Spinach: An Ad-hoc Query Engine on Top of Spark SQL

Cheng Hao(hao.cheng@intel.com)

Outlines

- Major features
- Getting Starts
- Implementation & Design principles
- Micro benchmark the demo data
- Future Plan

About Me

- Active Spark Contributor in Apache Open Source
- Engineering Manager from BDT of Intel APAC
- Leading the IA Optimization for Spark at Intel

Motive

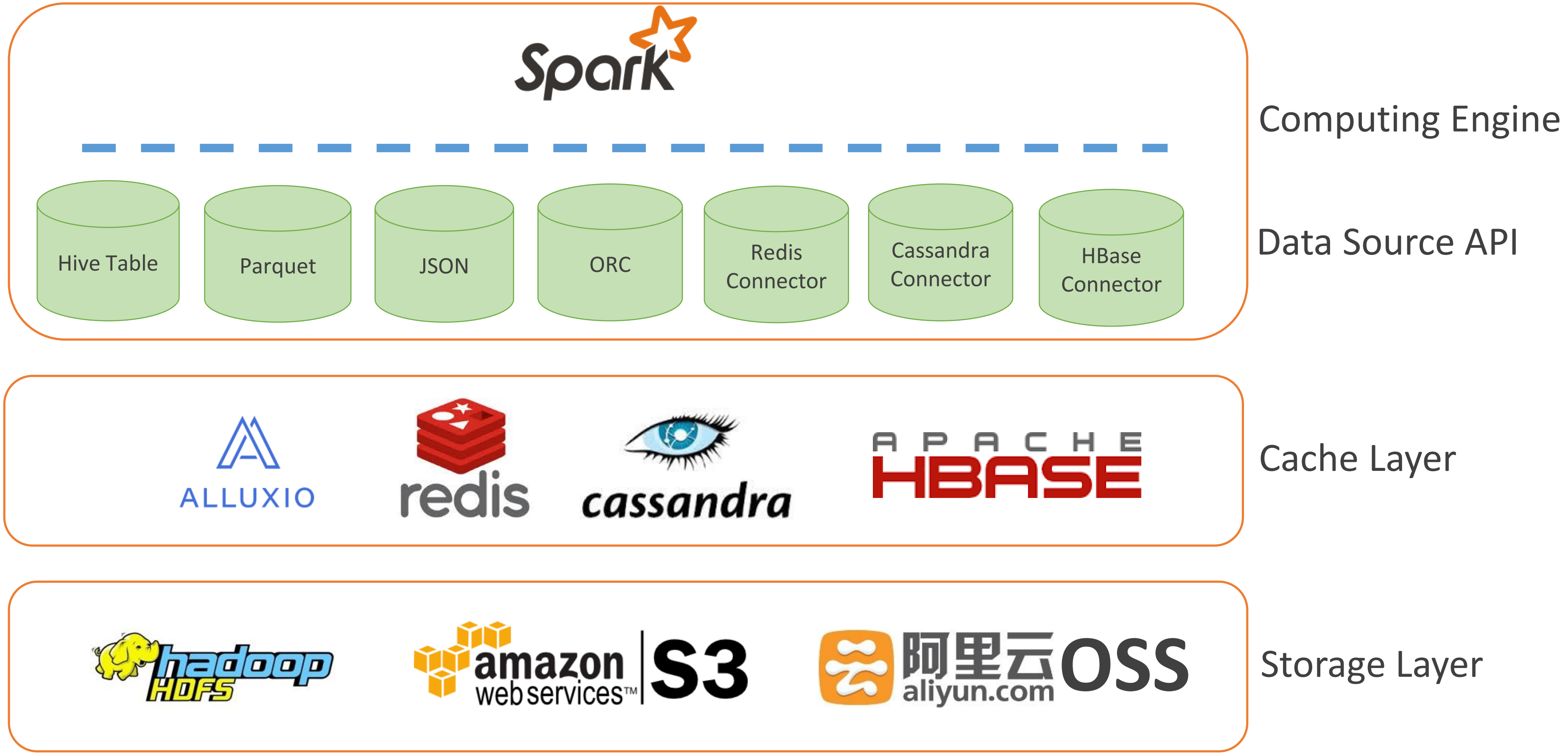
How to accelerate SQL queries with Spark SQL?

Motive

- Tungsten
  - (Offheap) Data oriented Memory Management
  - Cache-aware computation
  - Code Generation
- Tungsten II
  - Whole Stage Code Generation
  - Vectorization
- …

What Else?

