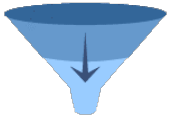
Apache CarbonData: An indexed columnar file format for interactive query with Spark SQL

Jihong MA, Jacky LI
HUAWEI

Report & Dashboard

OLAP & Ad-hoc



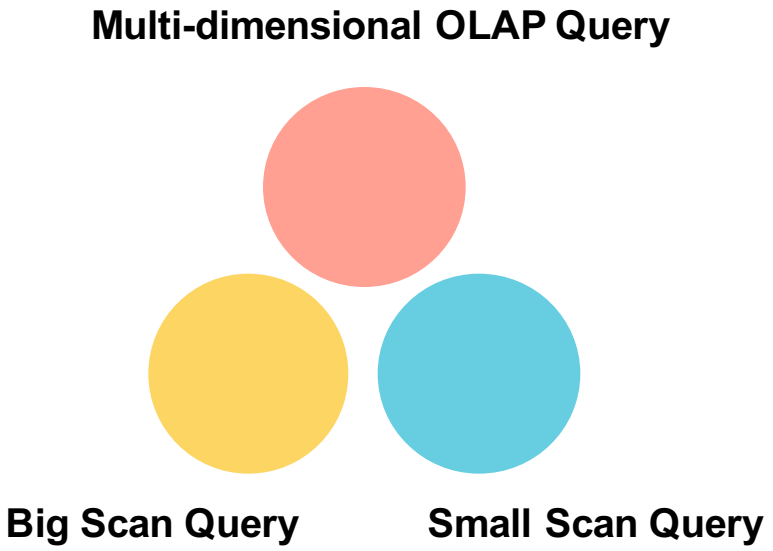
Batch processing

Real Time Analytics

Machine learning

Challenge

- Wide Spectrum of Query Analysis
  - OLAP Vs Detailed Query
  - Full scan Vs Small scan
  - Point queries



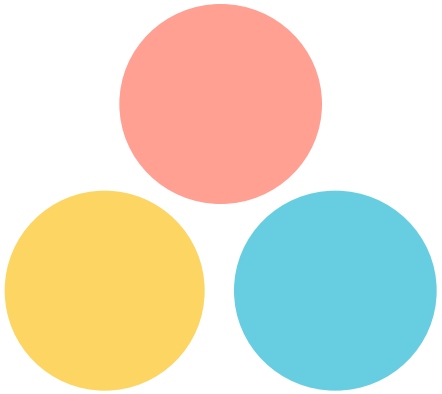
How to choose storage engine to facilitate query execution?

Available Options

1. NoSQL Database
   - Key-Value store: low latency, <5ms
   - No Standard SQL support
2. MPP relational Database
   - Shared-nothing enables fast query execution
   - Poor scalability: < 100 cluster size, no fault-tolerance
3. Search Engine
   - Advanced indexing technique for fast search
   - 3~4X data expansion in size, no SQL support
4. SQL on Hadoop
   - Modern distributed architecture and high scalability
   - Slow on point queries

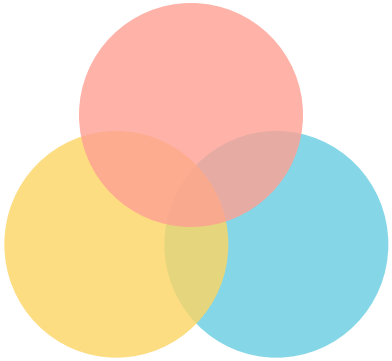
Motivation for A New File Format

Multi-dimensional OLAP Query



Big Scan Query **Small Scan Query**

CarbonData: Unified File Format



A single copy of data balanced to fit all data access

Apache CarbonData

- Apache Incubator Project since June, 2016
- Apache releases
  - 4 stable releases
  - **Latest 1.0.0, Jan 28, 2017**
- Contributors:

- In Production:

Compute

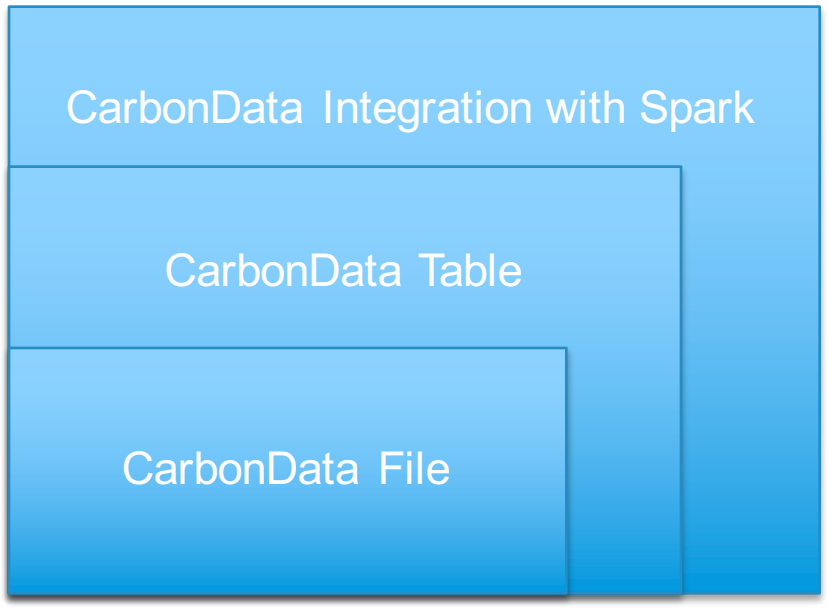
Storage

Introducing CarbonData

3 What it takes to deeply integrate with distributed processing engine like Spark?

2 What forms a CarbonData table on disk?

1 What is CarbonData file format?



CarbonData:

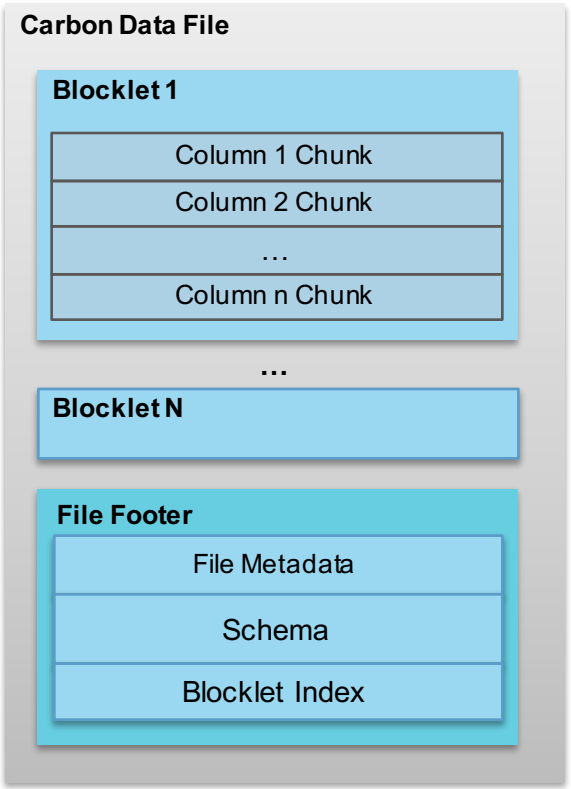
An Indexed Columnar File Format

CarbonData File Structure

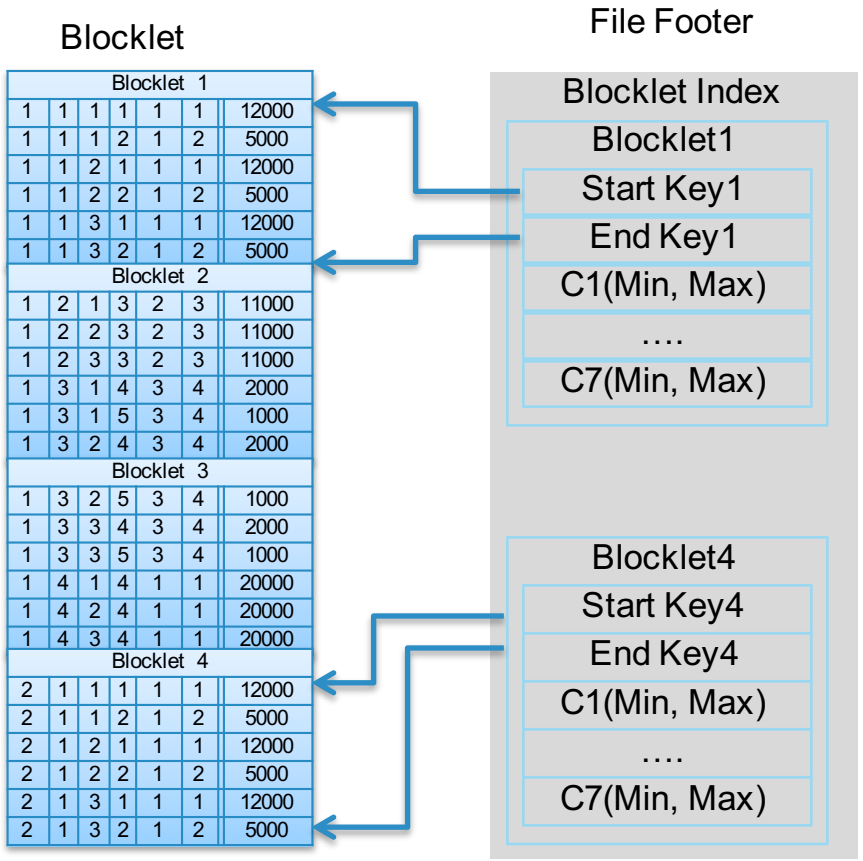
- **Built-in Columnar & Index**
  - Multi-dimensional Index (B+ Tree)
  - Min/Max index
  - Inverted index
- **Encoding:**
  - RLE, Delta, Global Dictionary
  - Snappy for compression
  - Adaptive Data Type Compression
- **Data Type:**
  - Primitive type and nested type

