

Analyzing Massive Data Sets

Summer Semester 2019

Prof. Dr. Peter Fischer

Institut für Informatik

Lehrstuhl für Datenbanken und Informationssysteme

Chapter 3: Spark

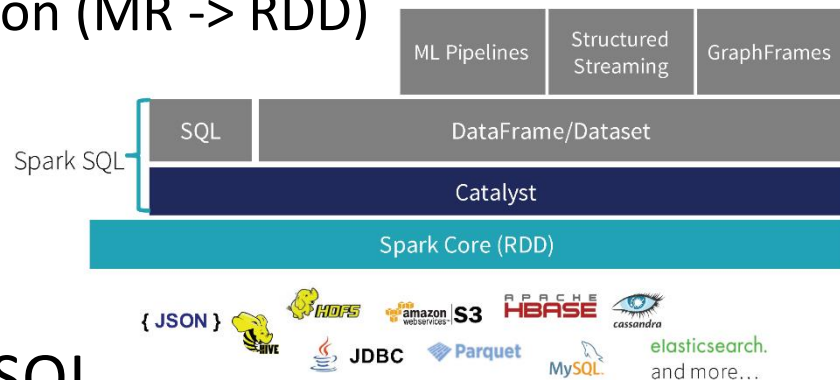
Limitations of Map-Reduce

- No support for general workflows (single-stage map-group-reduce)
- No support for iterative workloads (graph algorithms, learning methods)
- Restricted set of operations
- Low level of abstraction (-> specialized hooks into M/R: combiner, partitioner, ...)
- Low operational efficiency due to
 - mandatory materialization (failure resilience)
 - lack of data type knowledge
- Long response times (batch processing)

Apache Spark

- Core Idea: Distributed, shared memory with coarse transformations (RDD)
- Support (and integration) for various different processing requirements

- Unstructured, general distribution (MR -> RDD)
- Structured querying (SQL)
- Data Streams: ordered, infinite
- Graphs: iteration
- Machine Learning



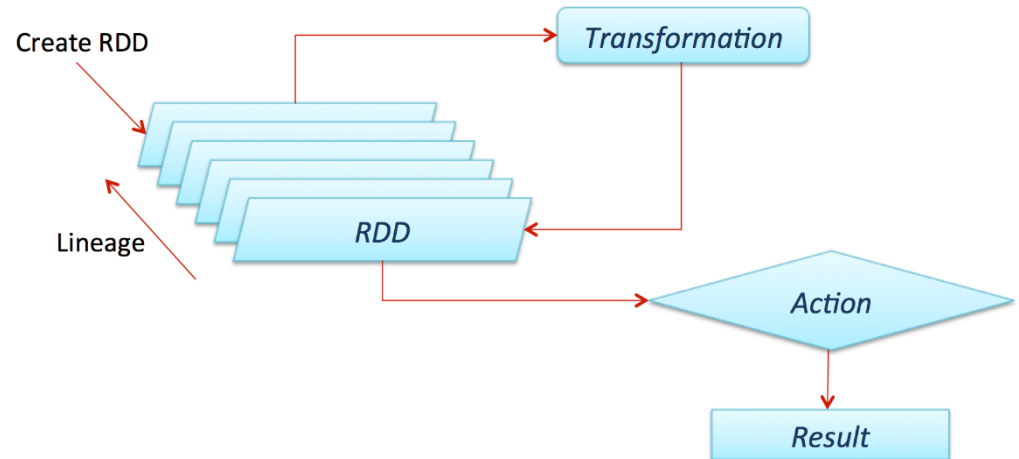
- APIs in Scala, Java, Python, R, SQL
- Integrates into cluster managers (Hadoop YARN, Kubernetes, Mesos) as well as standalone
- Ingestion of various data formats from wide range of source platforms (HDFS, K/V stores, (No)SQL DB)

RDDs (old core API)

- **Distributed** collection
 - In-memory caching
 - Programmable partitioning and co-locationing
- **Parallel, coarse** transformations
 - Richer set of operations than map and reduce
- **Immutable** data
 - Transformations generate new RDD
 - No locking, coordination
 - Possibly higher memory consumption
- Fault Tolerance via **Lineage**
 - Recompute results from input dependencies
 - Checkpointing only as optimization
- **Lazy** evaluation
 - Compute only when results are requested