Why Spinach Can Accelerate the Ad-hoc Query?



Computing Engine

Data Source API

Hive Table

Parquet

JSON

ORC

Redis Connector

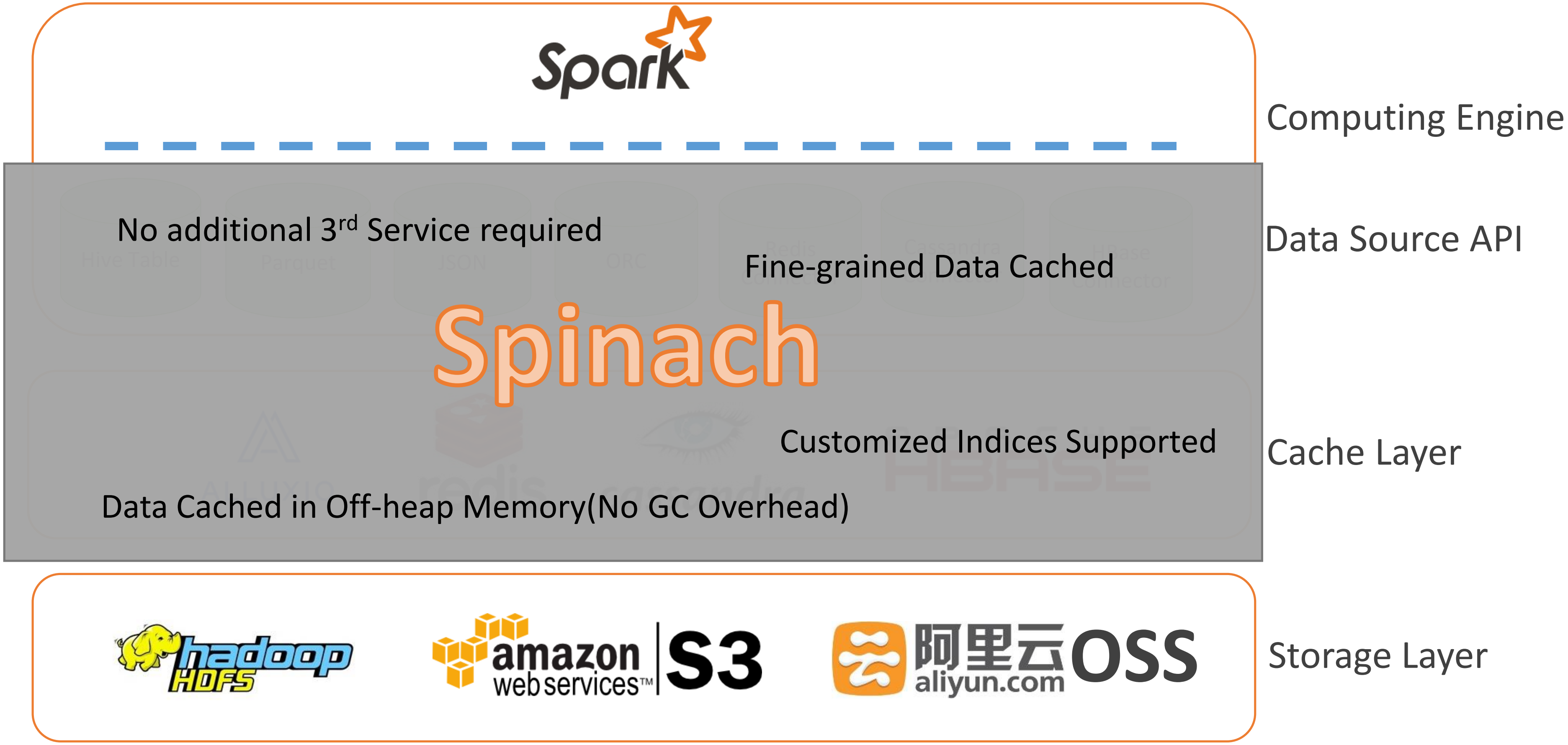
Cassandra Connector

HBase Connector

Cache Layer

Storage Layer

Why Spinach Can Accelerate the Ad-hoc Query?



