Big Data Tools

Ira A. Fulton Schools of Engineering
Arizona State University

Resilient Distributed Dataset (RDD)

What is an RDD?

- A RDD is a read-only, partitioned collection of records.
- RDDs can only be created through operations on either (1) data in stable storage or (2) other RDDs.
- It is a restricted Distributed shared – Memory System.

RDD Contains:

- A set of partitions: Atomic pieces of the dataset
- A set of dependencies on parent RDDs (For fault tolerance)
- A function for computing the dataset based on its parents (For fault Tolerance)

Metadata about its partitioning scheme and data placement

Resilient Distributed Dataset (RDD) – cont.

Two important features:

- Fault tolerance:
  - That is achieved through lineage retrieval
- Lazy Evaluation:
  - A RDD will not be created until a reduce-like job or persist job called.

- *Zaharia, Matei, et al. "Spark: cluster computing with working sets." in Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. 2010.*
- *Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." in Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.*

Data Management in Apache Spark: RDD Abstraction

- Resilient distributed datasets
- Partitioned collection of records
- Spread across the cluster
- read-only
- Caching dataset in memory

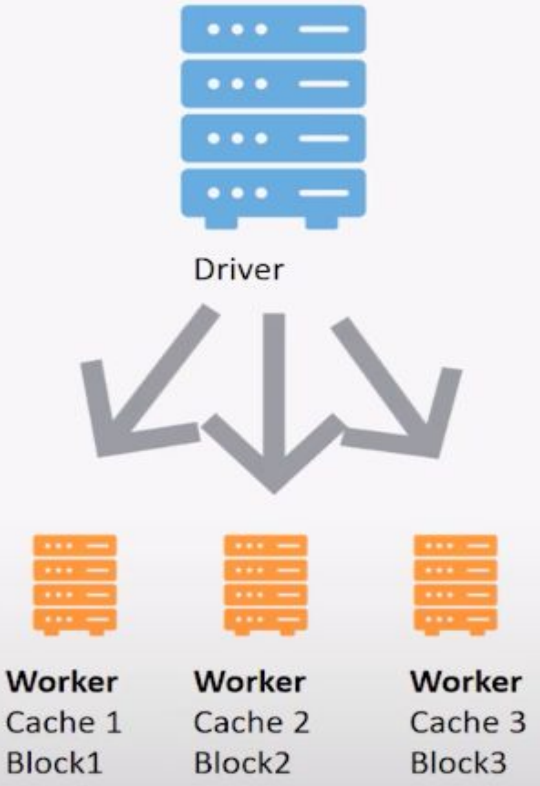
RDD Operations

- *Transformations* to build RDDs through deterministic operations on other RDDs
  - transformations include *map*, *filter*, *join*
  - lazy operation

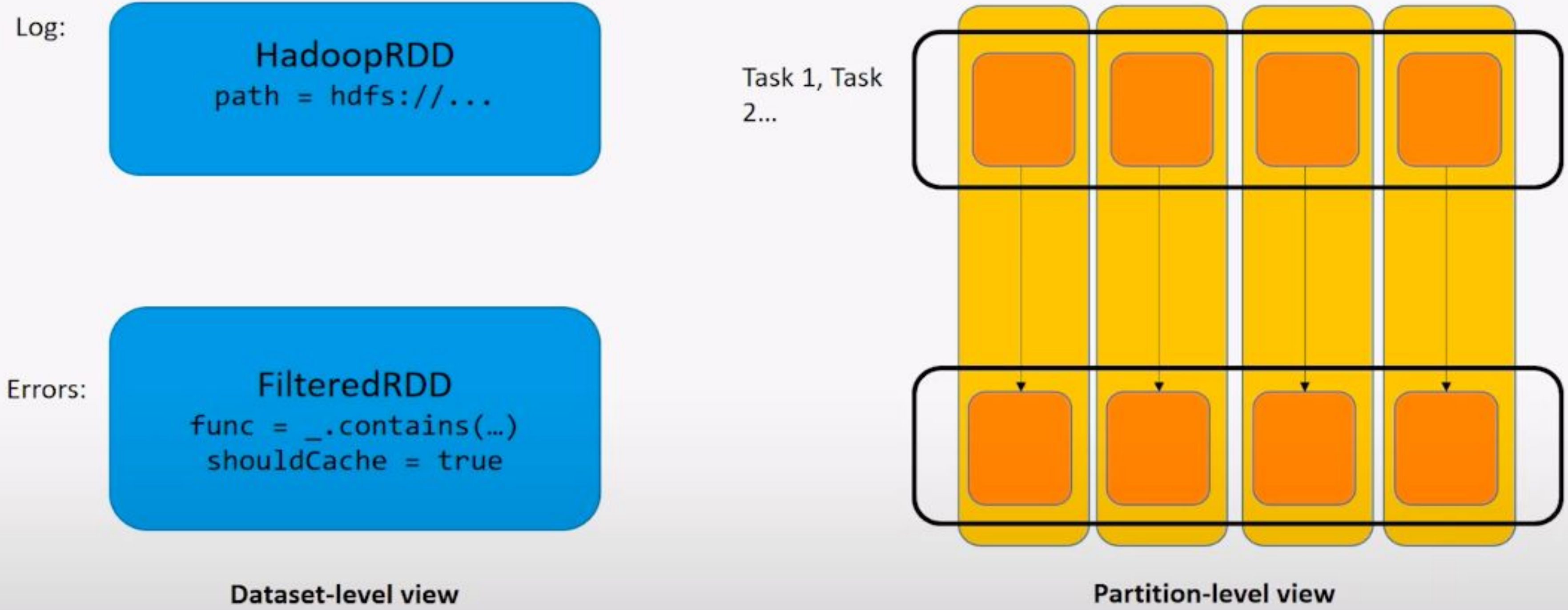
- *actions* to return value or export data
  - actions include *count*, *collect*, *save*
  - triggers execution

Job Example

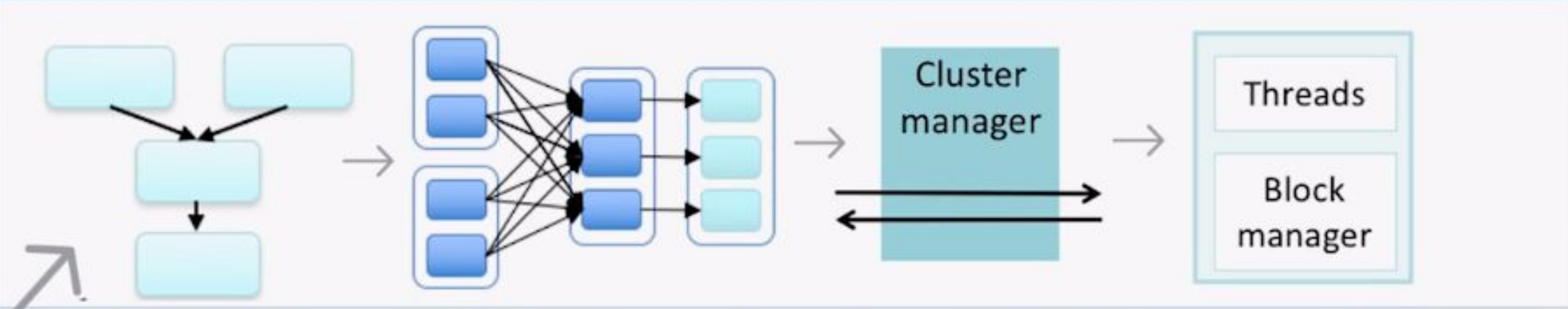
```
val log = sc.textFile(“hdfs://...”)

val errors =
file.filter(_.contains(“ERROR”))

errors.cache()



errors.filter(_.contains(“I/O”)).count
()

errors.filter(_.contains(“timeout”)).c
ount()
```



RDD Partition-Level View



Job Scheduling

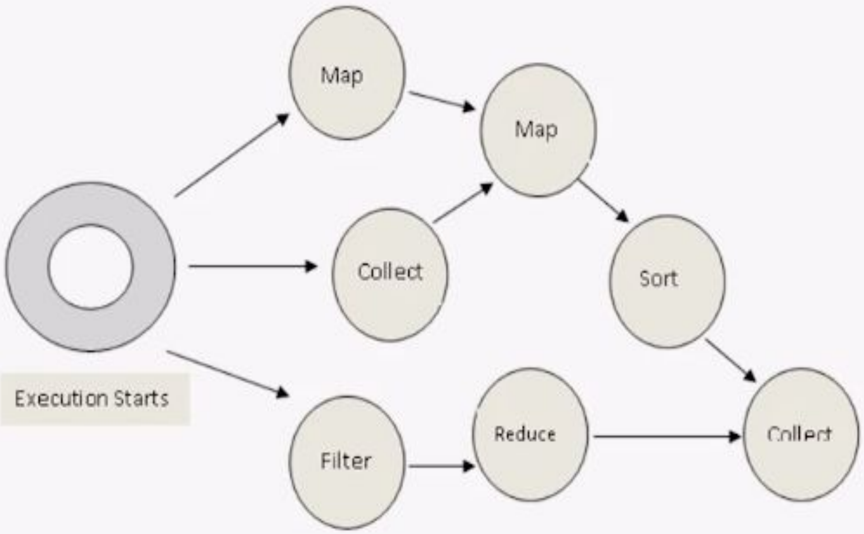


RDD Objects	DAGScheduler	TaskScheduler	Worker

`rdd1.join(rdd2)`
`.groupBy(…)`
`.filter(…)`
	split graph into *stages* of tasks	launch tasks via cluster manager	execute tasks

build operator DAG	submit each stage as ready	retry failed or straggling tasks	store and serve blocks