Working with RDDs

- Create RDD from
 - Normal collections
 - External sources (File, HDFS, ...)
- Transformations
 - Bulk operation on existing RDDs
 - Define new RDD
 - Keep track of input RDD(s): lineage
- Actions:
 - Turn RDD into "normal" values /collections
 - Trigger the execution



Examples of RDD operations

```
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);  
JavaRDD<Integer> distData = sc.parallelize(data);
```

```
JavaRDD<String> distFile = sc.textFile("data.txt");
```

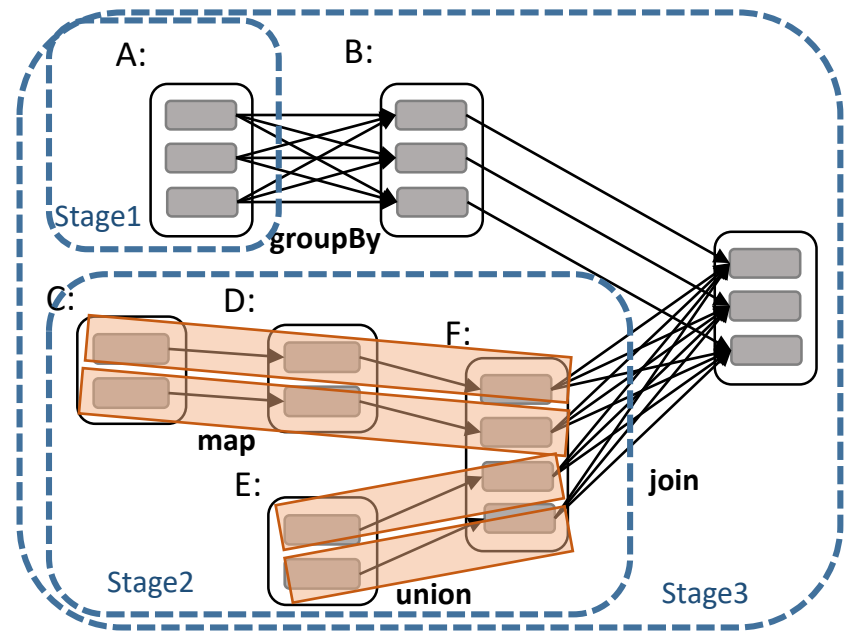
```
rdd.filter(p -> p.getAge() < 21)    // transformation, lambda expression  
    .map(p -> p.getLast())           // transformation  
    .saveAsObjectFile("under21.bin"); // action
```

```
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());  
int totalLength = lineLengths.reduce((a, b) -> a + b);
```

```
pairs = lines.map(lambda s: (s, 1)) # Python  
counts = pairs.reduceByKey(lambda a, b: a + b)
```

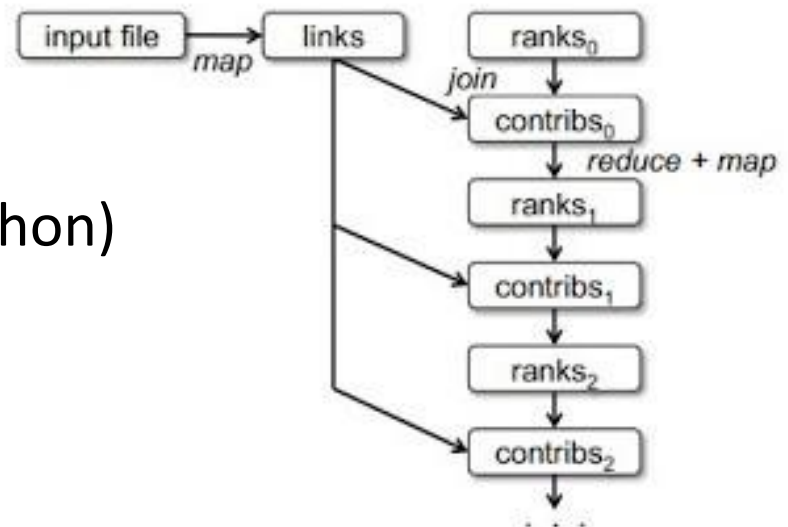
Representing and Executing RDDs

- Native objects in memory, Java/Kryo serialization when transferring over the network
- Controllable partitioning
- Two dependency types:
 - **Narrow:** Each parent partitions contributes to exactly one child partition:
 - map, filter, union, join with fitting partitions
 - **Wide:** At least one parent contributes to multiple children:
 - group by, general join, distinct, ...
- Narrow dependencies can be co-scheduled: stages