CarbonData File Layout

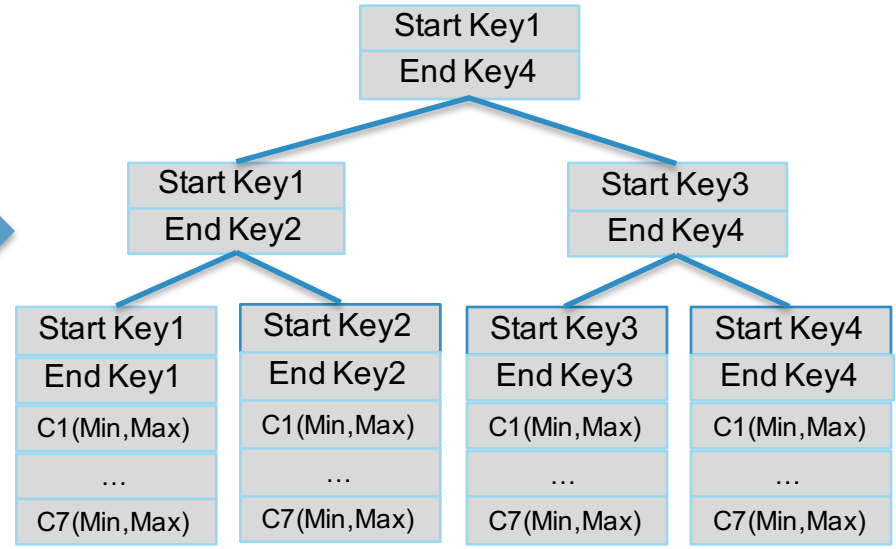
- Blocklet: A set of rows in columnar format
  - Data are sorted along MDK (multi-dimensional keys)
  - Clustered data enabling efficient filtering and scan
- Column chunk: Data for one column in a Blocklet
- Footer: Metadata information
  - File level metadata & statistics
  - Schema
  - Blocklet Index



File Level Blocklet Index



- Build in-memory file level MDK index tree for filtering
- Major optimization for efficient scan

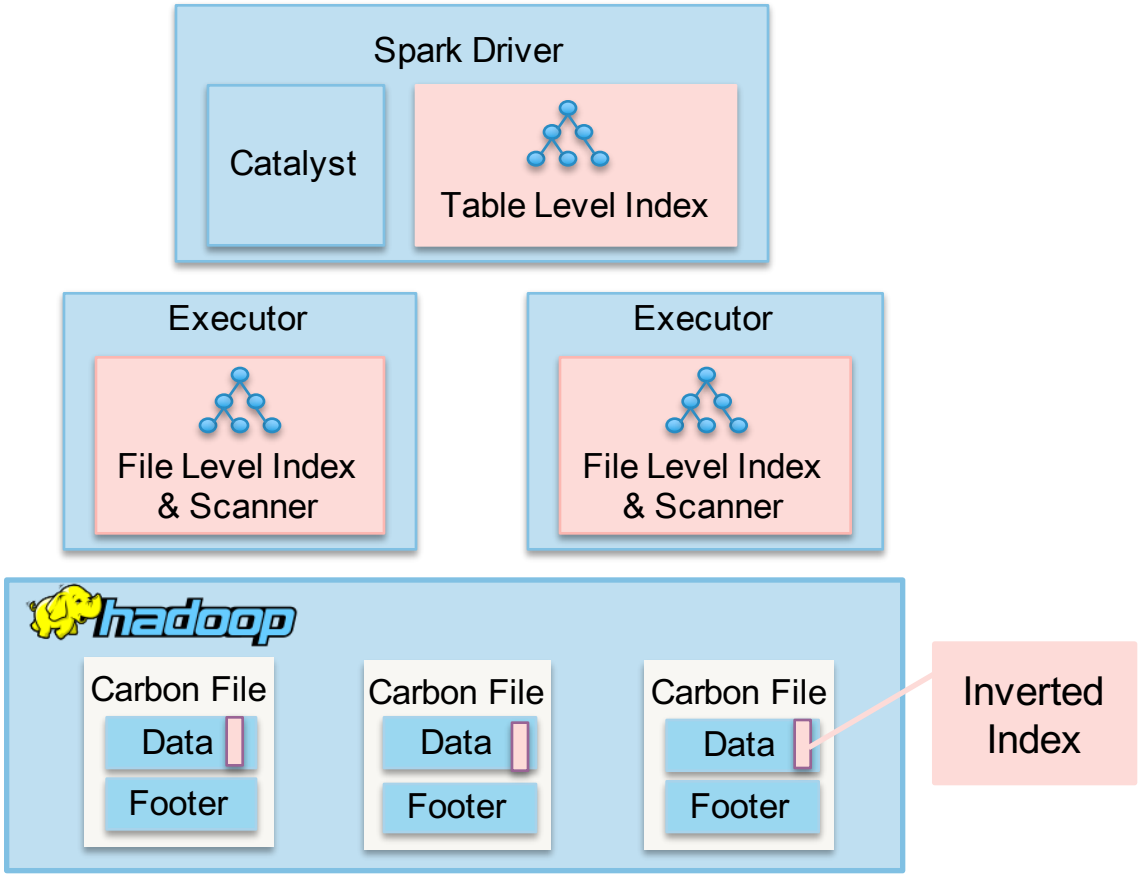


Rich Multi-Level Index Support

•Using the index info in footer, two level B+ tree index can be built:

- •File level index: local B+ tree, efficient blocklet level filtering
- •Table level index: global B+ tree, efficient file level filtering

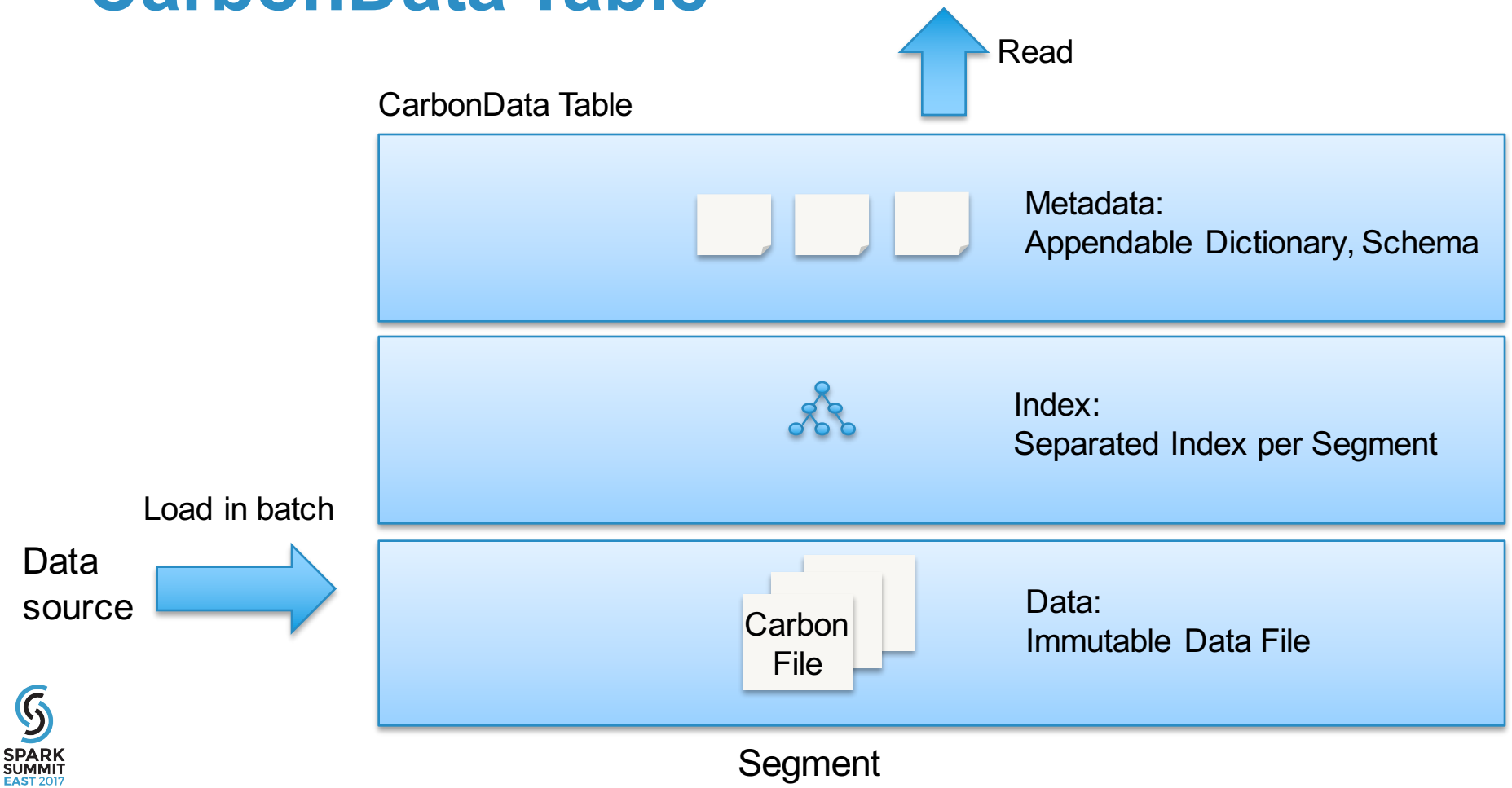
•Column chunk inverted index: efficient column chunk scan