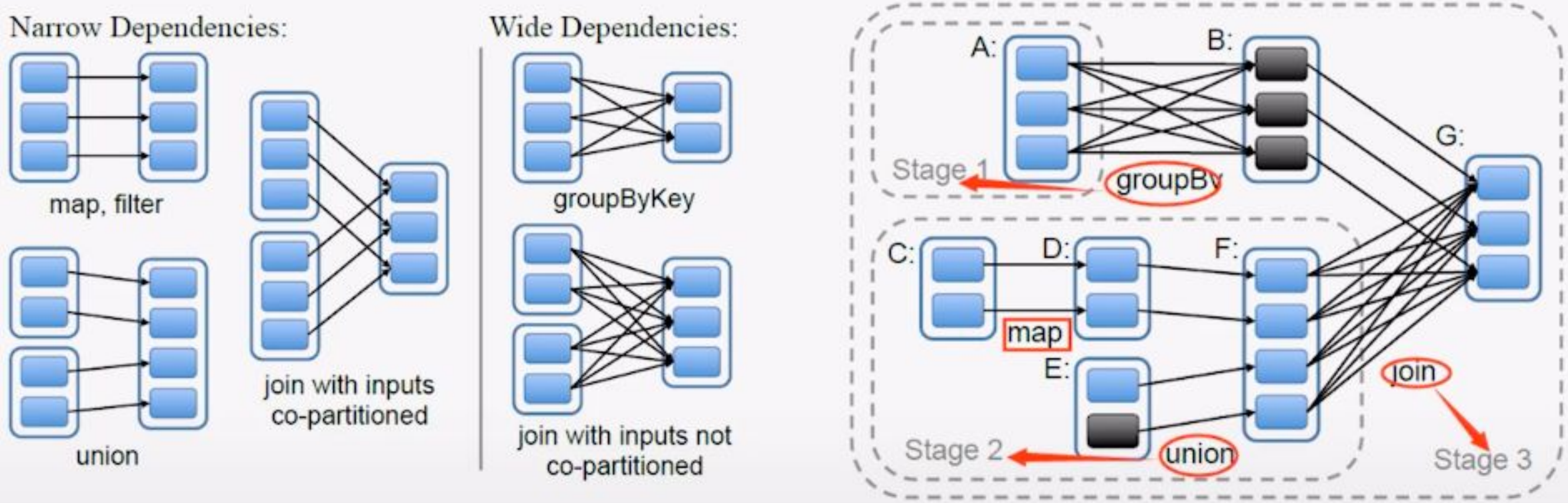
Directed Acyclic Graph (DAG) in Spark



DAG(Directed Acyclic Graph)

Data processing Operations are sorted in a directed acyclic graph

DAG Scheduler in Spark



Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.

Comparing Spark and Hadoop

Spark

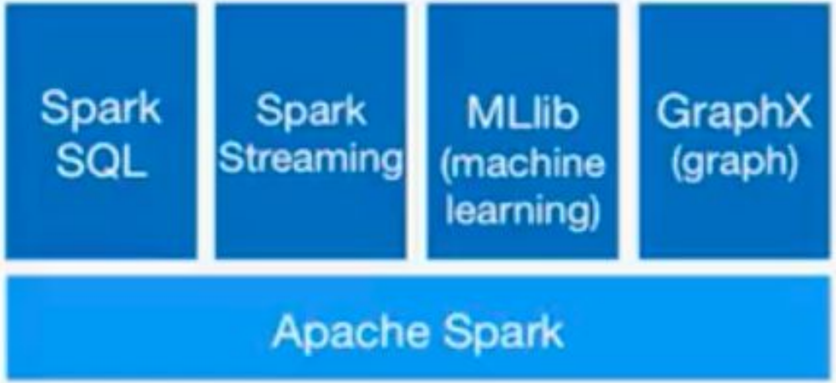
- Spark has an advanced DAG(Directed Acyclic Graph) execution engine
- Spark supports in-memory cluster computing – Thanks to RDD
- Rich Data Processing API (map, filter, reduce, join…)
- Runs on myriad storage engines (HDFS, Cassandra, HBase, S3…)

Hadoop

- Only supports two Runtime phases: Map / Reduce (and hidden data shuffling pahse)
- Intermediate data has to be on disk.
- Everything is programmed using Map/Reduce
- Loads data from HDFS

Spark Ecosystem

Spark Ecosystem

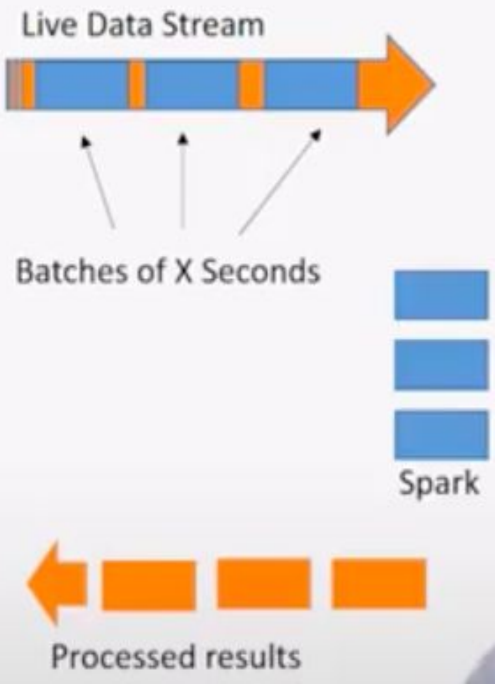


https://spark.apache.org

Spark Streaming: Discretized Stream Processing

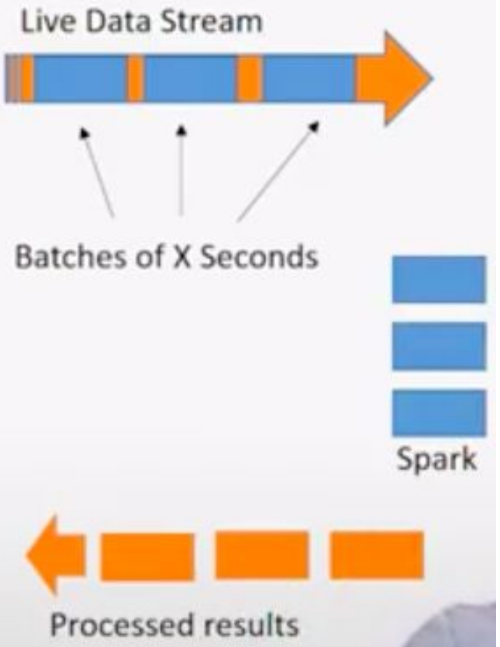
Run a streaming computation as a series of very small, deterministic batch jobs

- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



Spark Streaming: Discretized Stream Processing

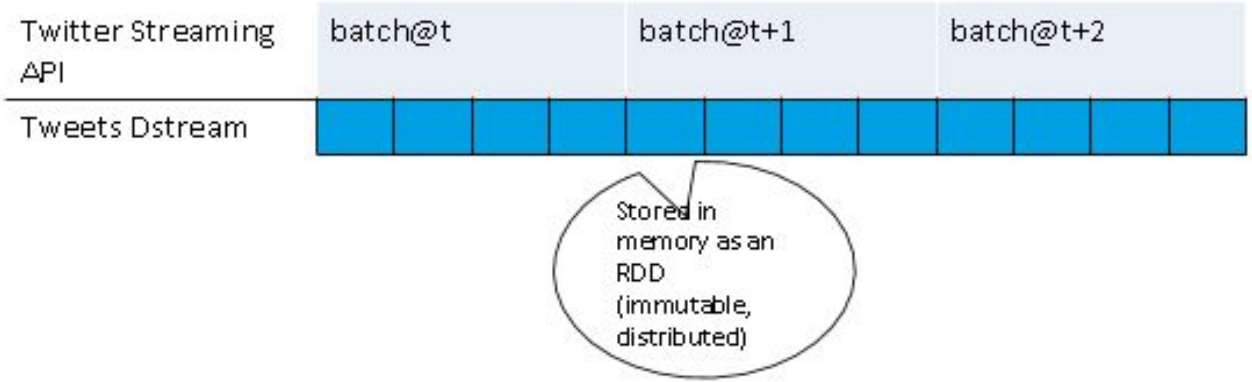
- Batch sizes as low as ½ second, latency ~ 1 second
- Potential for combining batch processing and streaming processing in the same system