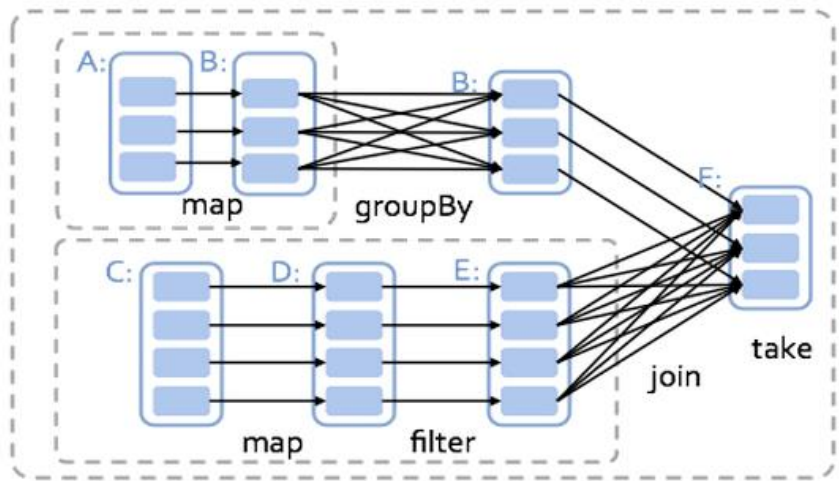
Persisting RDDs

- Framework as well as users can decide if an RDD should persisted
- If an RDD is not persisted, partitions could be discarded when memory is limited
- Explicitly persisting useful for many iterative workloads: cache network structure for path computation, pagerank, ...
- Various levels of persistence
 - Original in memory (Java, Python)
 - Serialized in memory (bytes)
 - On disk (bytes)
 - Replication (orthogonal)



Recovery and Fault Tolerance

- If nodes containing RDDs fail, the partitions can be recomputed based on the dependencies
- Walk back to input in stable storage and restore
- Narrow dependencies easy to recover
- Wide dependencies trigger large recomputations
- Cached RDDs can be reused
- Checkpoints
 - Store RDD in stable storage (e.g., HDFS)
 - Truncate lineage before (optional)
 - Speed up recovery



DataFrames

- Key idea: Declarative API, typed storage, optimizations
- Inspired by, but not identical to Python/R Data Frames
- Tabular structure with different data types (vs object collection in RDDs)
- Nested relational format like PIG, Hive (<-> JSON)
 - Atomic types (strings, integers, float)
 - Sequences, maps, ...
- Define or imply schema from sources
- Example expressions:
 `df.filter("age > 21");` // SQL style query string
 `df.filter(df.col("age").gt(21));` // expression builder
 `df.filter(df.col("happy") == true)` instead of
 `rdd.filter(lambda x: x.happy == true)`

Assessment of DataFrames

Strengths

- + Declarativity allows query plan optimization
- + Strongly typed data model allows for optimized storage
- + Query compilation avoid interpretation overhead

Limitations

- Compile time type checking restricted:
`df.col("happy")` may yield a runtime error
- No custom lambdas possible,
DataFrames first need to be converted to RDDs
- No syntax checking in Spark SQL
- DataFrame API somewhat limited

Data Sets

- Key idea: Combination of RDD and DataFrames
 - + When fully using strong typing everything is checked during compile time
 - + Optionally also weakly typed objects usable
 - + Retains optimizability and code compilation
- Like DataFrames, Dataset API's are internally mapped to RDDs, share lineage, recovery, execution,...

Sample expressions:

```
dataset.filter(person -> person.getAge() < 21);
```

- Spark 2.0 unifies Data Sets and DataFrames (~DataSet[Row])
- Python/R only use DataFrames (lack of type information)

Common Generic Transformations

- Single input RDD
 - map, flatMap
 - filter
 - distinct
 - sample
- Multiple input RDDs
 - union
 - Intersection
 - subtract (set)
 - cartesian / crossproduct
- Operations on specialized RDDs
 - numeric: count, sum, mean, max
 - key/value: group, reduce, (outer) join, cogroup, sort

Common RDD Actions

- Collect (all elements to an array)
- Count
- countByValue
- Take (n elements)
- First/Top
- takeOrdered
- Reduce/fold/aggregate
- Foreach
- saveAs{Text/Sequence/Object}File