Computing Engine

Data Source API

Cache Layer

Storage Layer

Spinach

No additional 3rd Service required

Fine-grained Data Cached

Customized Indices Supported

Data Cached in Off-heap Memory(No GC Overhead)

Getting Started

1. Start the Spark SQL Shell and Load the Spinach Package

```
$SPARK_HOME/bin/spark-sql --jars spinach-0.1.jar
```

2. Create a Spinach backend Data Source Table

```
spark-sql> CREATE TABLE src(a INT, b STRING, value INT) USING org.apache.spark.sql.execution.datasources.spinach;
```

3. Add Index Support the Data Source Table

```
spark-sql>CREATE INDEX idx_1 ON src (a);
```

4. Ad-hoc Query by auto enable the indices

```
spark-sql> INSERT INTO TABLE src SELECT key1, key2, value FROM xxx;
spark-sql> SELECT MAX(value) FROM src WHERE a > 100 AND a <= 120 AND b=‘spinach’;
spark-sql> CREATE INDEX idx_2 ON src (a, b);
spark-sql> SELECT MAX(value) FROM src WHERE a>=100 AND b=‘spinach’;
spark-sql> DROP INDEX idx_2;
spark-sql> SELECT MAX(value) FROM src WHERE a>=100;
```

Auto trigger the index idx_1

Auto trigger the index idx_2(TBD)

Trigger the index idx_1, but found too many records return, auto bypass index and fall back to full table scan

Spinach Implementation

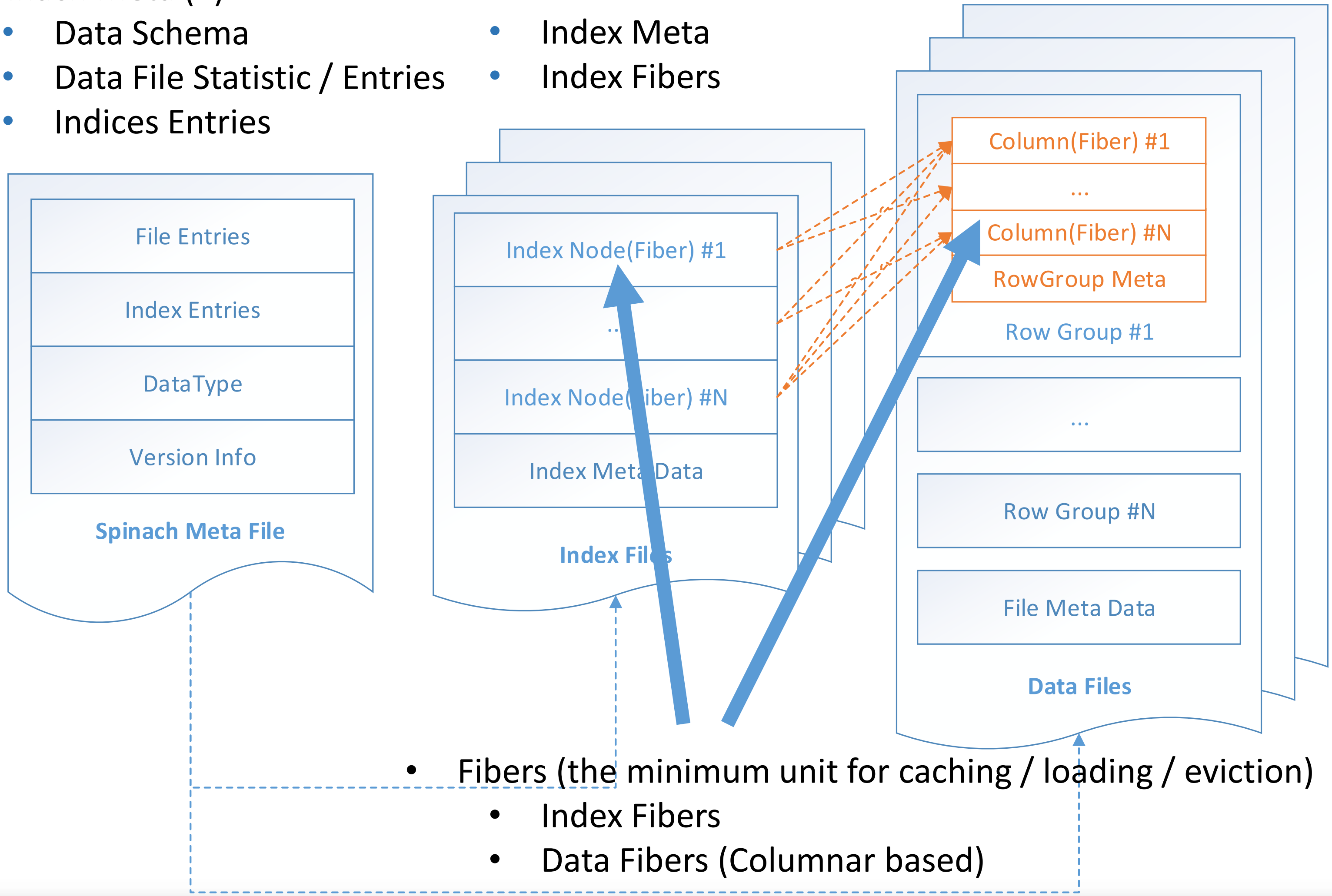
- **DDL Statement Extension (Index Management)**
  - Create / Add Index (Parser & Logical Node / Physical Execution)
  - Drop Index (Parser & Logical Node / Physical Execution)
- **Data Source Extension**
  - Implements the HadoopFSRelation interface (Support Partition & File Status Caching)
  - Abbr. (“spn”) for Spinach Data Source in Data Frame API

```
df.write.format(“spn”).save(“/path/to/spinach_test”)
sqlContext.read.format(“spn”).load(“/path/to/spinach_test”)
```