Example 1 – Get hashtags from Twitter

- Dstream: A sequence of RDD representing a stream of data

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
```

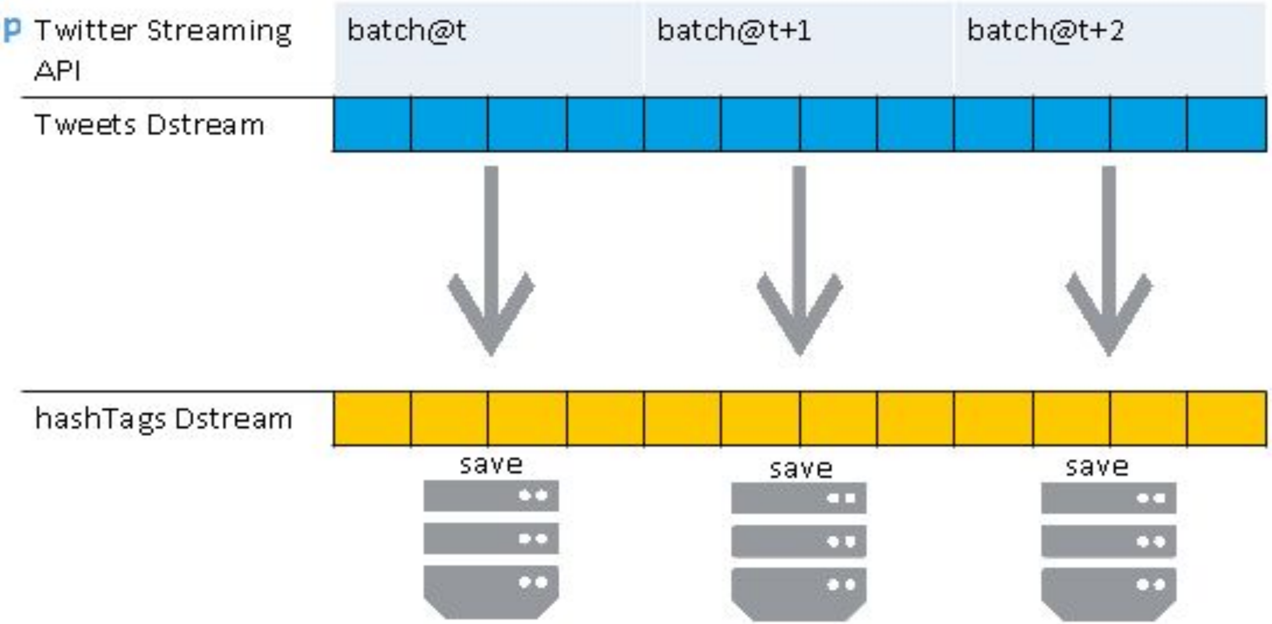


Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)

val hashTags = tweets.flatMap (status => getTags(status))
```

Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)

val hashTags = tweets.flatMap (status => getTags(status))

hashTags.saveAsHadoop
```



Java Example

Scala

```
val tweets =
ssc.twitterStream(<Twitter
username>, <Twitter password>)

val hashTags = tweets.flatMap
(status => getTags(status))

hashTags.saveAsHadoopFiles("hdf
s://...")
```

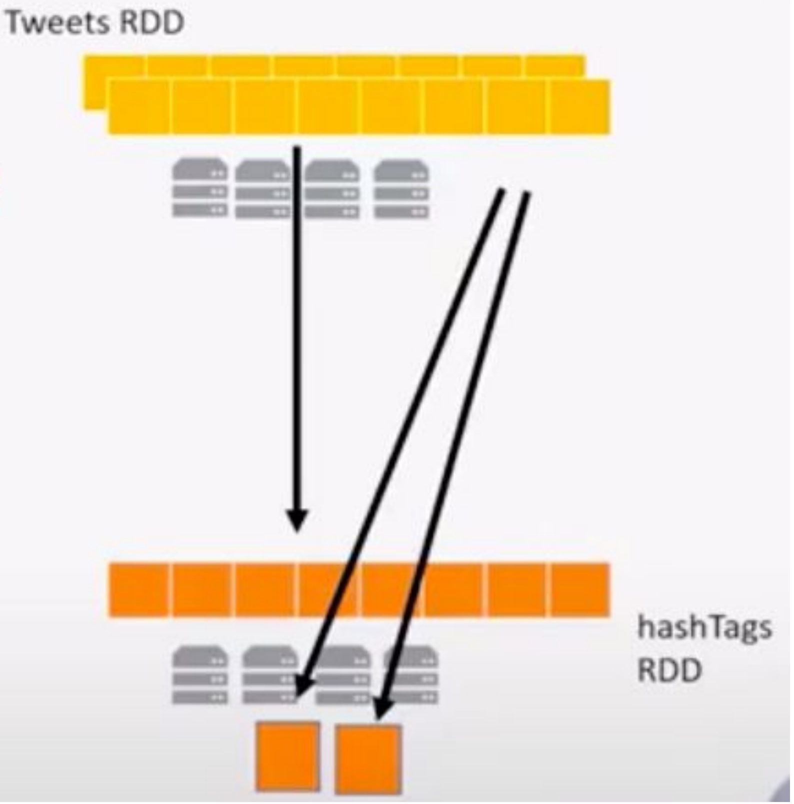
Java

```
JavaDStream<Status> tweets =
ssc.twitterStream(<Twitter
username>, <Twitter password>)

JavaDstream<String> hashTags =
tweets.flatMap(new
Function<...> {  })

hashTags.saveAsHadoopFiles("hdf
s://...")
```

Fault-tolerance

RDDs are

- remember the sequence of operations that created it from the original fault-tolerant input data
- Batches of input data are replicated in memory of multiple worker nodes, therefore fault-tolerant
- Data lost due to worker failure, can be recomputed from input data



Key Concepts

DStream

- Sequence of RDDs representing a stream of data
- Twitter, HDFS, Kafka, Flume, ZeroMQ, Akka Actor, TCP sockets

Transformations

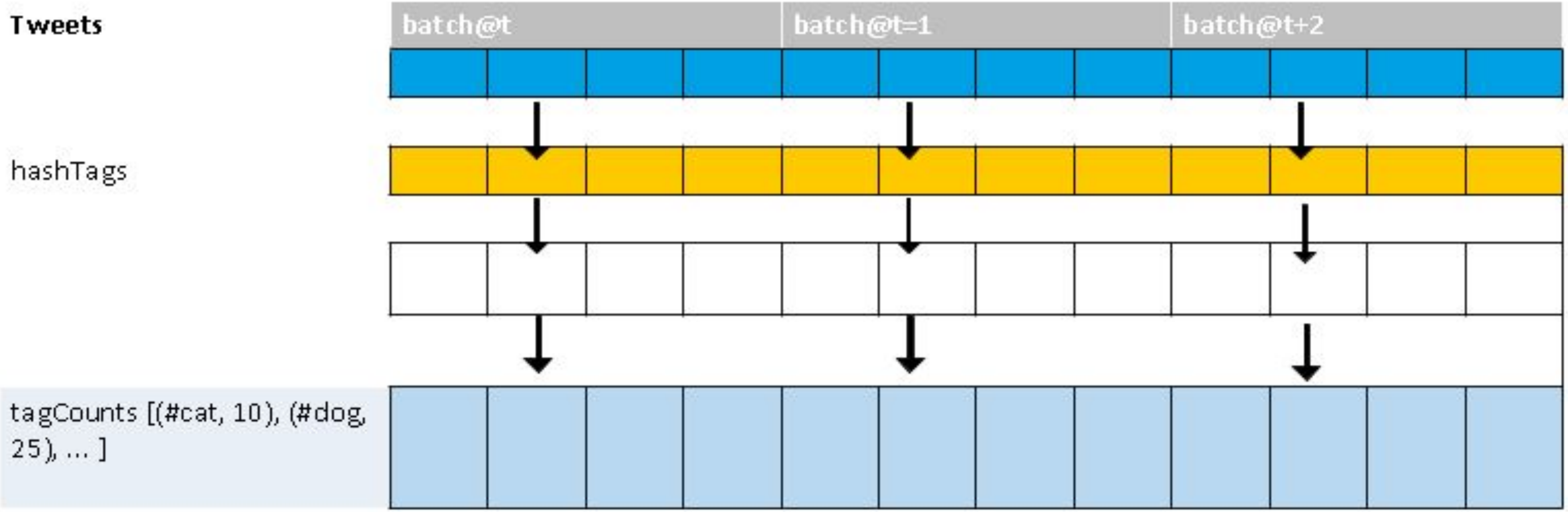
- Modify data from on DStream to another
- Standard RDD operations – map, countByValue, reduce, join, …
- Stateful operations – window, countByValueAndWindow, …

Output Operations – send data to external entity

- saveAsHadoopFiles – saves to HDFS
- foreach – do anything with each batch of results

Example: Count the hastags

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)

val hashTags = tweets.flatMap (status => getTags(status))

val tagCounts = hashTags.countByValue()
```