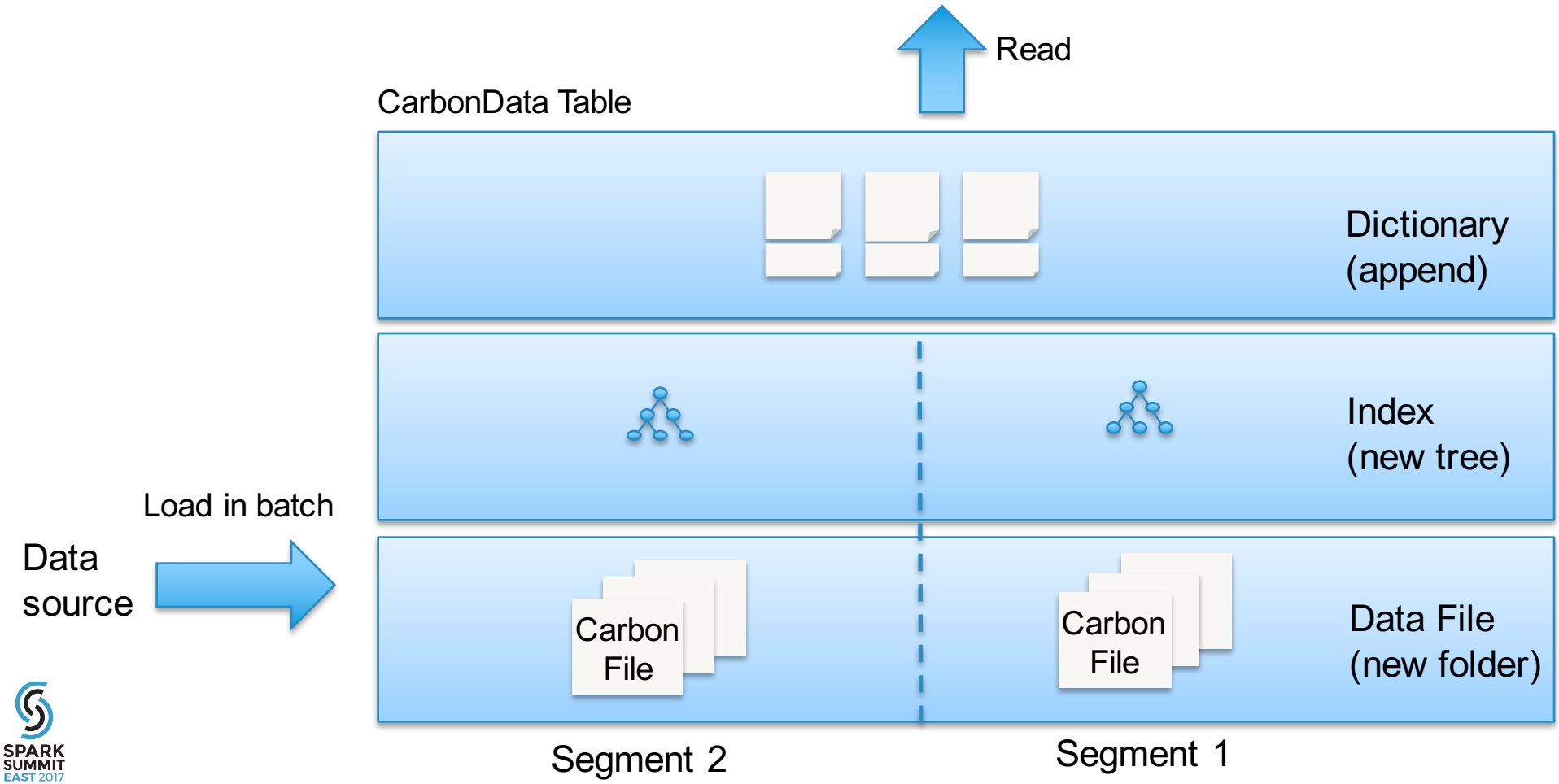
CarbonData Table:

