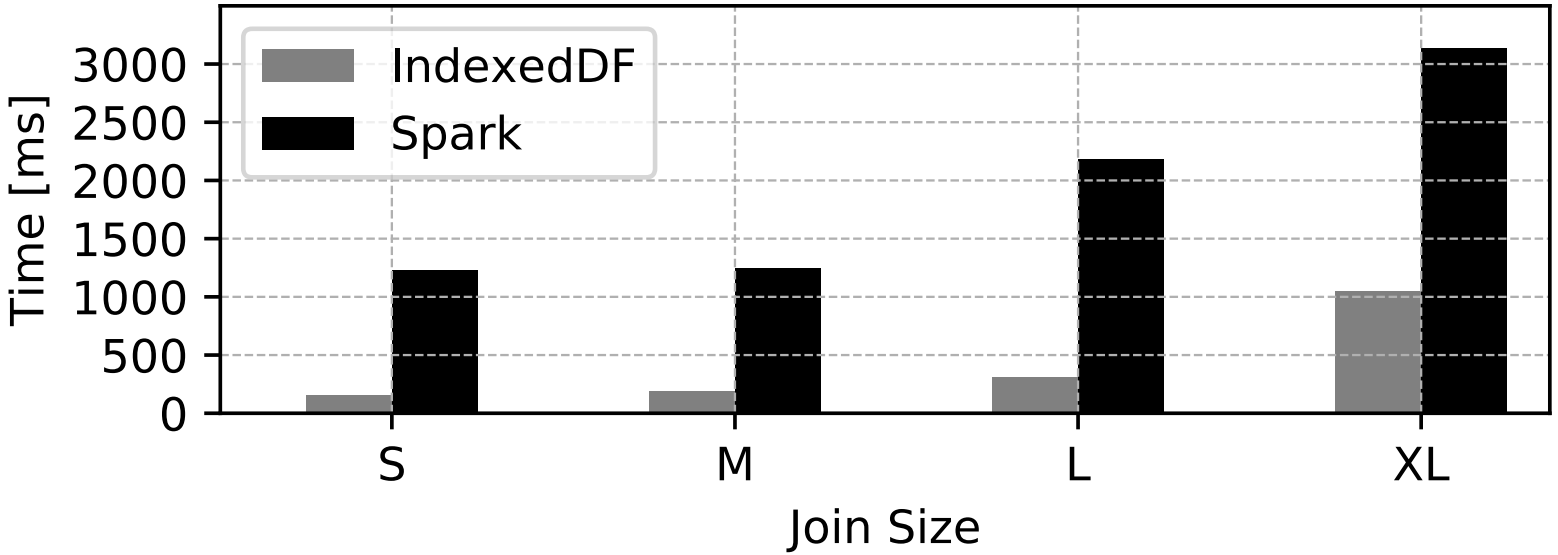
- Achieving **locality** (i.e., not moving around indexed data):
  - **append**: partition the input data by same key then shuffle + local append
  - **join:** partition the *right* relation by join key then shuffle + local lookup



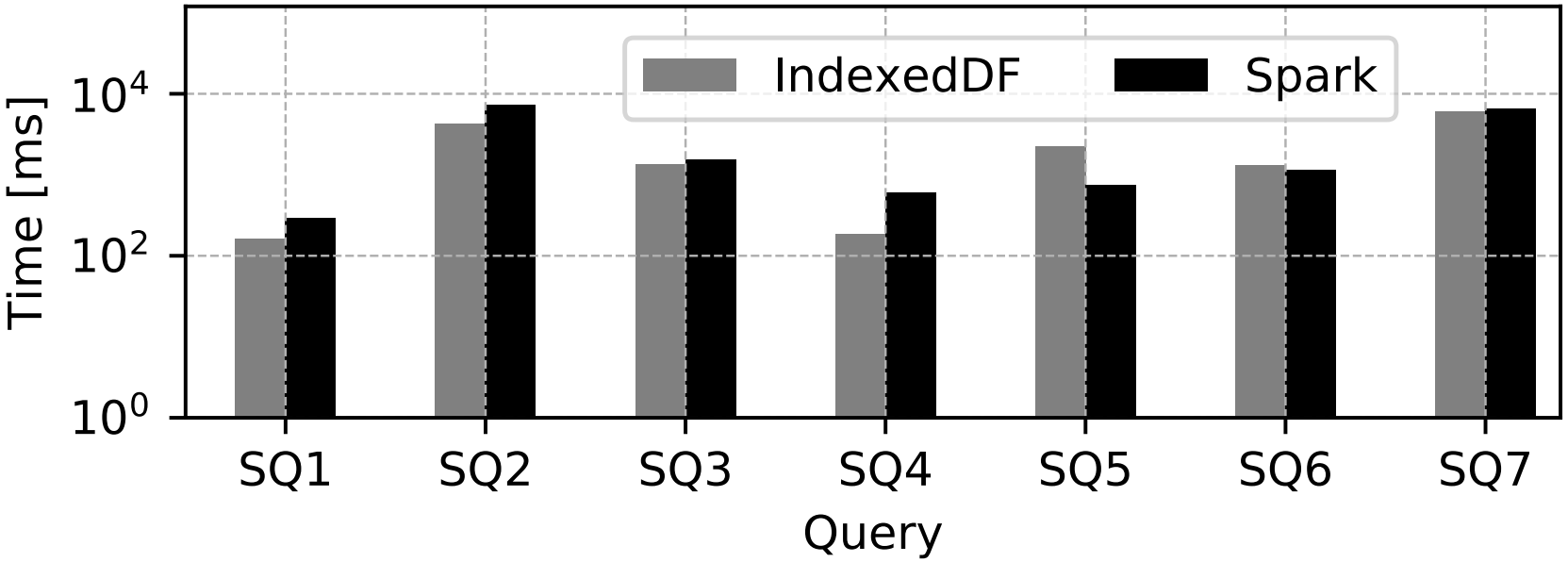
IndexedDF vs. Cached Spark: Various Operators