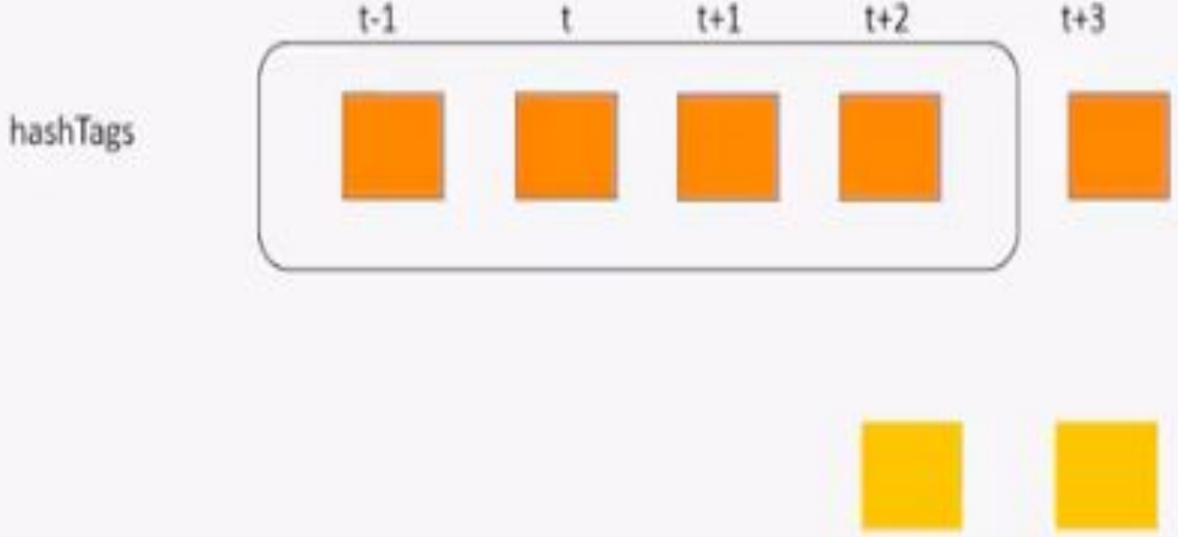


Example: Count the hashtags over last 10 mins

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```

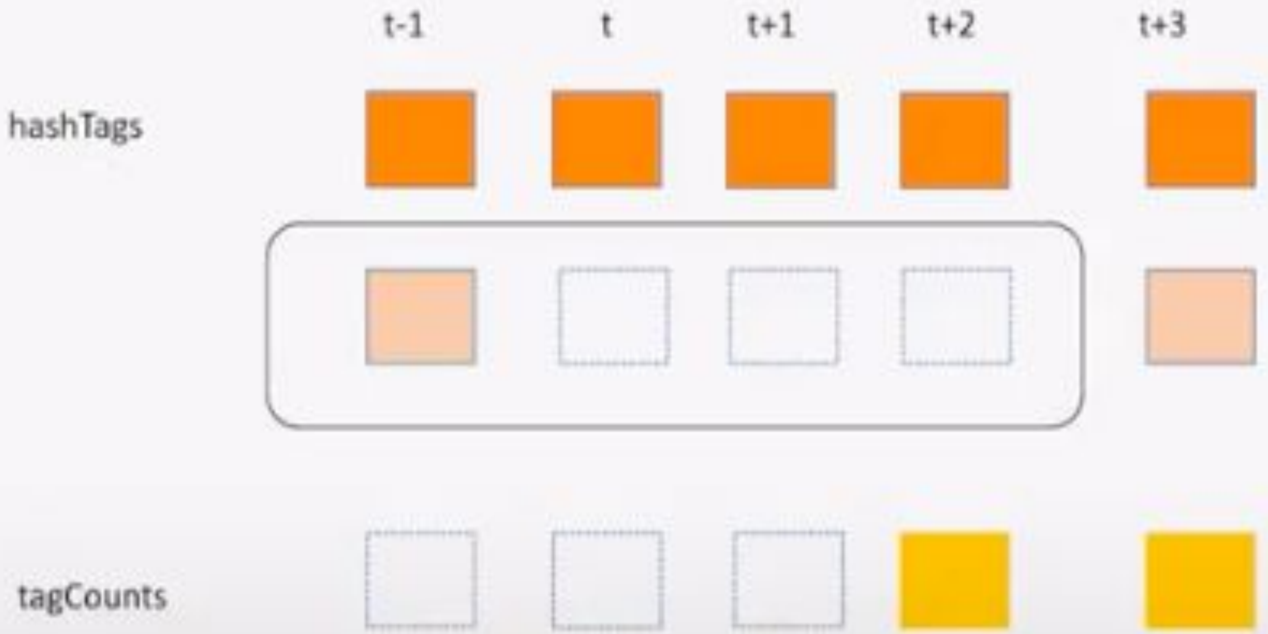
Example: Count the hashtags over last 10 mins

```
Val  = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```



Example: Smart window-based countByValue

```
val tagCounts = hashtags.countByValueAndWindow(Minutes(10),Seconds(1))
```



Smart window-based *reduce*

- Technique to incrementally compute count generalizes to many reduce operations
  - Need a function to "inverse reduce" ("subtract" for counting)

- Could have implemented counting as:

```
hashTags.reduceByKeyAndWin
dow(_ + _, _ - _,
Minutes(1), …)
```

Apache Hadoop Ecosystem

Apache Hadoop Ecosystem

Apache Hive → SQL-like (HiveQL) data warehouse on top of Hadoop

Apache Pig → Process data flow programs (Pig Latin Language) on top of Hadoop

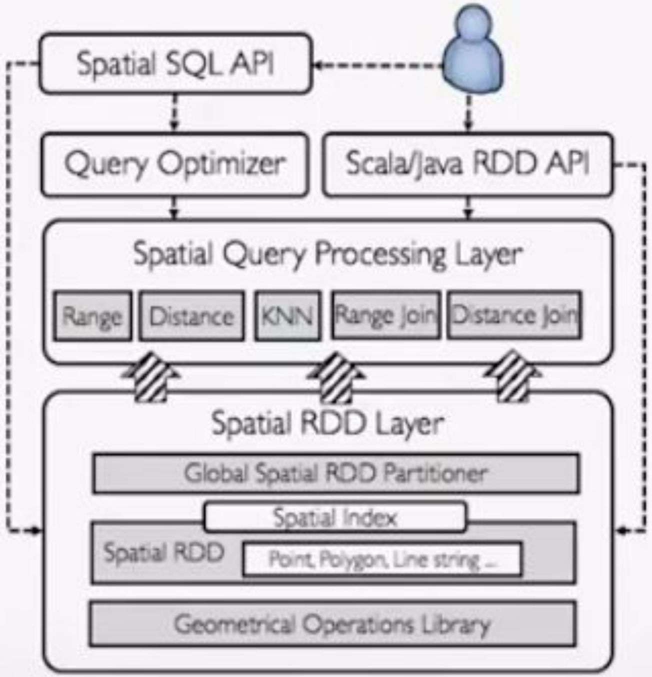
Apache Hbase → NoSQL distributed database (based on BigTable) on top of HDFS

Apache ZooKeeper → distributed configuration service, synchronization service and registry for Hadoop

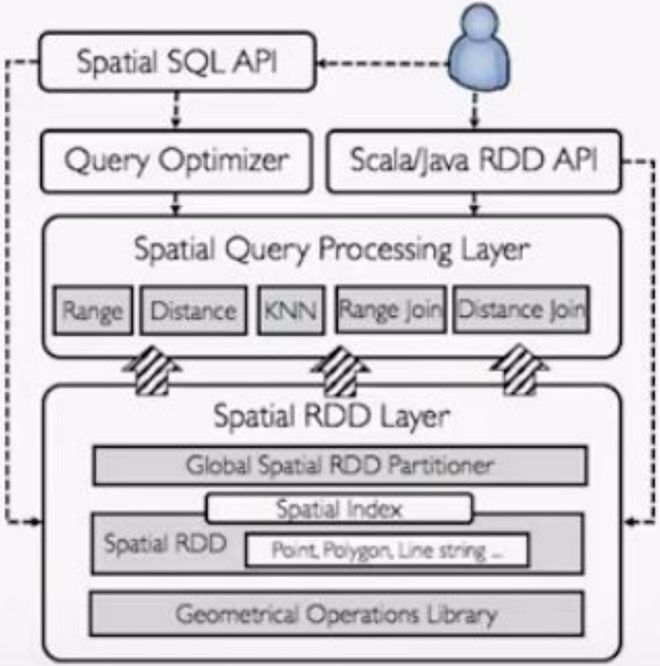
Spatial Data Management in Apache Spark

GeoSpark

Geospark overview

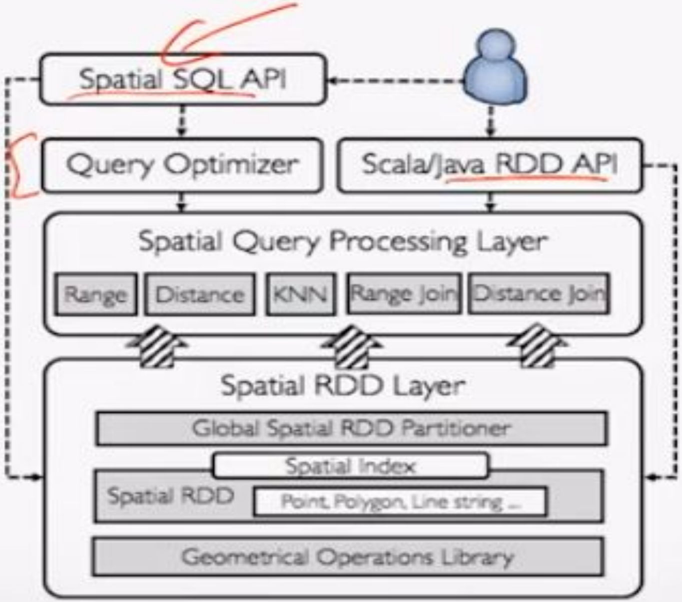


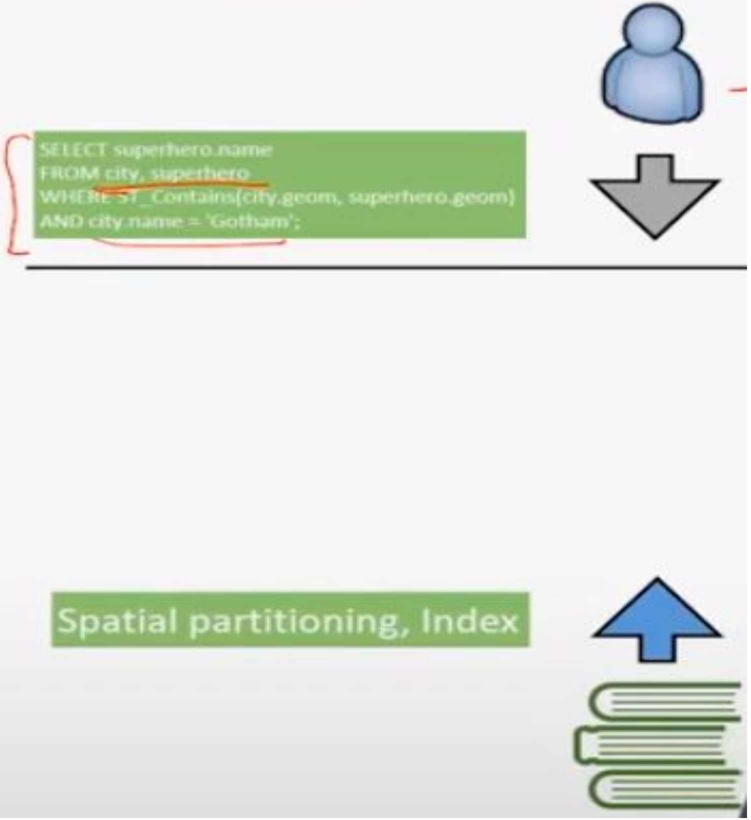
Geospark overview



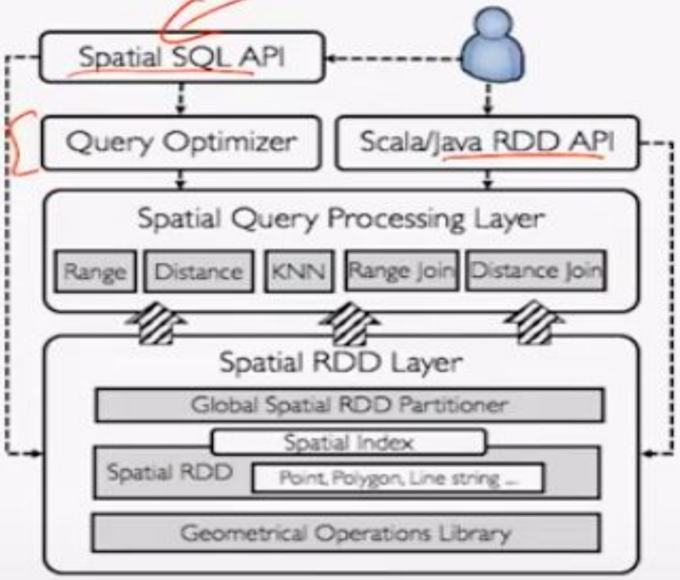