Data Segment + Metadata

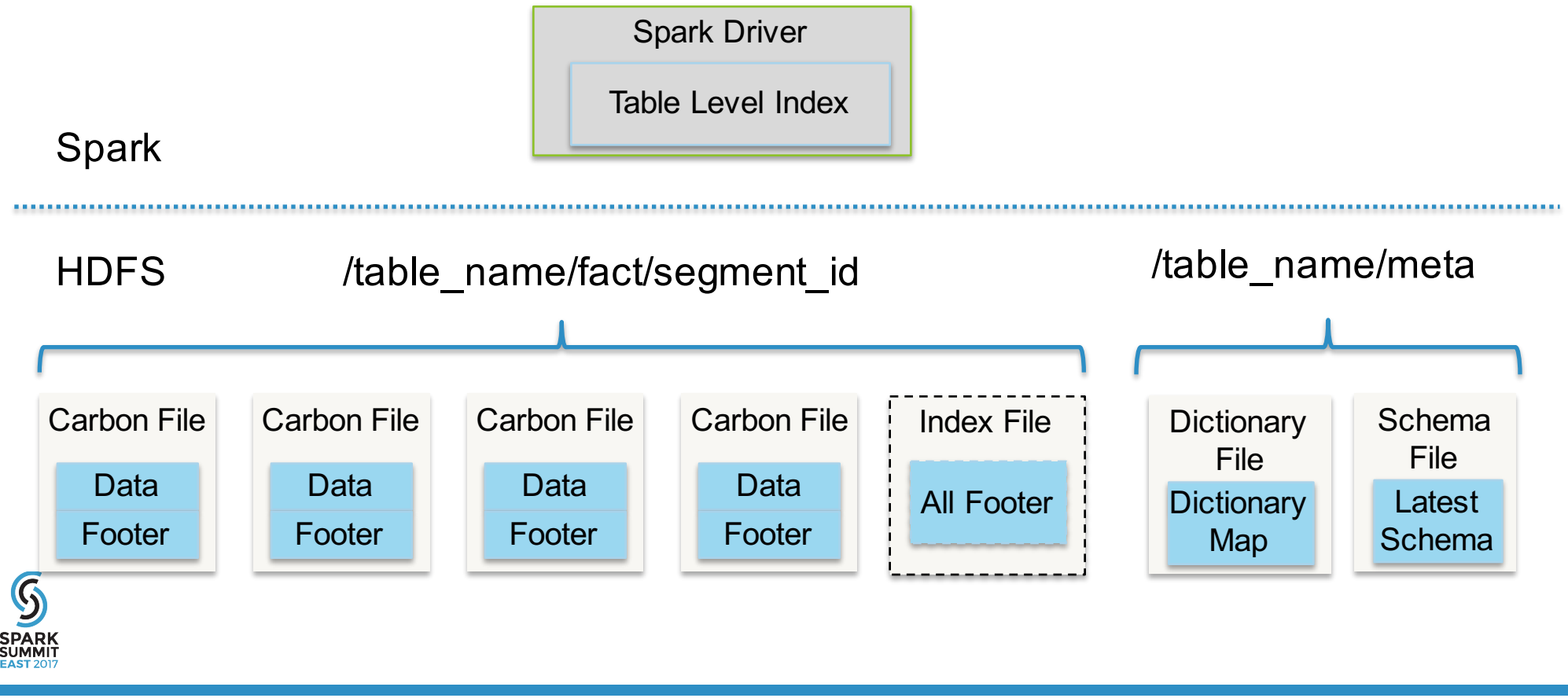
CarbonData Table



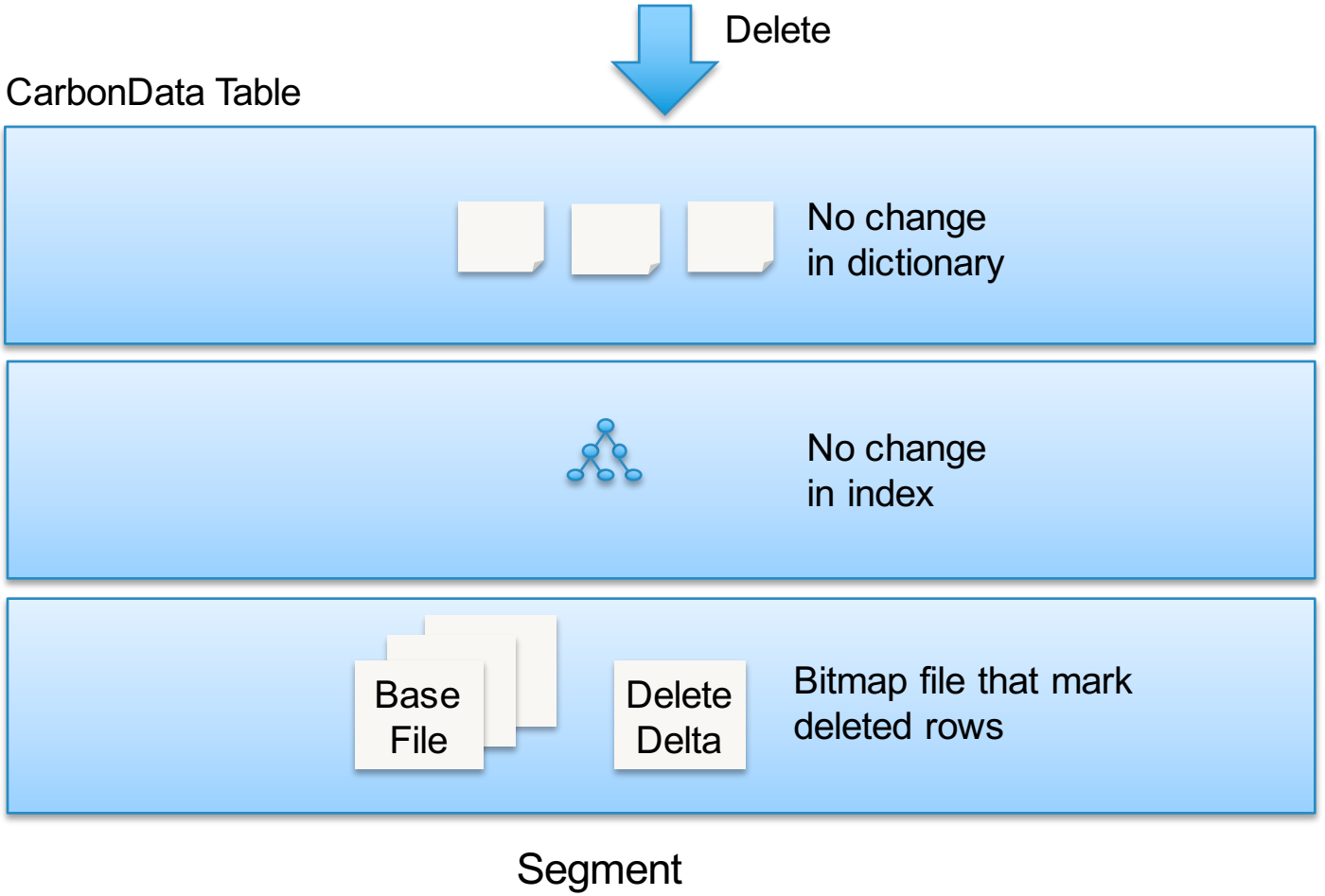
CarbonData Table



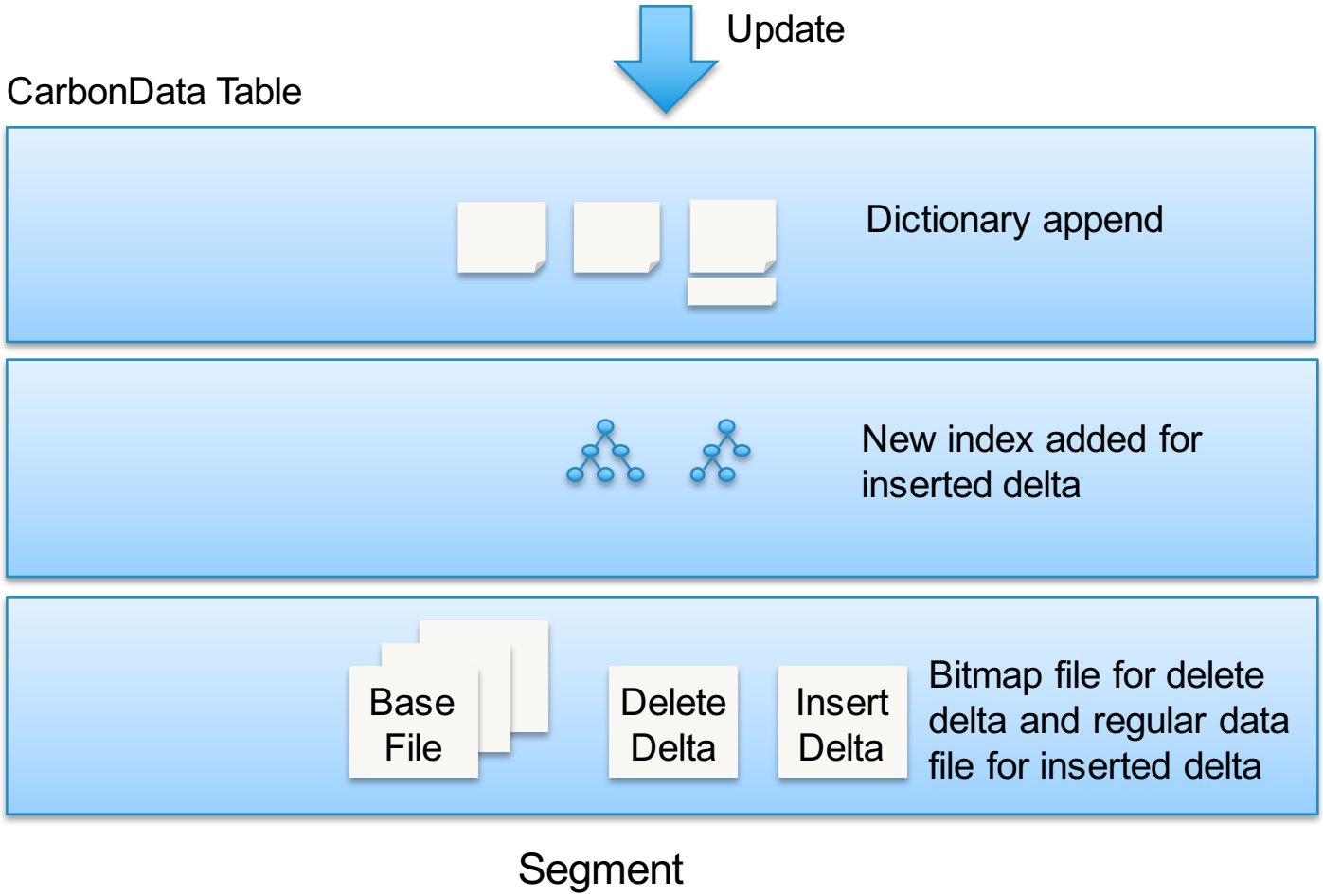
CarbonData Table Layout



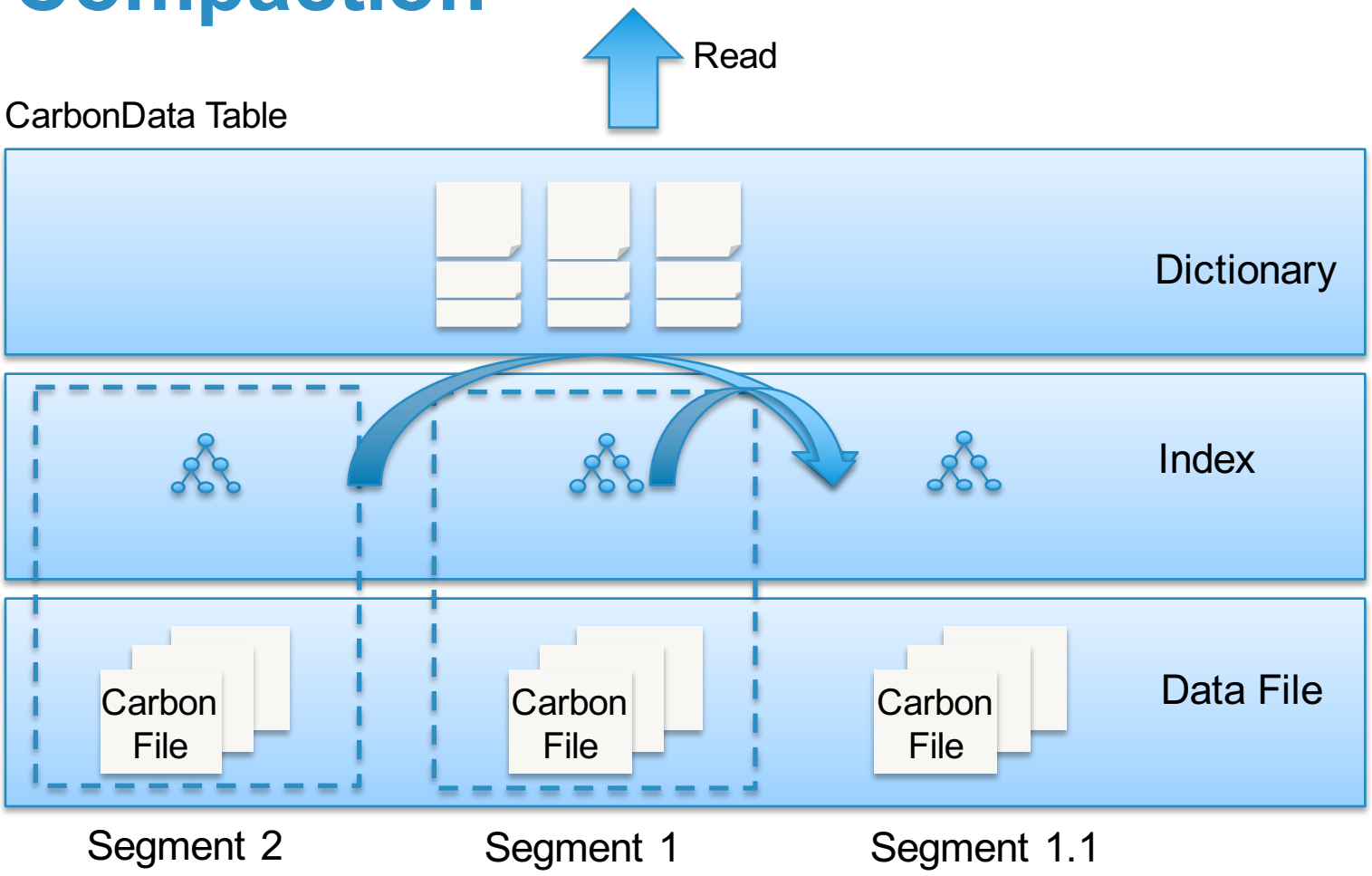
Mutation: Delete



Mutation: Update

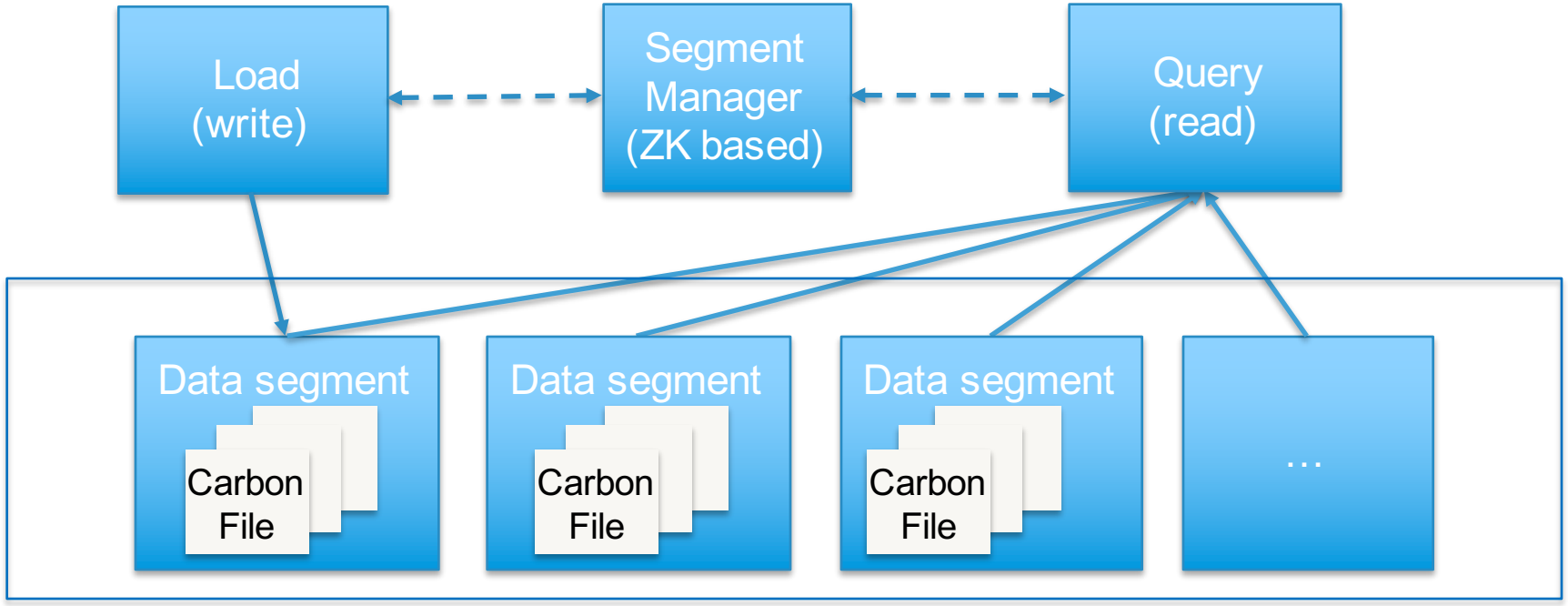


Data Compaction



Segment Management

Leveraging ZooKeeper to manage the Segment State