- **Enable the extensions**
  - SpinachContext (SQLContext)
  - Make SQLContext configurable in ThriftServer / SparkSQL Shell / Spark-shell
  - Spark Executor HeartBeat extenstions (talk later)

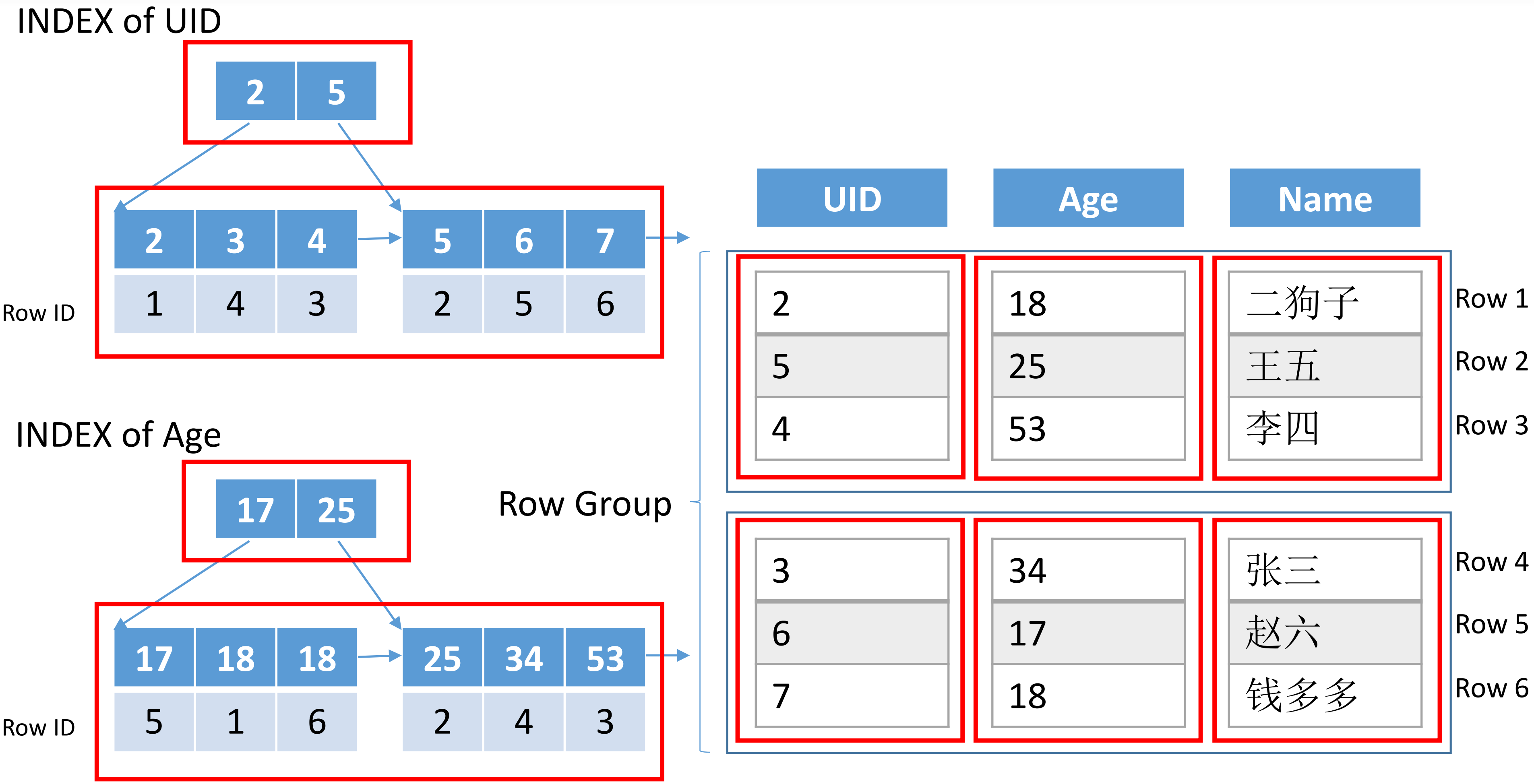
Spinach Data Format

- Spinach Meta (1)
  - Data Schema
  - Data File Statistic / Entries
  - Indices Entries
- Index File (N * M Files)
  - N is the number of Data Files