```sql
SELECT superhero.name
FROM city, superhero
WHERE ST_Contains(city.geom, superhero.geom)
AND city.name = 'Gotham';
```

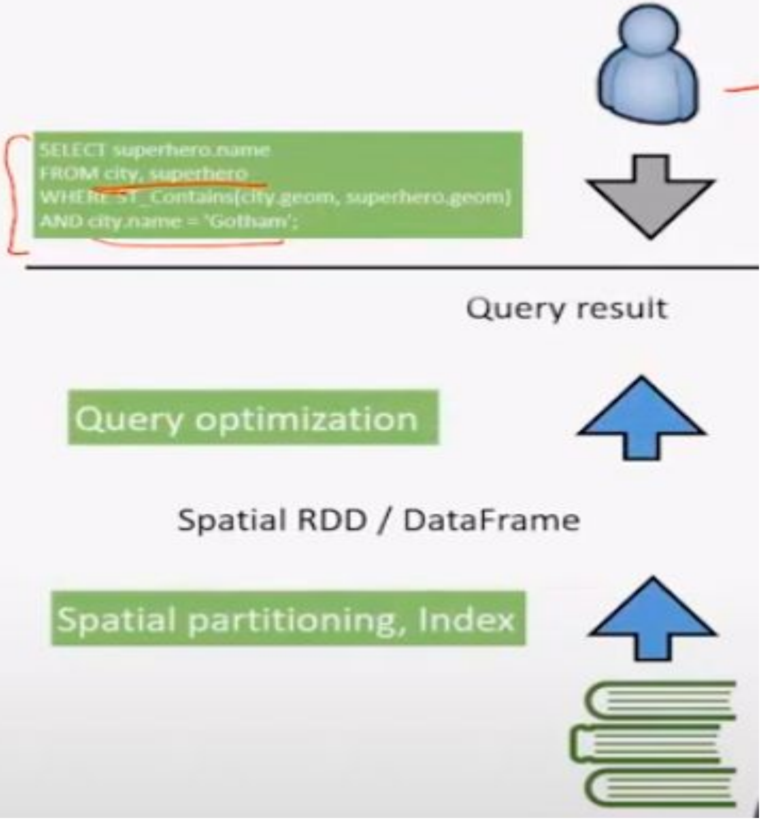
Geospark overview





Geospark overview





Spatial partitioning

Spatial partitioning

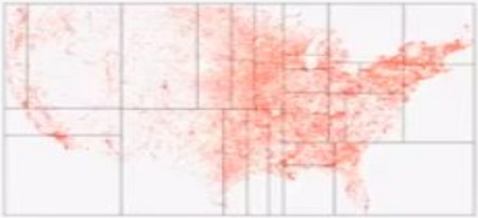
Range query, Join query

Not scalable

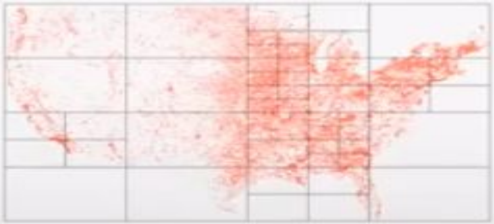
Scalable and fast

Spatial partitioning

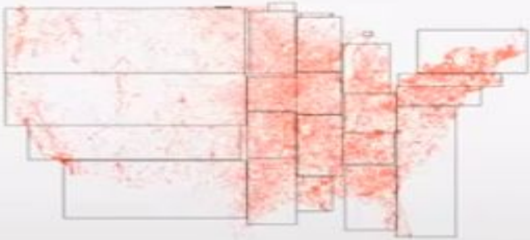
Uniform grids



KDB-Tree

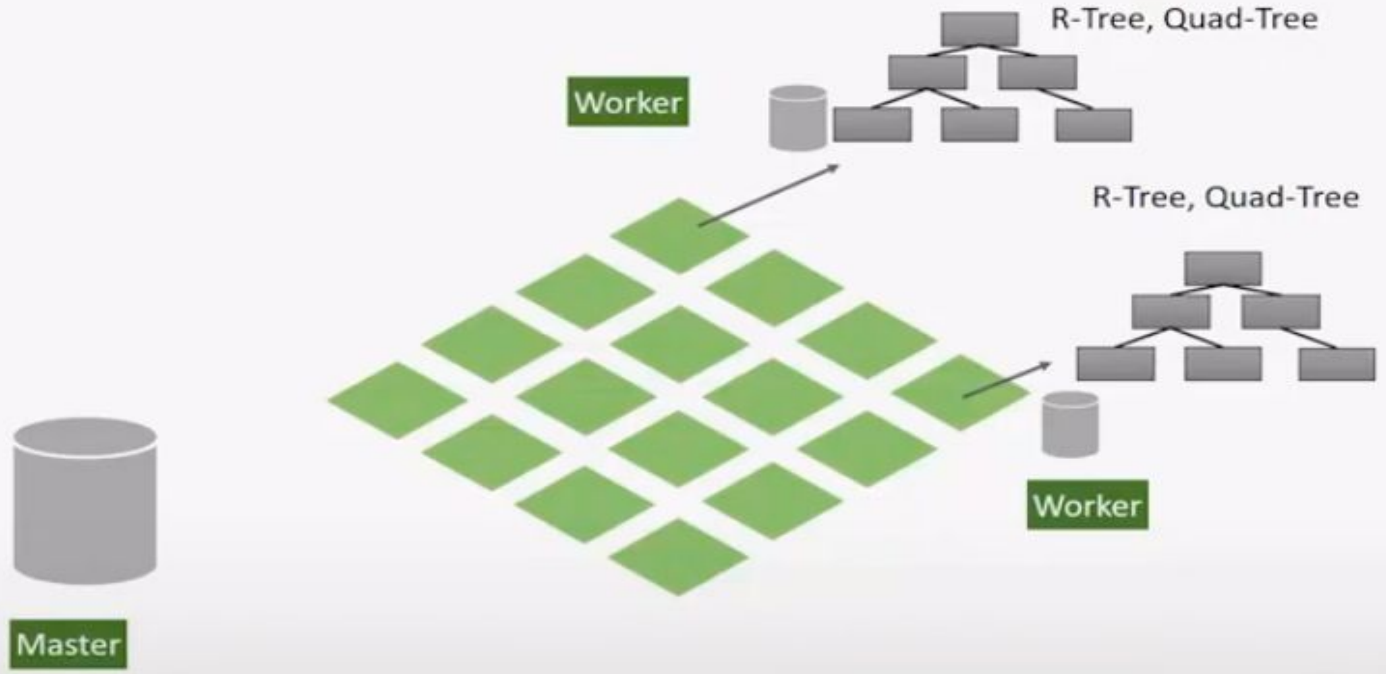


Quad-Tree

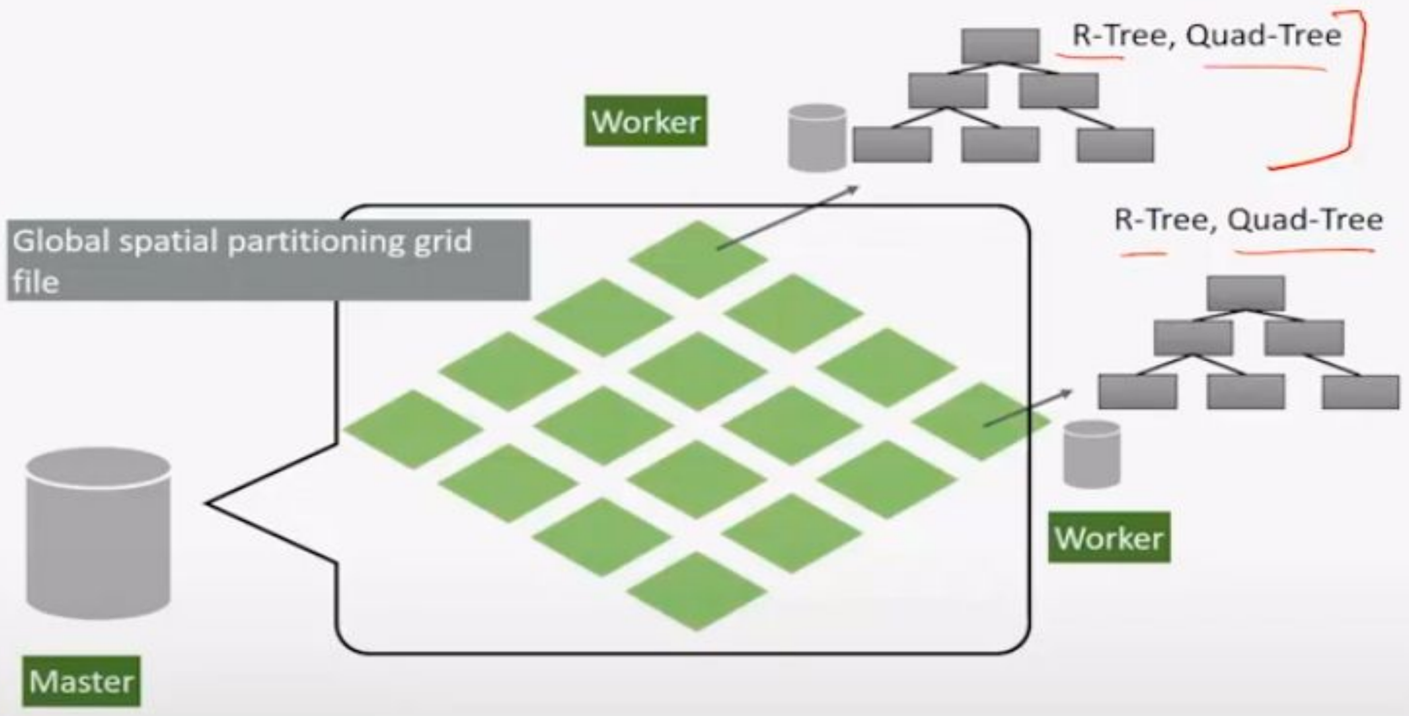


R-Tree

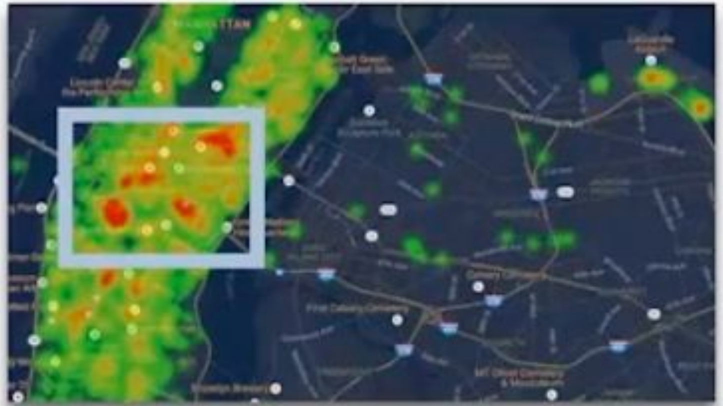
Spatial indexing



Spatial indexing

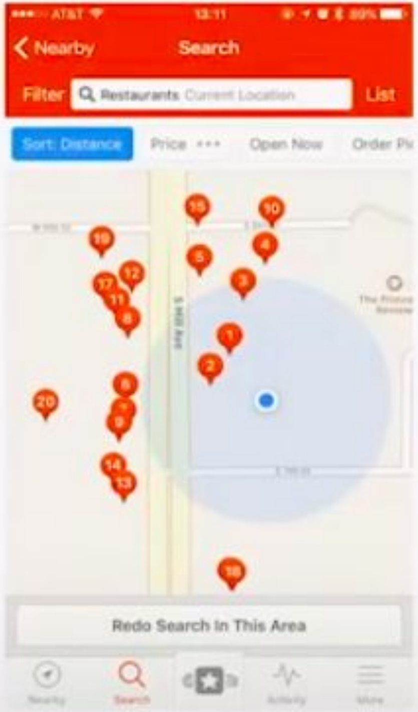


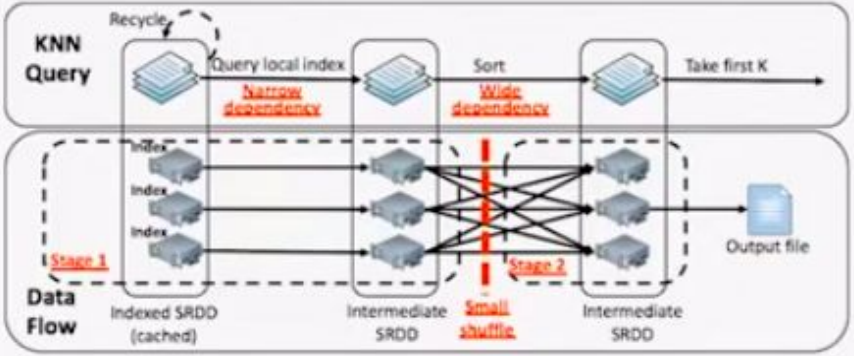
Spatial range query