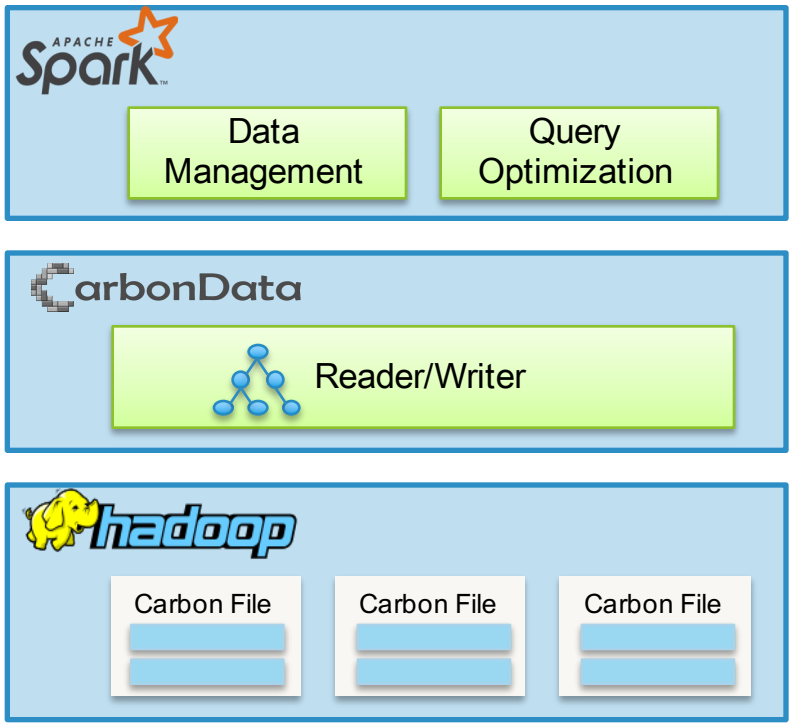


SparkSQL + CarbonData:

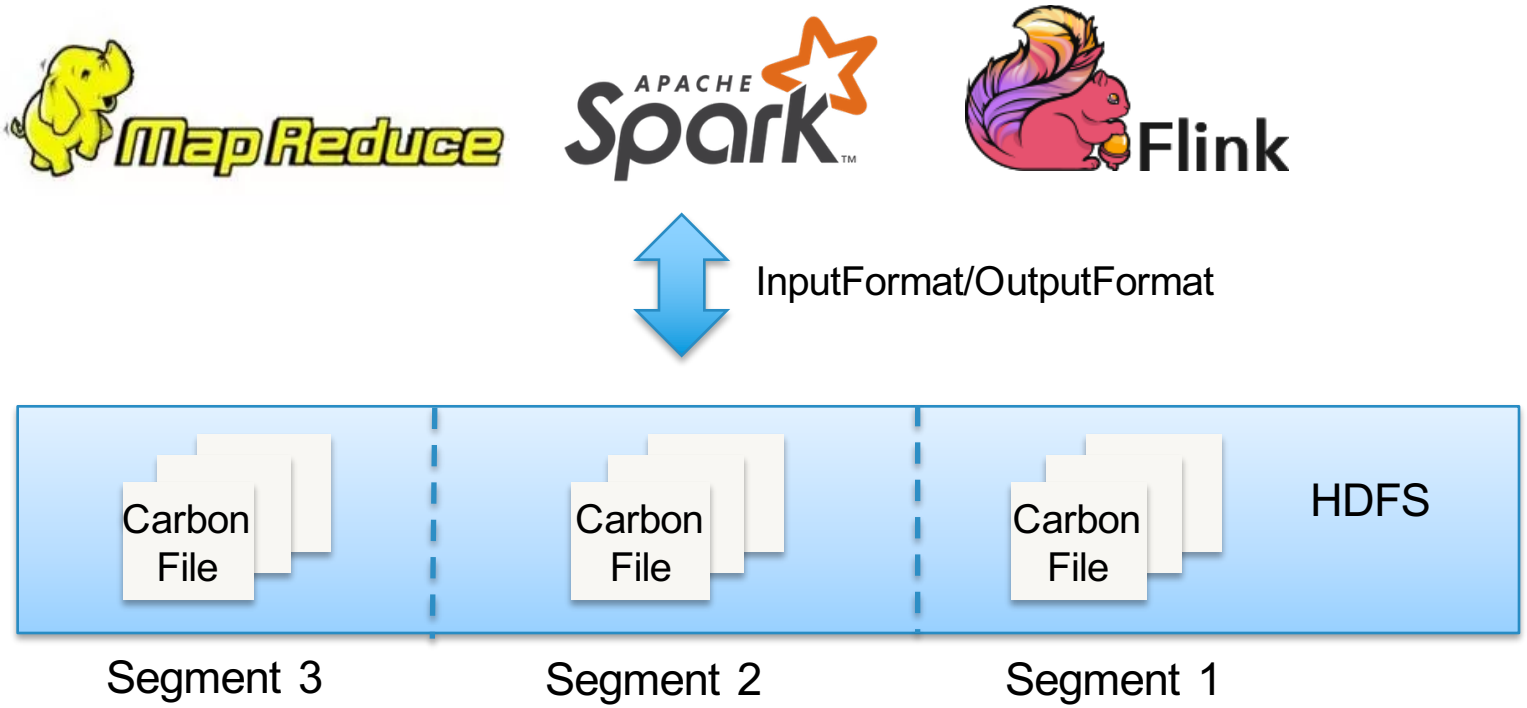
Enables fast interactive data analysis

Carbon-Spark Integration

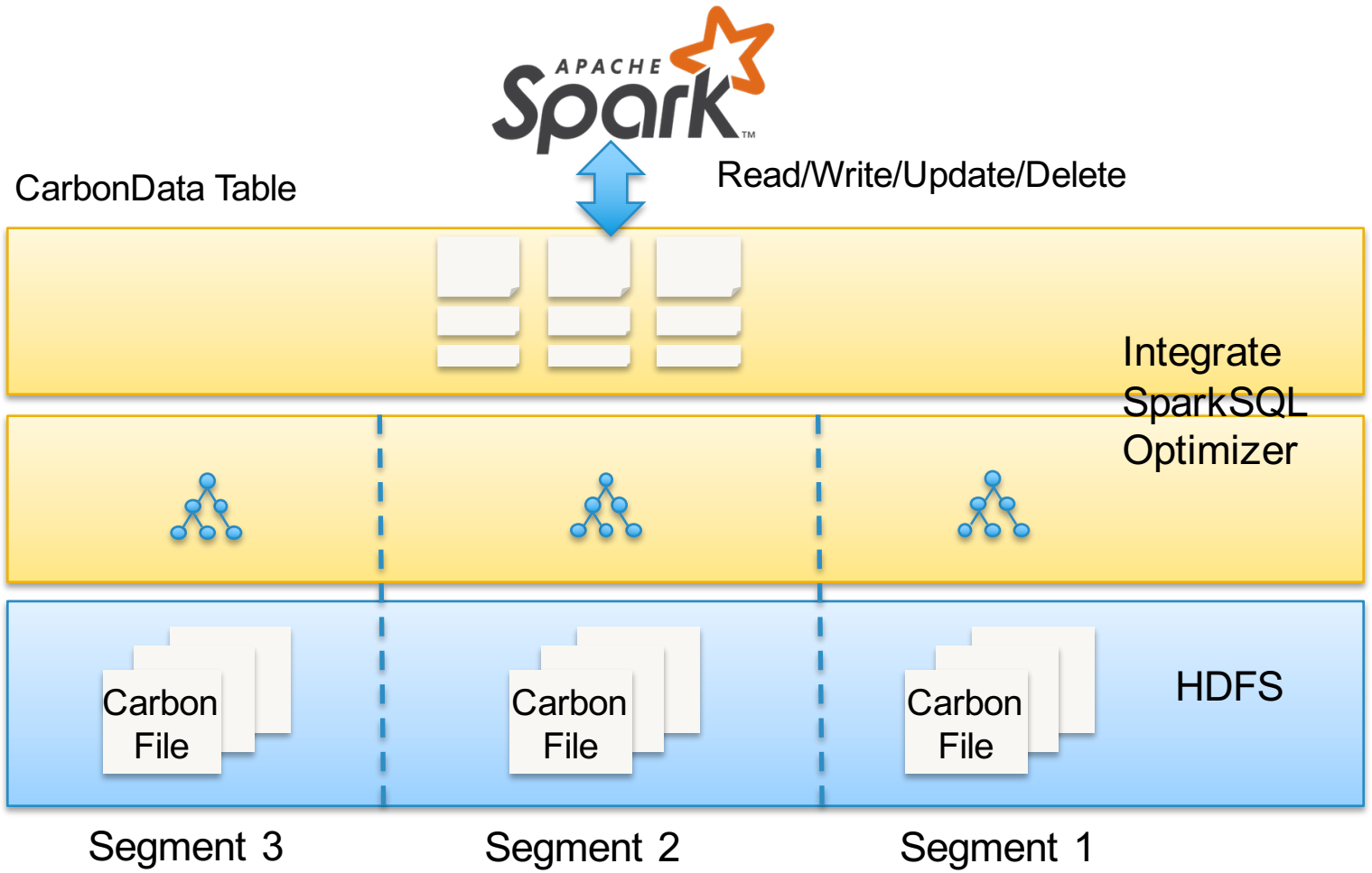
- Built-in Spark integration
  - Spark 1.5, 1.6, 2.1
- Interface
  - SQL
  - DataFrame API
- Operation:
  - Load, Query (with optimization)
  - Update, Delete, Compaction, etc