  - M is the number of Indices
  - Index Meta
  - Index Fibers
- Data File (N Files)
  - RowGroups
  - Fibers in Each Row Group
  - File Meta



- Fibers (the minimum unit for caching / loading / eviction)
  - Index Fibers
  - Data Fibers (Columnar based)

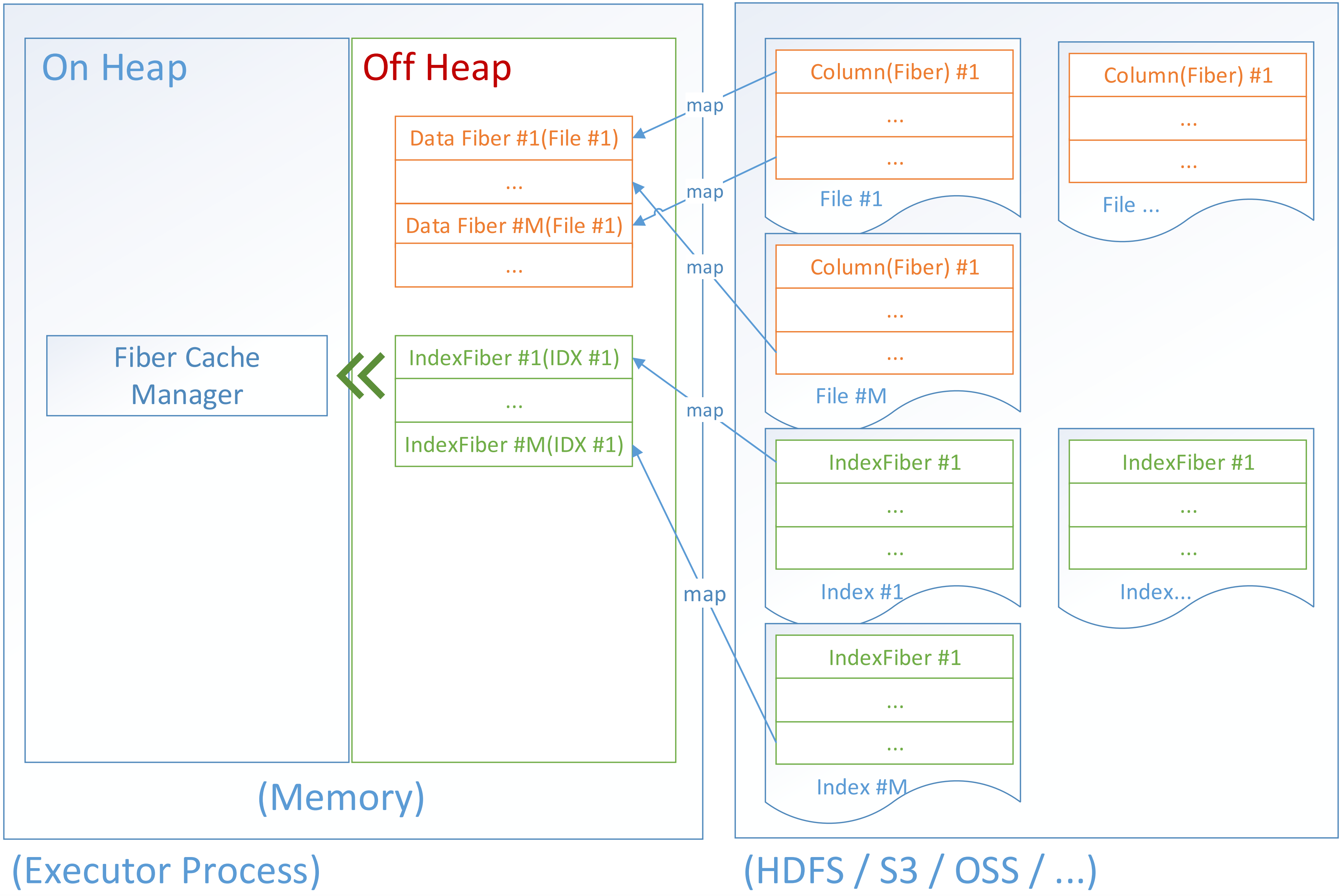
Fiber & Index



UID	Age	Name	
2	18	二狗子	Row 1
5	25	王五	Row 2
4	53	李四	Row 3
3	34	张三	Row 4
6	17	赵六	Row 5
7	18	钱多多	Row 6