```
SELECT *
FROM TaxiTripTable
WHERE ST_Contains(Manhattan, TaxiTripTable.pickuppoint)
```

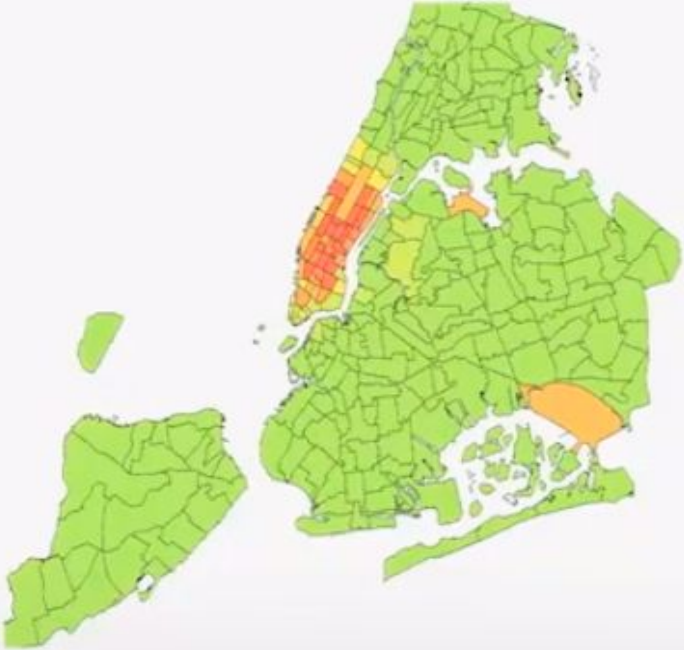
Spatial KNN query





SELECT ST_Neighbors(MyLocation Restaurants.Locations, 20)
FROM Restaurants

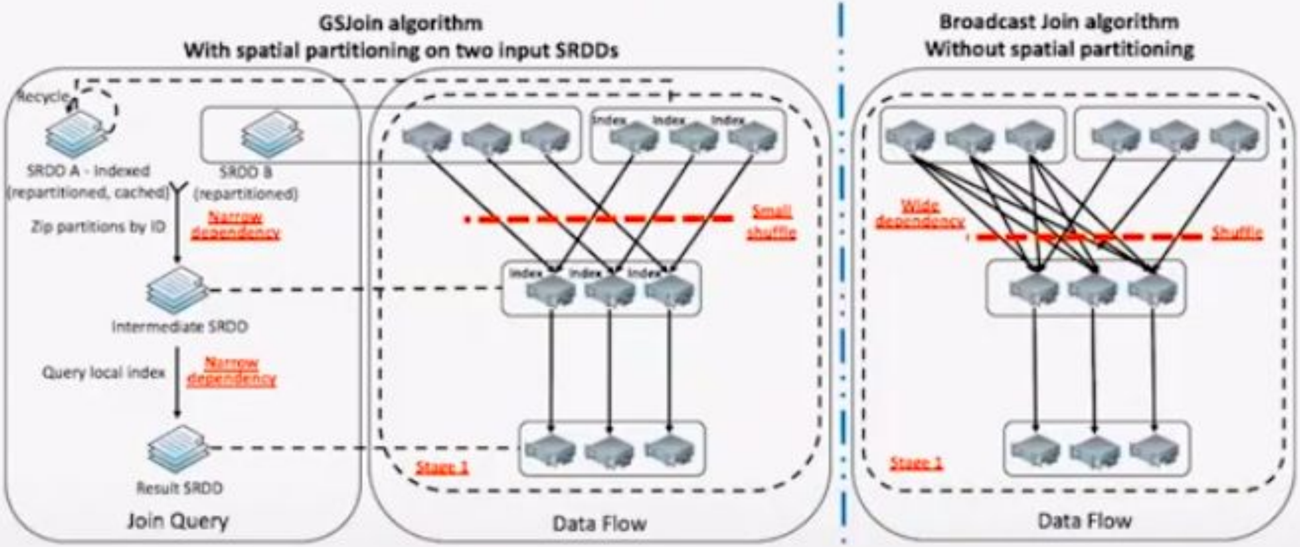
Spatial join query



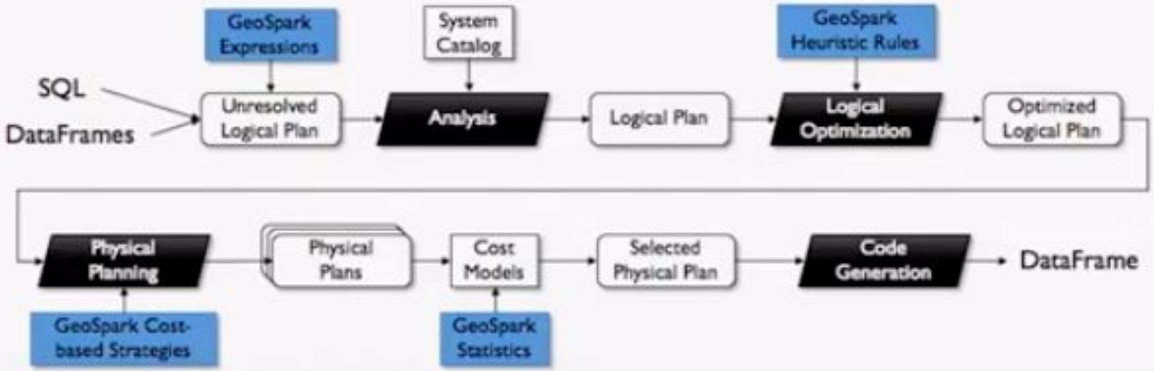
Spatial join query