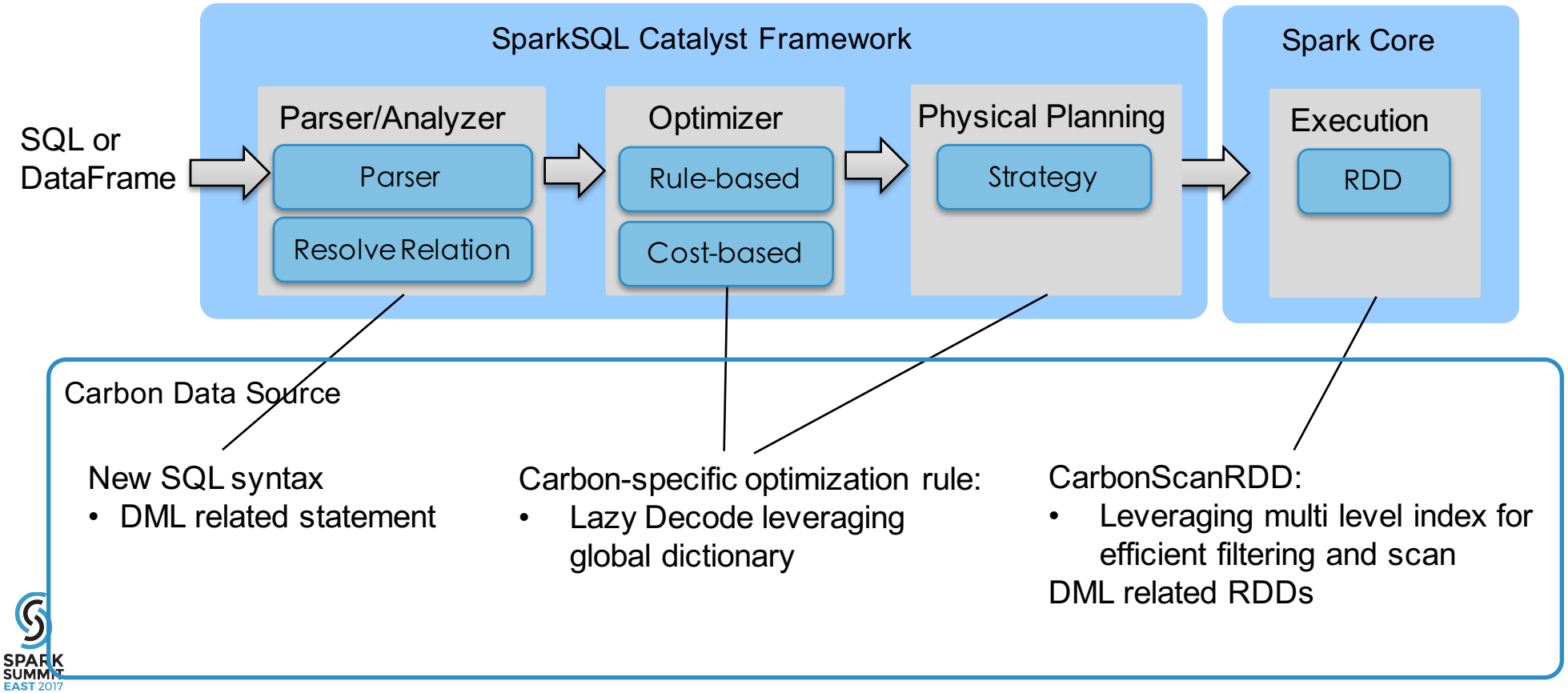
Integration through File Format



Deep Integration with SparkSQL

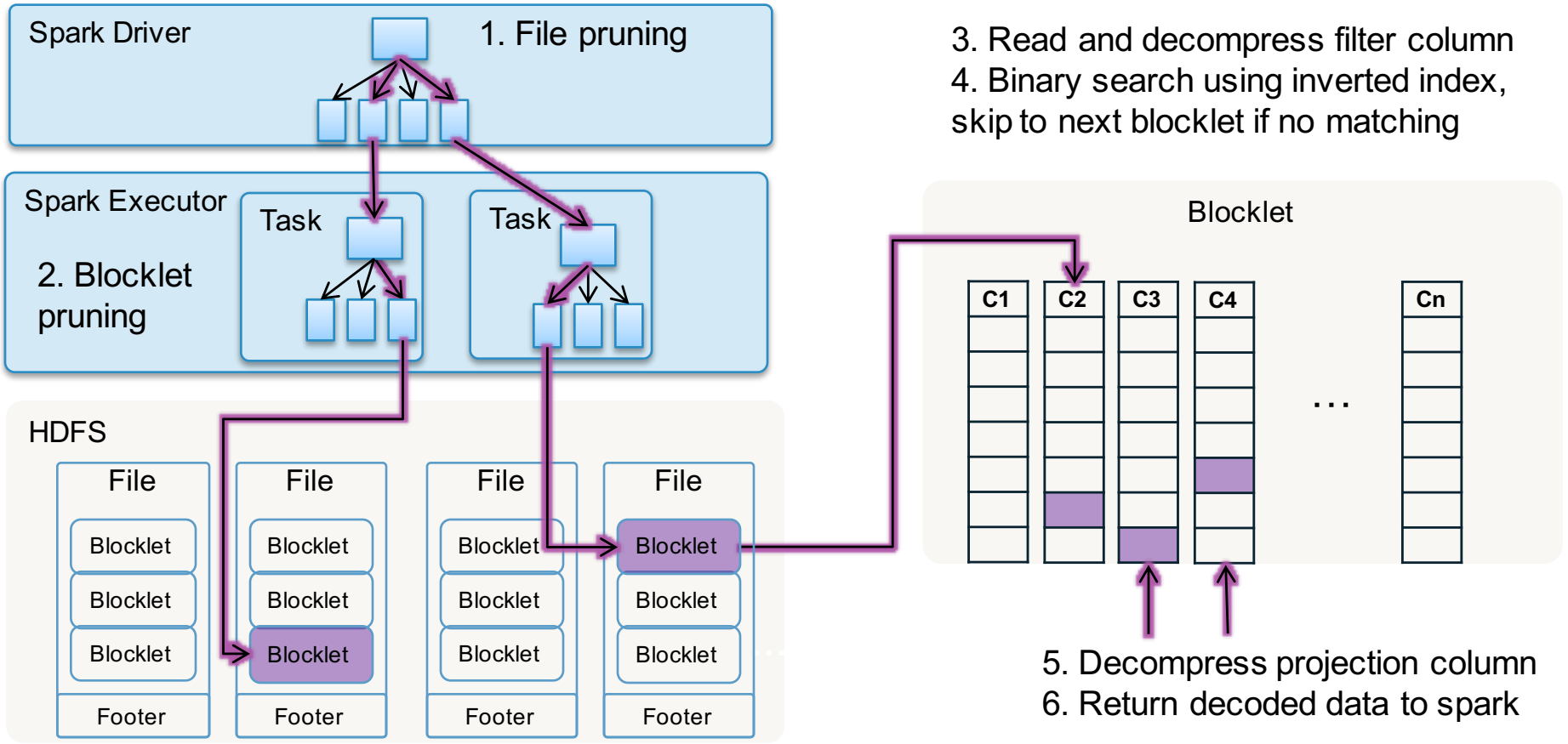


CarbonData as a SparkSQL Data Source



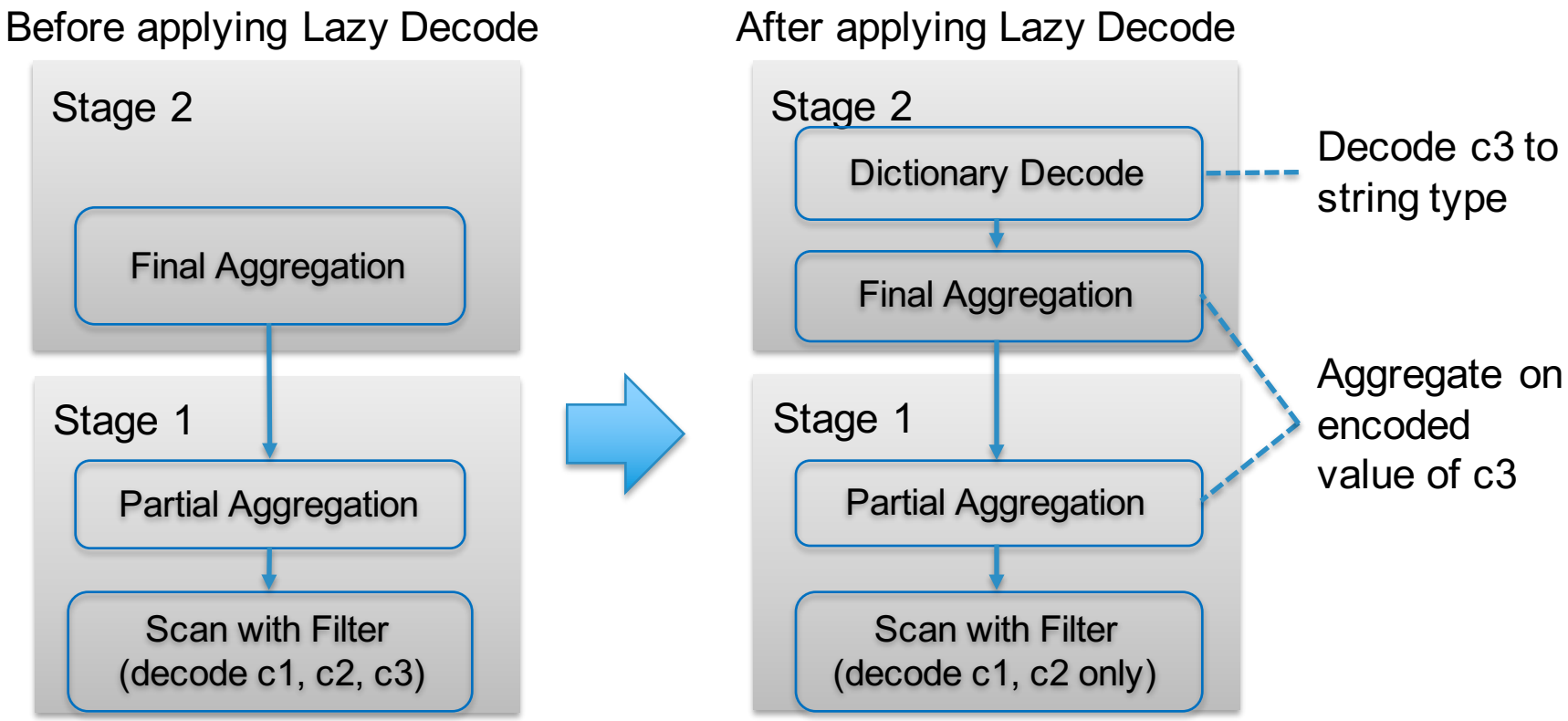
Efficient Filtering via Index

```
SELECT c3, c4 FROM t1 WHERE c2='boston'
```



Lazy Decoding by Leveraging Global Dictionary

```
SELECT c3, sum(c2) FROM t1 WHERE c1>10 GROUP BY c3
```



Usage: Write

- Using SQL

```
CREATE TABLE tablename (name String, PhoneNumber String) STORED BY “carbondata”

LOAD DATA [LOCAL] INPATH 'folder path' [OVERWRITE] INTO TABLE tablename
OPTIONS(property_name=property_value, ...)

INSERT INTO TABLE tablennme  select_statement1 FROM table1;
```

- Using Dataframe

```
df.write
    .format(“carbondata")
    .options("tableName“, “t1"))
    .mode(SaveMode.Overwrite)
    .save()
```

Usage: Read

- Using SQL

```
SELECT project_list FROM t1
WHERE cond_list
GROUP BY columns
ORDER BY columns
```

- Using Dataframe

```
df = sparkSession.read
          .format("carbondata")