Fibers are in the red boxes

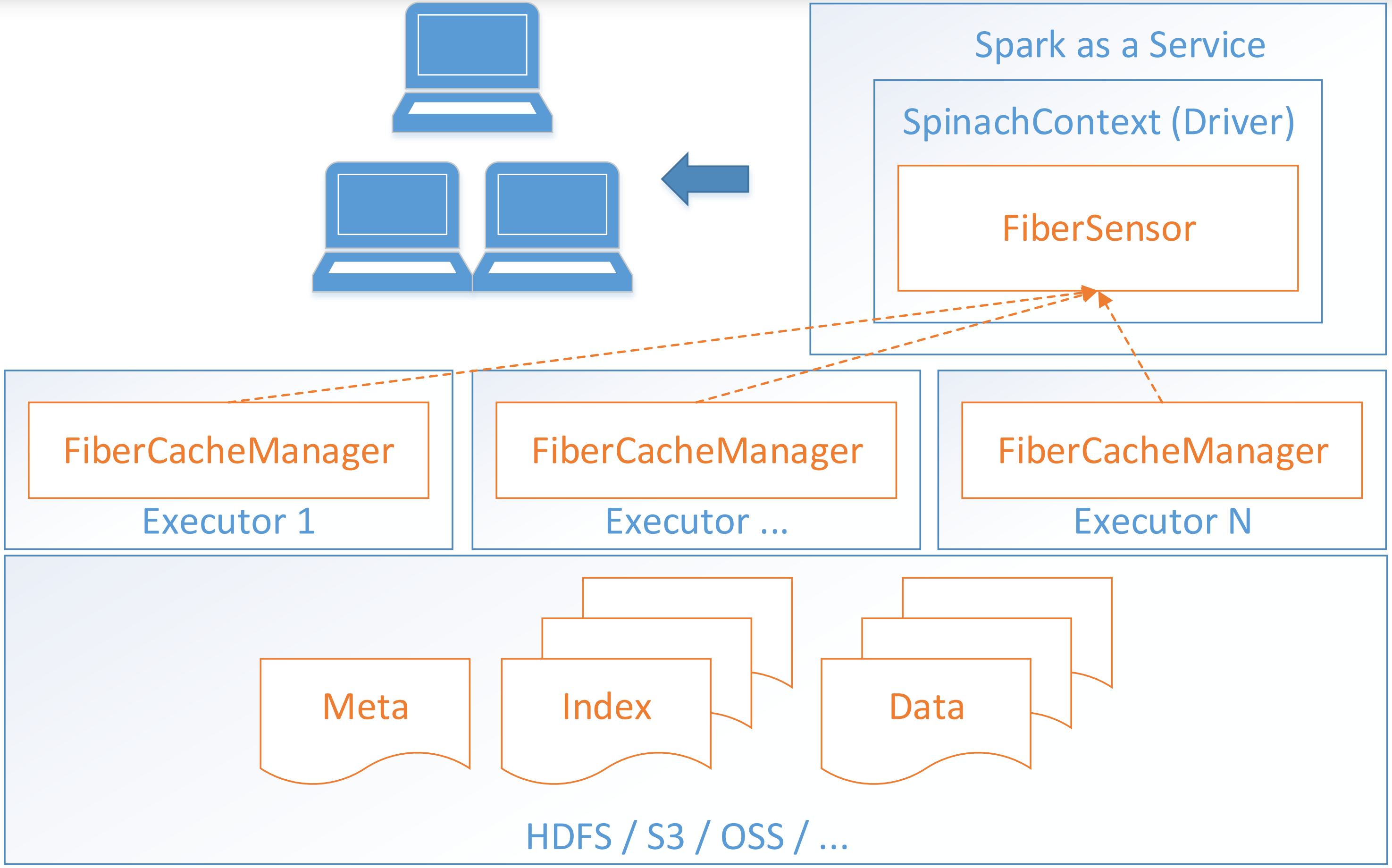
Executor Memory Layout (Query Snapshot)