```
SELECT *
FROM TaxiZones, TaxiTripTable
WHERE ST_Contains(TaxiZones.bound, TaxiTripTable.pickuppoint)
```

Spatial join query



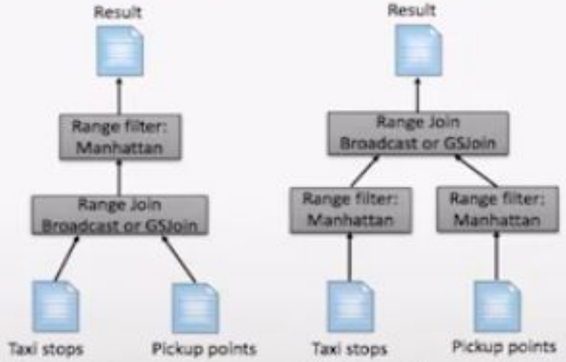
query optimizer (v1.2.0)



Heuristics based optimization

- Predicate pushdown

```
SELECT *
FROM TaxiStopStations, TaxiTripTable
WHERE ST_Contains(TaxiStopStations.bound, TaxiTripTable.pickuppoint)
AND ST_Contains(Manhattan, TaxiStopStations.bound)
```



(a) No predicate pushdown (b) With predicate pushdown

Heuristics based optimization

- Predicate merging

```
SELECT *
FROM TaxiTripTable
WHERE ST_Contains(Manhattan, TaxiTripTable.pickuppoint) AND ST_Contains(Queens,
TaxiTripTable.pickuppoint)
```

Heuristics based optimization

- Predicate merging

```
SELECT *
FROM TaxiTripTable
WHERE ST_Contains(Manhattan, TaxiTripTable.pickuppoint) AND ST_Contains(Queens,
TaxiTripTable.pickuppoint)
```

(a) AND, take the intersection

```
SELECT *
FROM TaxiTripTable
WHERE ST_Contains(Manhattan, TaxiTripTable.pickuppoint) OR ST_Contains(Queens,
TaxiTripTable.pickuppoint)
```

Heuristics based optimization

- Predicate merging

```
SELECT *
FROM TaxiTripTable
WHERE ST_Contains(Manhattan, TaxiTrip ickuppoint) AND ST_Contains(Queens,
TaxiTripTable.pickuppoint)
```

(a) AND, take the intersection

```
SELECT *
FROM TaxiTripTable
WHERE ST_Contains(Manhattan, Tax ht) OR ST_Contains(Queens,
TaxiTripTable.pickuppoint)
```

(b) OR, take the union

Heuristics based optimization

- Intersection query rewrite

```
SELECT ST_Intersection(Lions.habitat, Zebras.habitat)
FROM Lions, Zebras
```

```
SELECT ST_Intersection(Lions.habitat, Zebras.habitat)
FROM Lions, Zebras
WHERE ST_Intersects(Lions.habitat, Zebras.habitat);
```

Heuristics based optimization

- Intersection query rewrite

```
SELECT ST_Intersection(Lions.habitat, Zebras.habitat)
FROM Lions, Zebras
```

Cross join, slow