- Produce non-RDD output

Dataset/Frame Operations

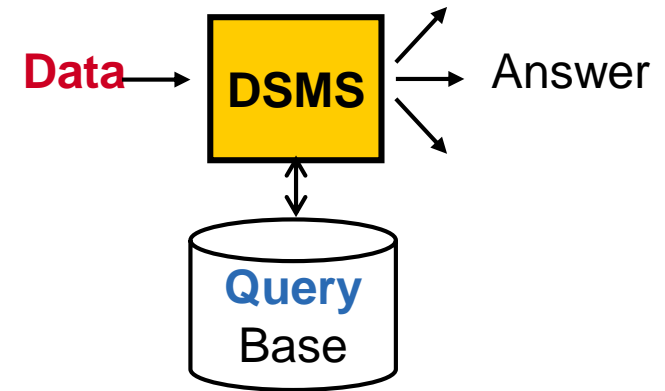
- select, as, drop
- filter, where
- head/limit
- Distinct
- Union, except, intersect
- Join, crossJoin
- Map, MapPartitions
- Aggregations (global)
- groupBy
- cube/rollup
- Metadata: columns, dtypes, count
- cache/persist, unpersist, Checkpoint, storageLevel
- toRDD, write, toDF

Spark and SQL

- Express SQL queries directly on dataframe/sets
- Fully composable with expressions
- SQL dialect close to HiveQL
 - Analytical SQL: SQL 92+nested columns, windows, cube,...
 - DDL with partitioning information
 - User-defined functions in Scala, Java, Python
- Tungsten:
 - Type-specialized, column storage based on Schema
 - Java bytecode generation instead of iterators
 - vectorization in recent versions
- Catalyst: cost-based, extensible query optimizer
- Input/Output to JSON, Parquet, relational DB
- Expression pushdown even into sources (JDBC database, Parquet file)