          .option("tableName", "t1")
          .load("path_to_carbon_file")

df.select(…).show
```

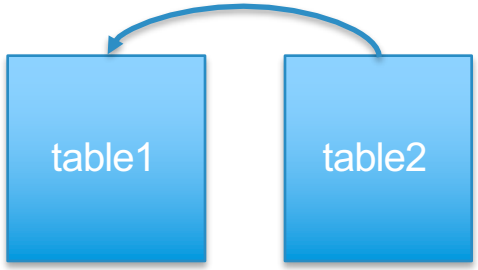
Usage: Update and Delete

Modify one column in table1

```
UPDATE table1 A
SET A.REVENUE = A.REVENUE - 10
WHERE A.PRODUCT = 'phone'
```

Modify two columns in table1 with values from table2

```
UPDATE table1 A
SET (A.PRODUCT, A.REVENUE) =
     (
      SELECT PRODUCT, REVENUE
      FROM table2 B
      WHERE B.CITY = A.CITY AND B.BROKER = A.BROKER
     )
WHERE A.DATE BETWEEN '2017-01-01' AND '2017-01-31'
```



Delete records in table1

```
DELETE FROM table1 A
WHERE A.CUSTOMERID = '123'
```

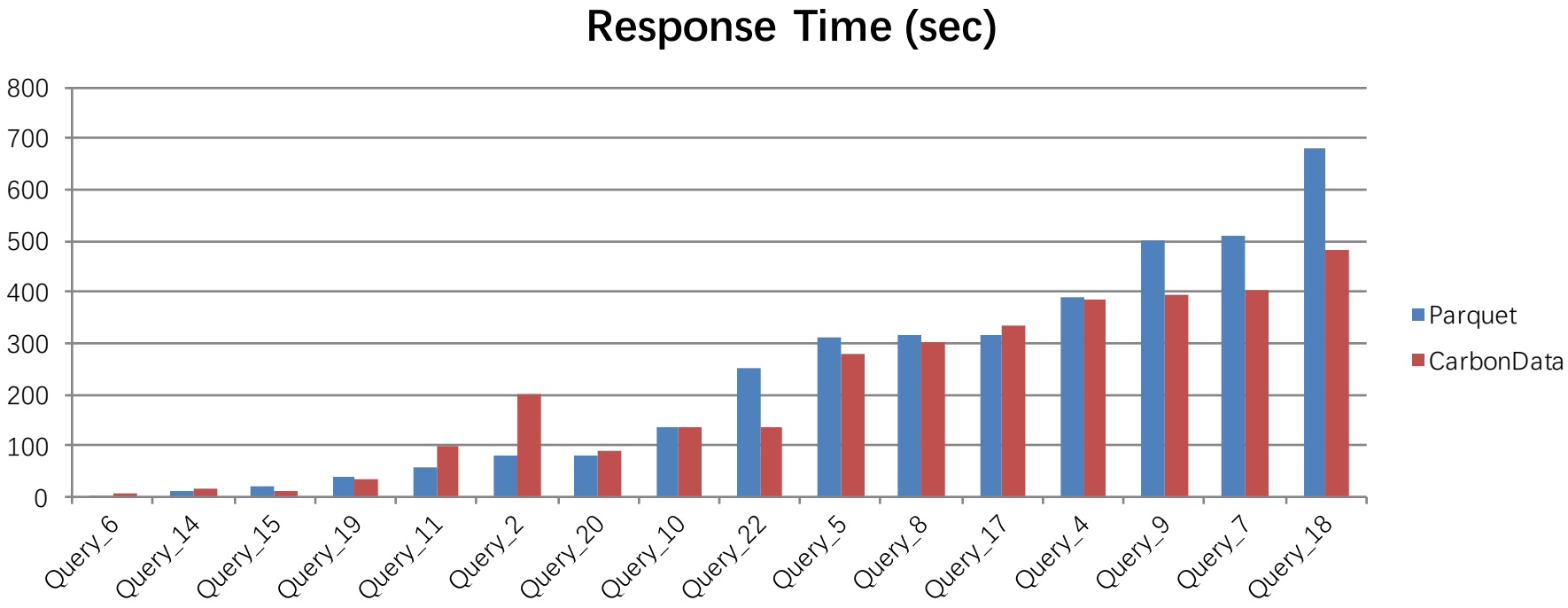
Performance Result

Test

1. TPC-H benchmark (500GB)
2. Test on Production Data Set (Billions of rows)
3. Test on Large Data Set for Scalability (1000B rows, 103TB)

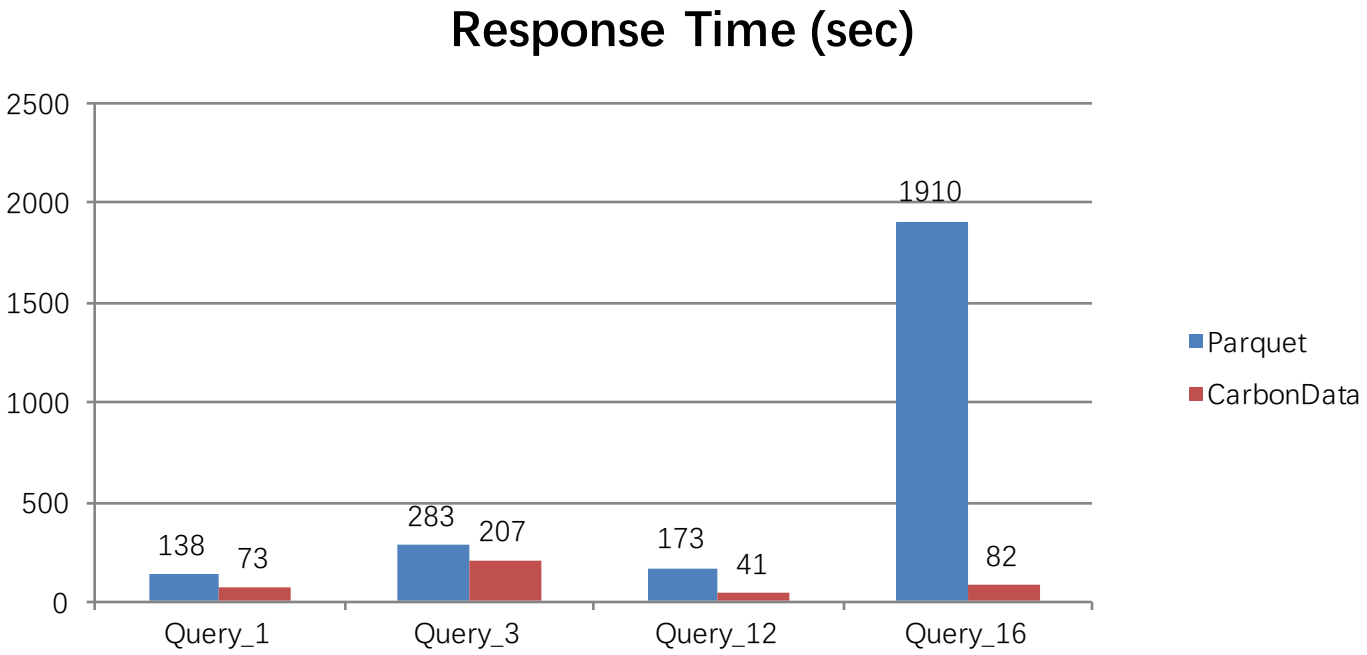
- Storage
  - Parquet:
    - Partitioned by time column (c1)
  - CarbonData:
    - Multi-dimensional Index by c1~c10
- Compute
  - Spark 2.1