```
SELECT ST_Intersection(Lions.habitat, Zebras.habitat)
FROM Lions, Zebras
WHERE ST_Intersects(Lions.habitat, Zebras.habitat);
```

Optimized GeoSpark inner join, fast

Cost based optimization

- Cost: based on GeoSpark statistics, MBR, count
- Index scan selection: Index scan VS DataFrame scan, based on query selectivity
- Spatial join algorithm selection: partition-wise GeoSpark join VS broadcast join

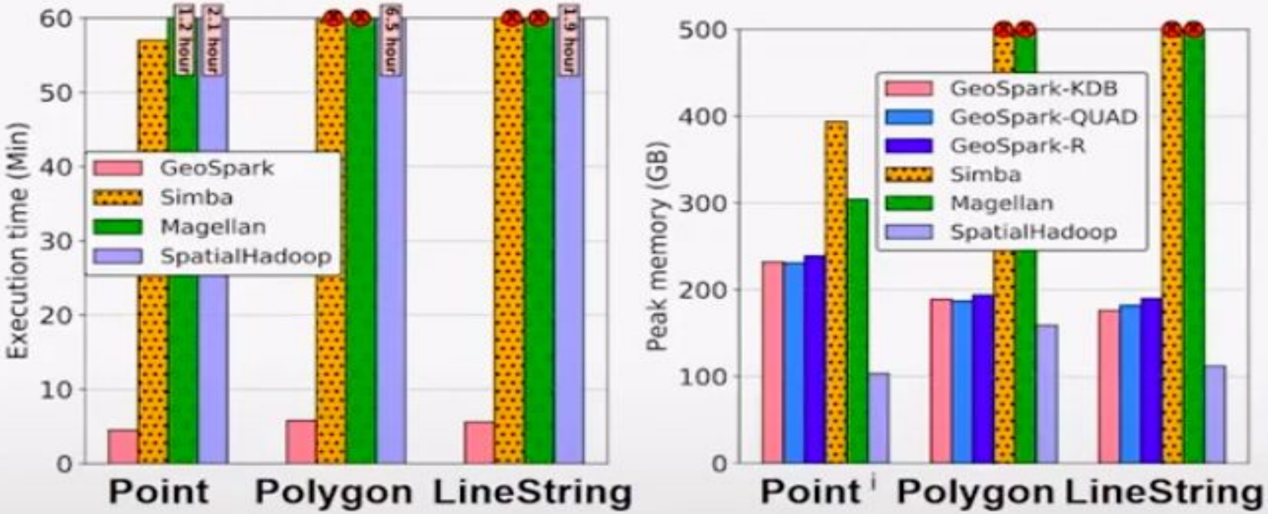
Spatial join query

1.3 billion points join 171 thousand polygons

72.7 million line strings join 171 thousand polygons

263 million polygons join 171 thousand polygons

4 machines

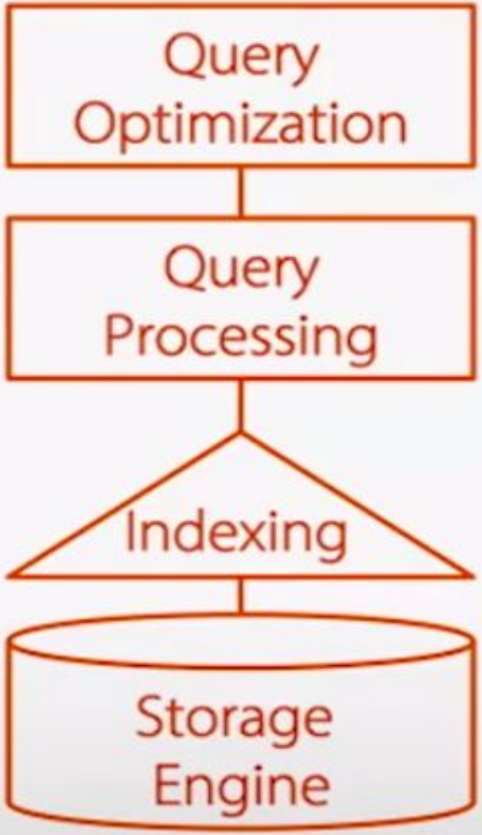


(a) Execution time

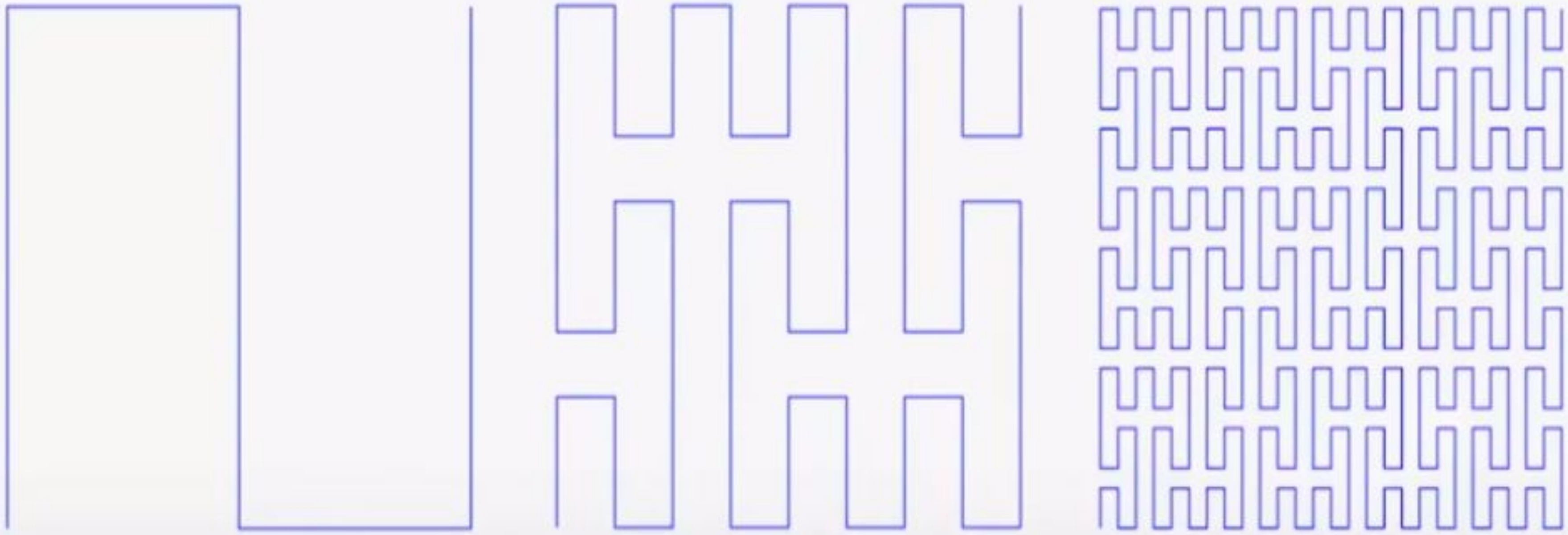
(b) Peak memory utilization

Optimize the database engine for spatial data

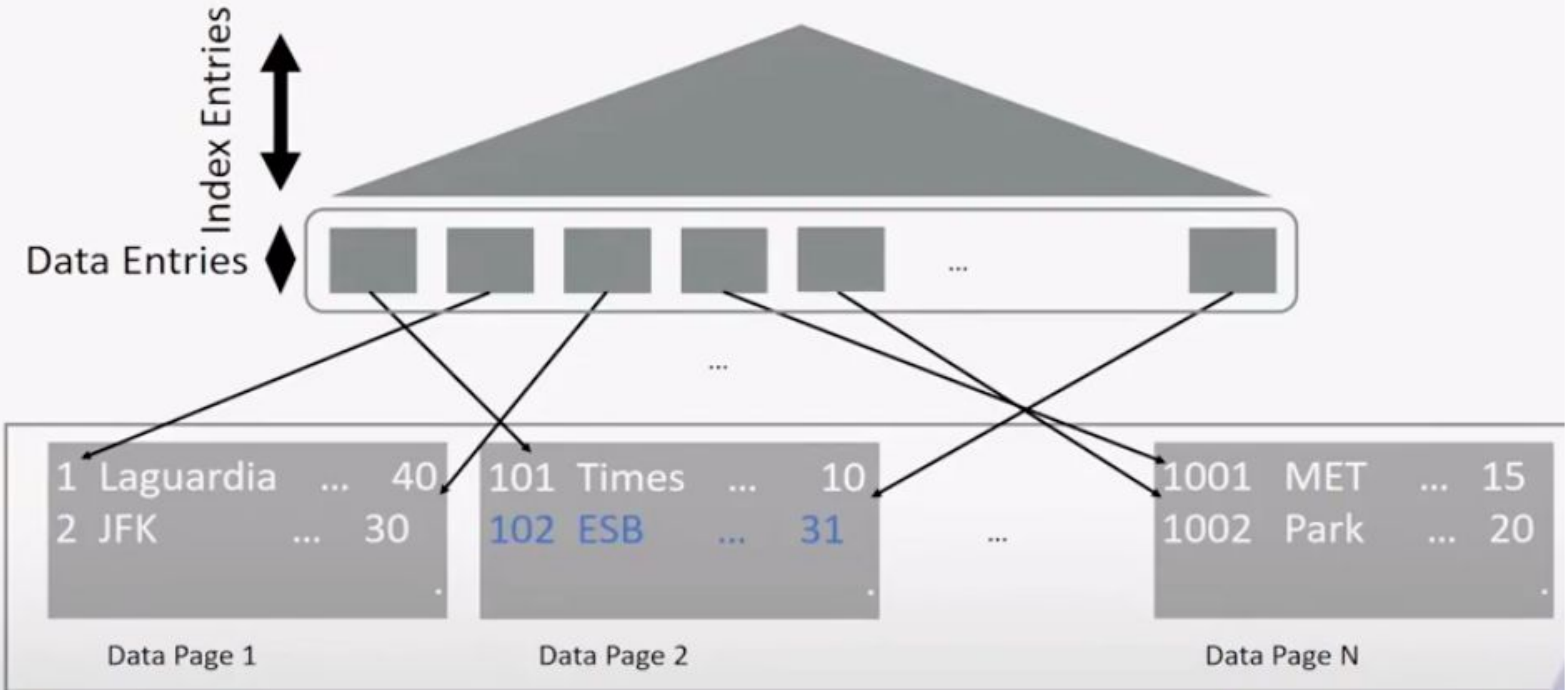
Optimize the database engine for spatial data



Convert Spatial Data to 1D data

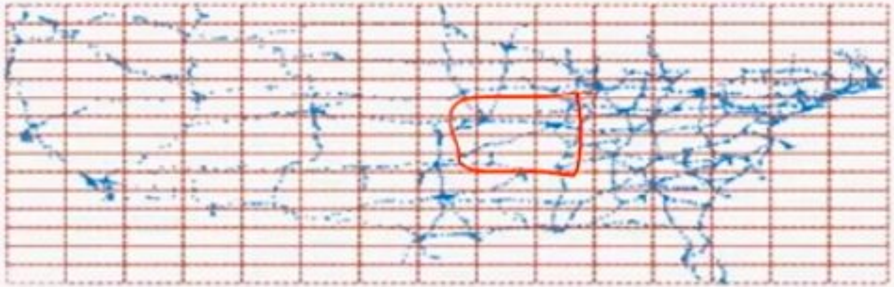


Spatial Indexes



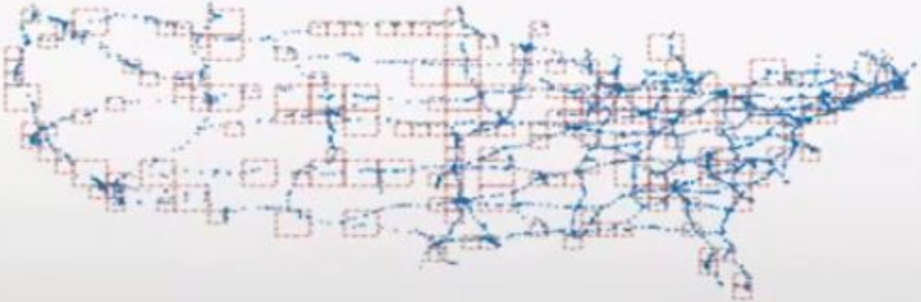
Spatial Indexing Methods

Uniform Grid

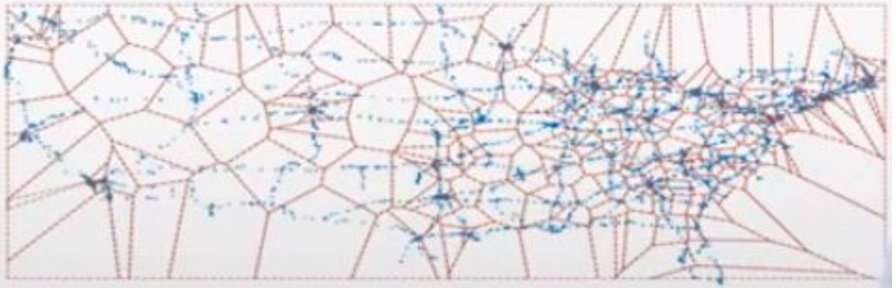


R-tree

Quad Tree



Voronoi Diagram



Conclusion

- Spatial data is special
- We had to extend SQL to support spatial data
- Many opportunities to optimize the query processing and indexing/storage layers for spatial data