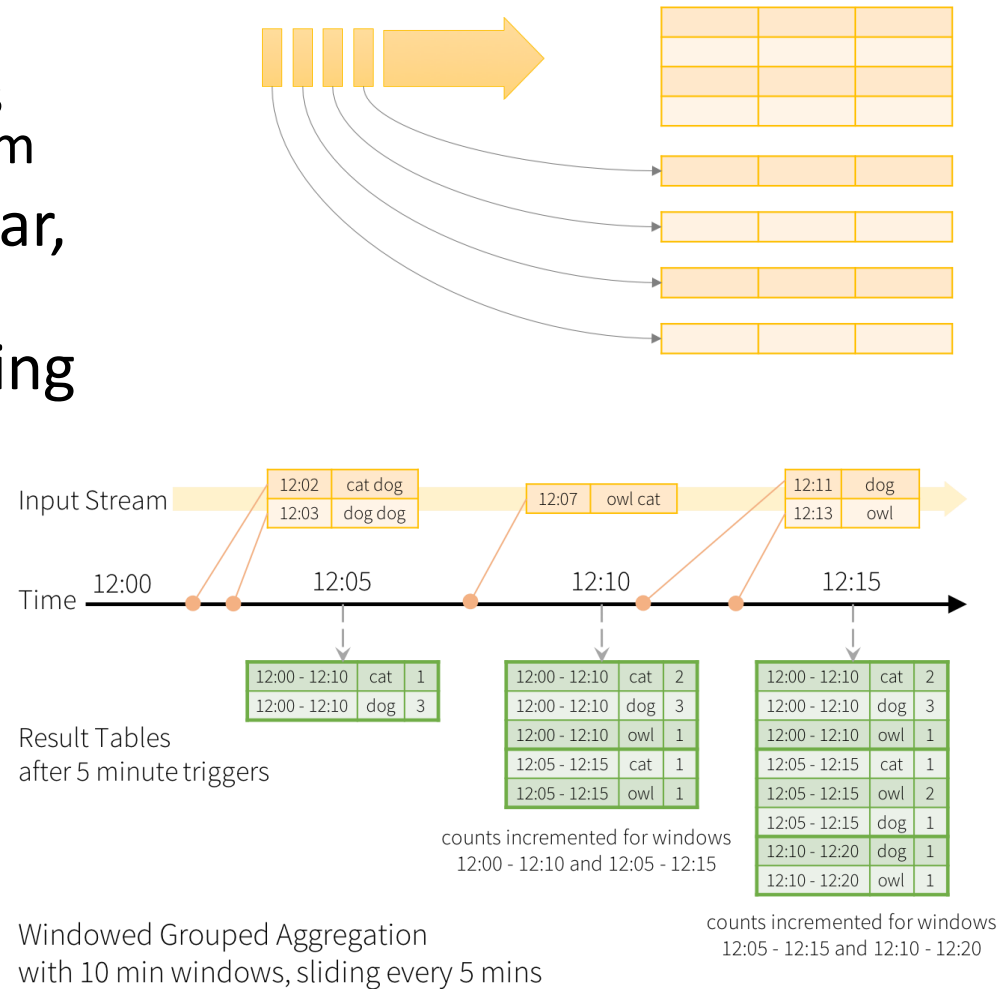
Streams

- **Continuous queries over ordered, possibly infinite data**
- Often:
 - Varying and/or very high data rates
 - Low response times required
- Standard approach:
 - Process data without storing
 - Tuple-at-a-time
- Challenges in large-scale, distributed streaming:
 - Consistency/Coordination
 - Recovery
 - Out-of-order arrival
- Spark Streaming: discretized stream aka mini-batching



Structured Streaming

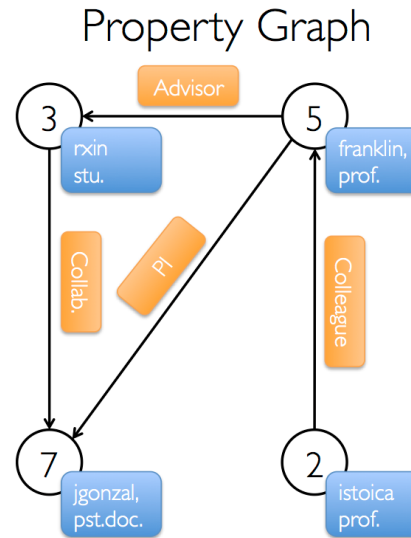
- Unbounded Data Frame:
 - append new rows
 - Keep old rows as long as expressions request them
- Execute queries in regular, short intervals (100 ms)
- Bound relevant state using window expressions
- Watermarking for late arrivals
- Limited support for streaming expressions
- Higher latency
- Easier integration with other models



Graphs: GraphX

- Property Graph Model

- Vertices (with id) & Directed edges (multigraph)
- Properties at vertices and edges
- behave like RDDs (immutable, distributed)
- Implemented as pair of typed RDDs (vertex, edge)



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

- Property Operators: map() on vertices, edges, triple $\langle e, v, e \rangle$
- Simple Structural operations:
 - subgraphs: on edge/vertex predicates, mask with existing graph,
 - reverse edges
 - combine parallel edges into one
- Join graph with RDDs
- Collect neighborhood data – follow edges

GraphX – Iterative Computation

- Many graphs problem require iteration/recursion
 - Reachability / Connected Components
 - Shortest path
 - Pagerank
- Pregel API - Vertex-centric iteration
 - All vertices collect information from their neighbors
 - All vertices perform a computation on the received data and their state
 - All vertices propagate information to their neighbors
 - There is a global synchronization barrier (bulk synchronous): next round can only start after all vertices finished
 - Iteration ends at fixed point (no more changes) or after a count
- Spark maps this graph parallelism to RDD data parallelism

Machine Learning Support

- MLLib (RDDs, focus on methods, no longer developed) and ML (DataFrames, focus on orchestration, catching up in features)
- Core Concept of ML – Pipelines:
 - Dataframes (as defined before)
 - Transformers: add new columns
 - feature transformation (e.g. text to feature vector)
 - learned models (predicted vales)
 - Estimator: fit data, generate model
- Supported operations:
 - Feature extraction and selection
 - Classification/Regression
 - Clustering
 - Recommendations
 - Model selection and tuning

Wrap up

- Big data analytics is evolving to include
 - More **complex** analytics (e.g. graphs, machine learning)
 - More **interactive** ad-hoc queries
 - More **real-time** stream processing
- Spark provides broad support and unification
- Concepts
 - Distributed shared memory
 - coarse-grained transformation
 - Lineage-based recovery
 - Type information
 - Mix of imperative and declarative expressions -> optimizer
- Ongoing arms race with other big data platforms for speed, functionality, usability