File Format

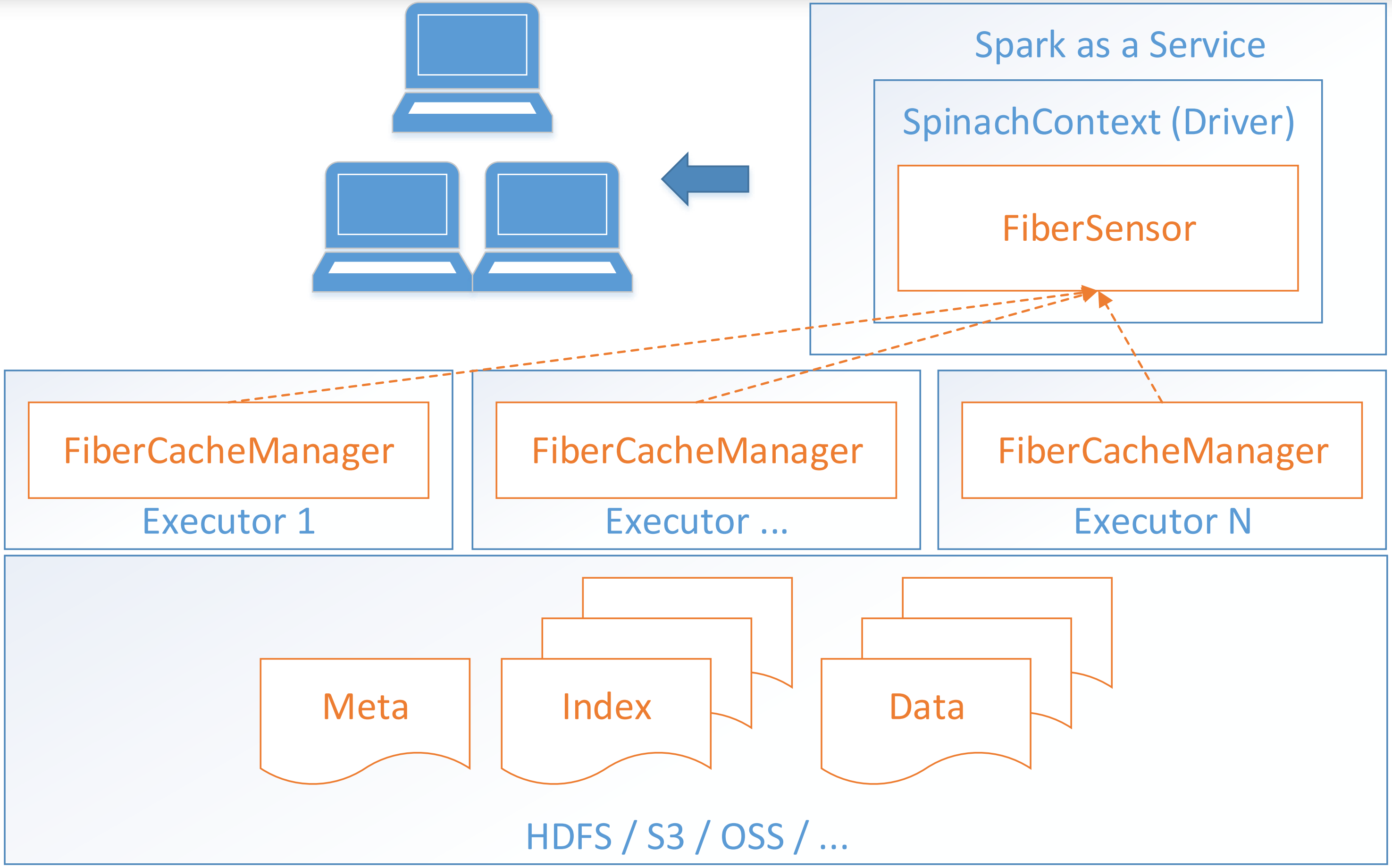
- Spinach Meta
  - Describe the data schema and statistic info
  - Describe how the indices are organized
  - Different cache strategy for fast accessing
- Data Fiber
  - Columnar Storage
  - Aim to fully compatible with Spark SQL Data Types (nested data types are TBD)
  - Vectorization friendly in the offheap Memory, without any encoding
  - Data Type aware encoder/decoder in the storage layer (TBD)
  - Decouple with concrete data format to support more columnar storage based format like ORC, Parquet (TBD)
- Index Fiber (Sort based)
  - Row(index keys) based Storage
  - MySQL like B+ Tree Implementation
  - Separate the files for index & data, for better managing the indices efficiently, and decouple with the data format.