TPC-H: Query



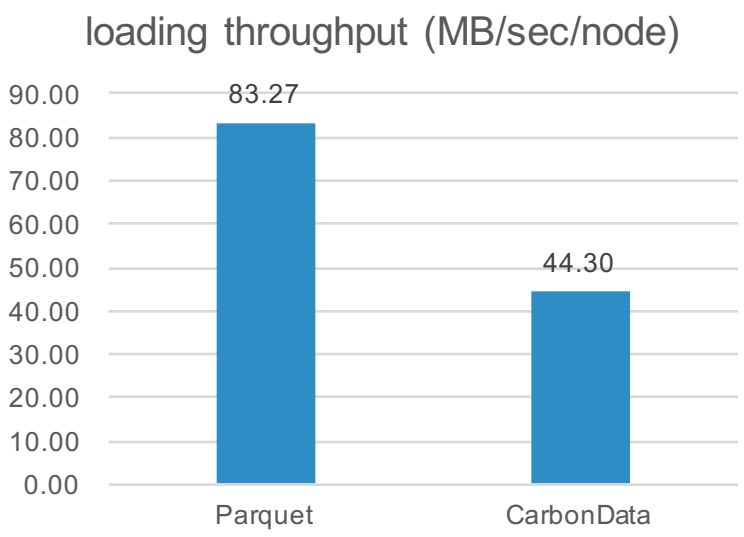
For big scan queries, similar performance, ±20%

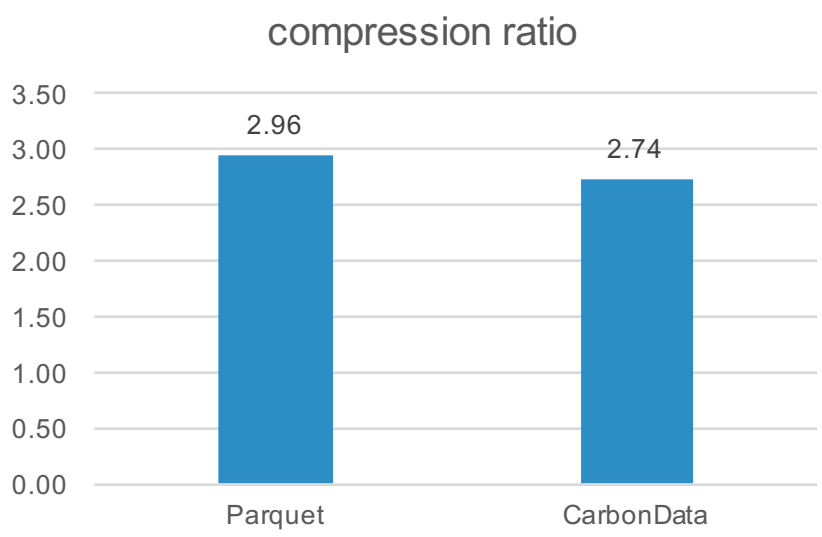
TPC-H: Query



For queries include filter on fact table,
CarbonData get 1.5-20X performance by leveraging index

TPC-H: Loading and Compression





Test on Production Data Set

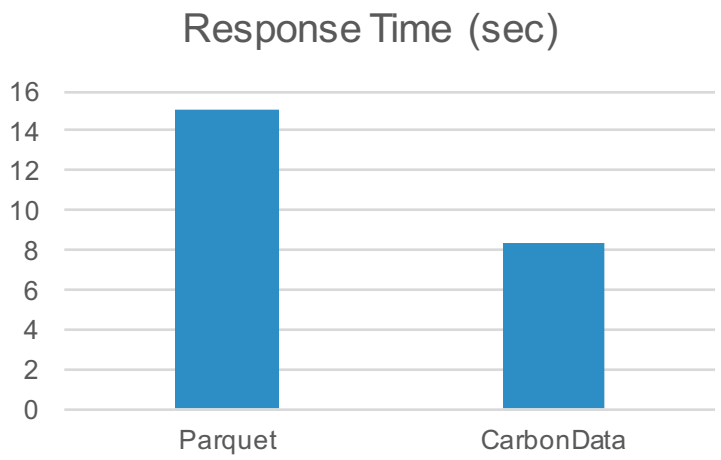
Filter Query

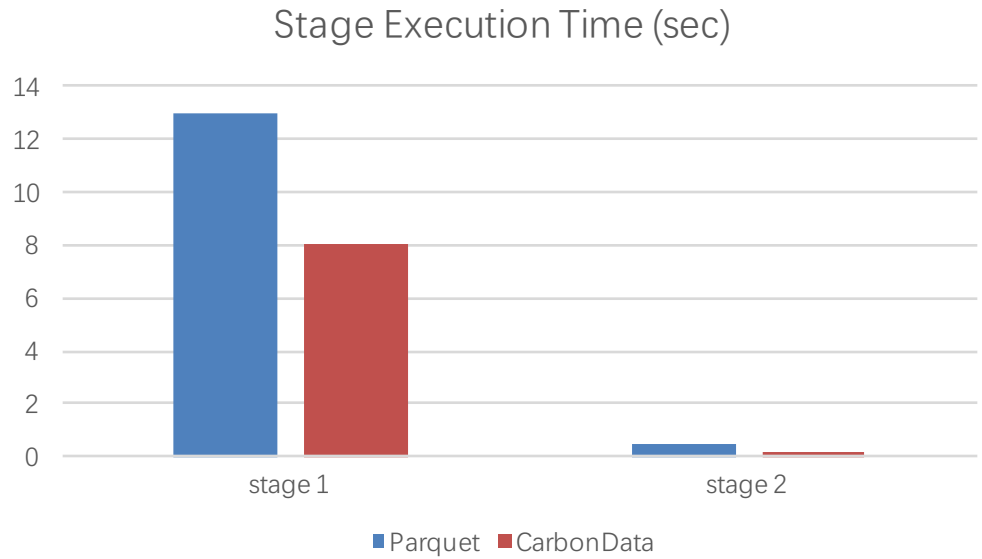
	Filter								Response Time (sec)		Number of Task	
Query	c1	c2	c3	c4	c5	c6	…	c10	Parquet	CarbonData	Parquet	CarbonData
Q1	✓	✓	✓	✓					6.4	1.3	55	5
Q2		✓	✓						65	1.3	804	5
Q3		✓		✓					71	5.2	804	9
Q5		✓			✓				64	4.7	804	9
Q4			✓	✓					67	2.7	804	161
Q6			✓			✓			62	3.7	804	161
Q7				✓		✓			63	21.85	804	588
Q8								✓	69	11.2	804	645

Observation:
Less scan task (resource) is needed because of more efficient filtering by leveraging multi-level index

Test on Production Data Set

Aggregation Query: no filter, group by c5 (dictionary encoded column)





Observation: both partial aggregation and final aggregation are faster, because aggregation operates on dictionary encoded value

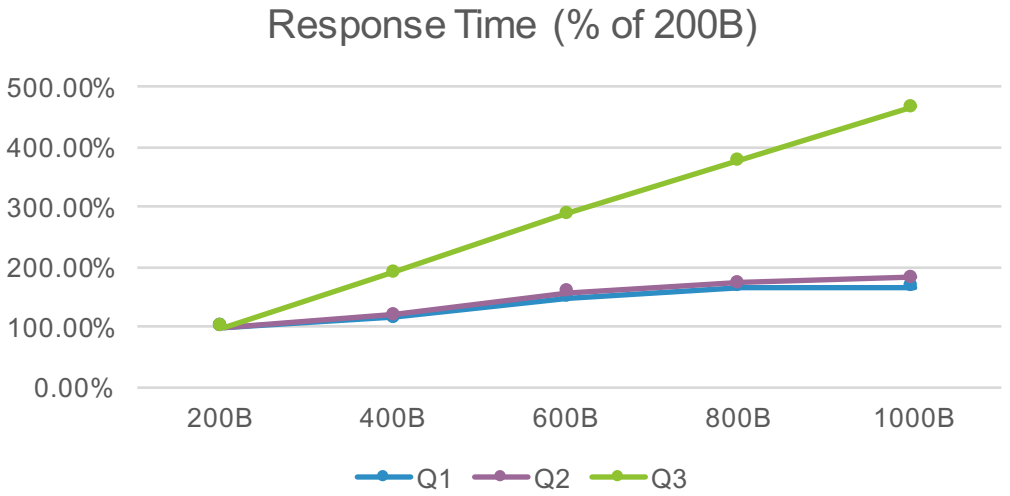
Test on large data set

Data: 200 to 1000 Billion Rows (half year of telecom data in one china province)

Cluster: 70 nodes, 1120 cores

Query:

Q1: filter (c1~c4), select *
Q2: filter (c10), select *
Q3: full scan aggregate



Observation

When data increase:

- Index is efficient to reduce response time for IO bound query: Q1, Q2
- Spark can scale linearly for CPU bound query: Q3

What's Coming Next

What's coming next

- Enhancement on data loading & compression
- Streaming Ingest:
  - Introduce row-based format for fast ingestion
  - Gradually compact row-based to column-based for analytic workload
  - Optimization for time series data
- Broader Integration across big data ecosystem: Beam, Flink, Kafka, Kylin

Apache CarbonData

- Love feedbacks, try out, any kind of contribution!
  - Code: https://github.com/apache/incubator-carbondata
  - JIRA: https://issues.apache.org/jira/browse/CARBONDATA
  - dev mailing list: dev@carbondata.incubator.apache.org
  - Website: http://http://carbondata.incubator.apache.org

  - **Current Contributor: 64**
  - **Monthly resolved JIRA issue: 100+**

Thank You.

Jihong.Ma@huawei.com

Jacky.Ilikun@huawei.com