IndexedDF vs. Cached Spark: Join, various sizes



IndexedDF vs. Cached Spark: LDBC SNB (SF 300)



Conclusion and Future Work

- Presented as Demo @SIGMOD
- Indexing on Dataframes is feasible
- Good performance improvement (3X - 8.5X) for joins
- Promising performance improvement for SNB
- Explore behavior for more complex queries and different workloads

Dataset & Workloads

- Datagen scale factor 1000 (graph)
- Edge table: 1B rows
- Vertex table: 10M rows
- Workload: join edge table (indexed) with vertex table (non-indexed)

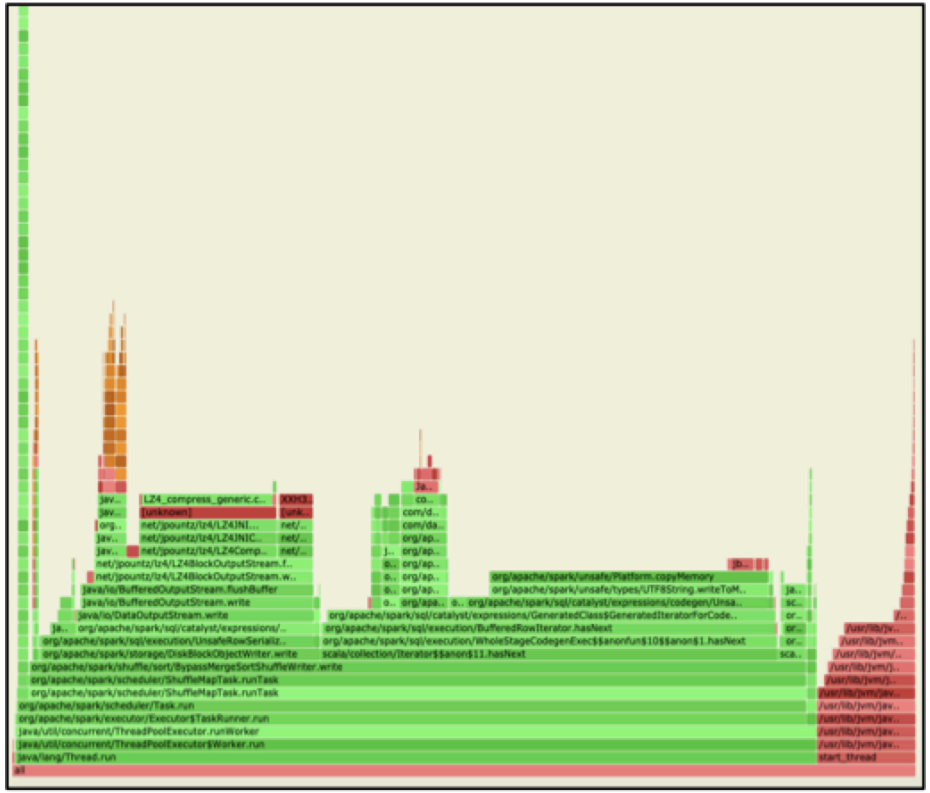
Workload Scale	Edge Table	Vertex Table
XS	1B rows	10K rows
S	1B rows	100K rows
M	1B rows	1M rows
L	1B rows	10M rows

Empirical Evaluation

- Hardware platform:
  - DAS-5 cluster @VU Amsterdam
  - Each node 16 cores (two NUMA nodes), 64 GB RAM, FDR InfiniBand (56 Gbit/s)

- Experiments performed:
  - Indexed DataFrame vs. vanilla Spark (cached)
  - Multiple join sizes
  - Multiple SQL operators
  - LDBC SNB simple queries

Databricks Runtime 4.3 - 5 runs



55 seconds

Indexed DataFrame - 5 runs

2 seconds