Architecture Overview



- Fiber Cache Manager
  - Resides in each executor process
  - Manage an off-heap memory pool & Data Loading & Evicting Strategy
  - Update the fiber cache statistic info periodically with Fiber Sensor via executor heartbeat RPC.

Architecture Overview



- Fiber Sensor
  - Resides in Spark Driver Process
  - Global Fiber Cache Distribution statistic info
  - Fiber(index/data) cache-aware for preferred data location in tasks assignment

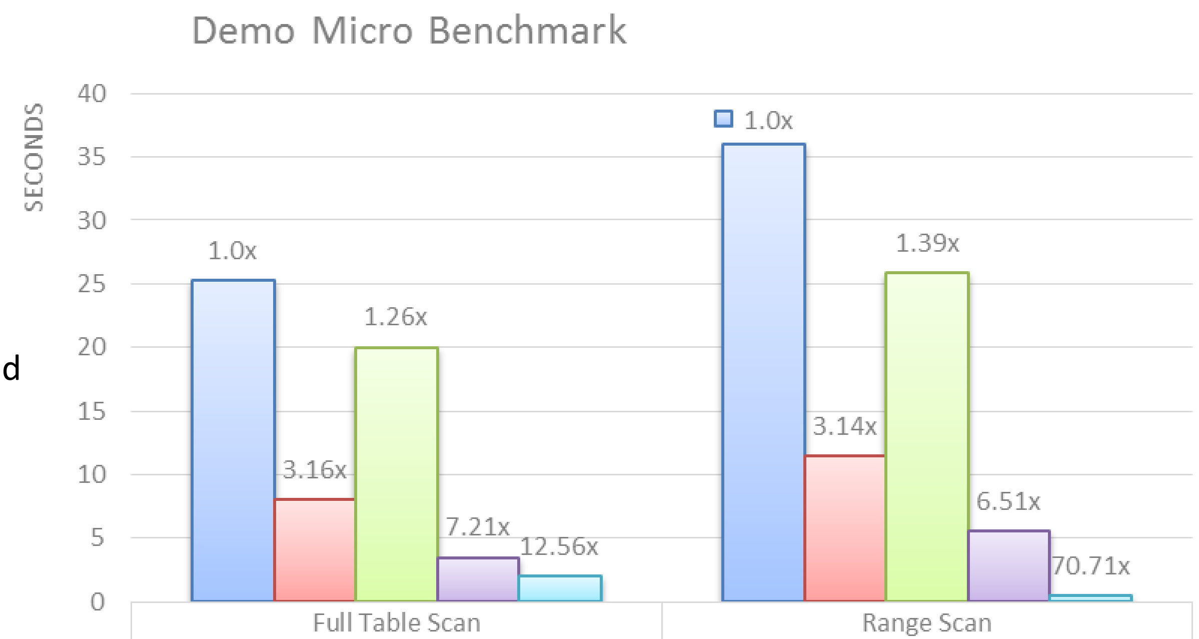
Demo Micro-benchmark

Hardware

- 1 Master + 3 Slaves
- Xeon E5-2699 v3
- 256GB / node
- 8x1TB SATA / node
- 10Gb network

Data

- ~100GB Uncompressed
- Record Count: 1.3B



	Full Table Scan	Range Scan
🟦 Parquet(Compressed Cold Data)	25.25	36.06
🟥 Parquet(Compressed) (OS Cached)	8	11.5
🟩 Parquet(NonCompression) (OS Cached)	20	25.9
🟪 Spark InMemory Cache*	3.5	5.54
🟦 Spinach	2.01	0.51

- Demo Micro-benchmark probably very different when data value patterns are different*
- Full table scan:
  - df.selectExpr("count(str1)", "count(int1)", "count(str2)").show
- Range Key Scan: ()
  - df.filter("str2 >= 'China-6234567' and str2 <= 'China-6234596'").selectExpr("count(str1)", "sum(int1)").show

Summary

- Data Source Extension
- Fined-grained Data Cache
- User Defined Indices
- Vectorization Friendly
- Off-heap Memory

Future Plan

- Open Source (POC stage)
- More Index type (e.g. bloom filter) & better encoder/decoder
- Nested data type
- Support other columnar storage based data formats (ORC/Parquet)
- More Flexible Fiber Caching & Evicting strategy
- Optimize task assignment algorithm(preferred location)
- Auto-detect & cache the shared common sub queries result

Your Idea Matters!!!

Thanks