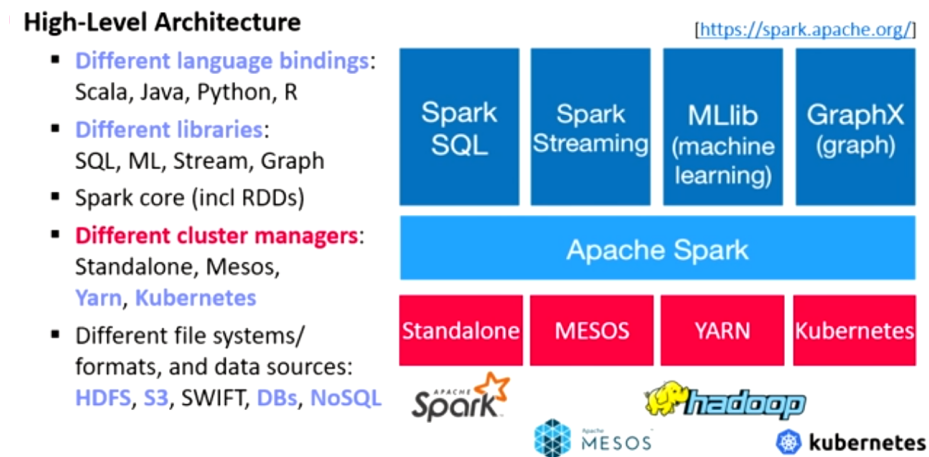


Overview

- alternative to [[MapReduce]]
 - many shared concepts
- executors with in-memory storage
 - lazy evaluation
 - fault tolerance via [[RDD]] lineage
- high performance
 - in-memory storage
 - fast job scheduling
- rich, functional high-level [[API]]
 - general computation DAGs
 - unified platform

Architecture

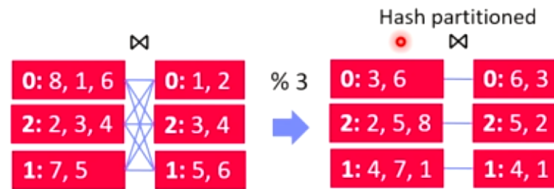


Spark Partitions

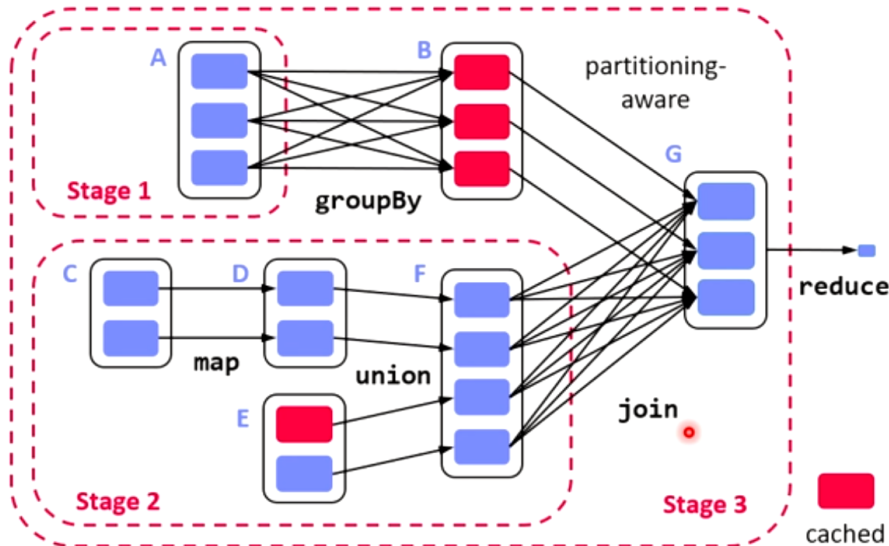
- logical key-value collections split into physical partitions 128MB
- partitions are granularity of tasks, I/O, shuffling, evictions
 - **Partitioning via Partitioners**
 - Implicitly on every data shuffling
 - Explicitly via `R.repartition(n)`
- partitioning preserving
 - operations which keep keys unchanged do not change partitions

Partitioning-Exploiting

- Join: $R3 = R1.join(R2)$
- Lookups:
 $v = C.lookup(k)$



Lazy Evaluation, Caching and Lineage



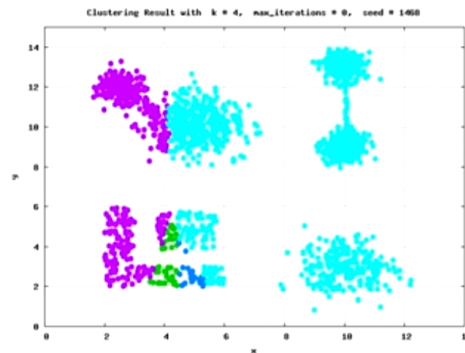
-
- example: k-means clustering

k-Means Algorithm

- Given dataset D and number of clusters k , find cluster centroids ("mean" of assigned points) that minimize within-cluster variance
- Euclidean distance: $\text{sqrt}(\text{sum}((a-b)^2))$

Pseudo Code

```
function Kmeans(D, k, maxiter) {
    C' = randCentroids(D, k);
    C = {};
    i = 0; //until convergence
    while( C' != C & i <= maxiter ) {
        C = C';
        i = i + 1;
        A = getAssignments(D, C);
        C' = getCentroids(D, A, k);
    }
    return C'
}
```



Example: K-Means Clustering in Spark

```
// create spark context (allocate configured executors)
JavaSparkContext sc = new JavaSparkContext();

// read and cache data, initialize centroids
JavaRDD<Row> D = sc.textFile("hdfs://user/mboehm/data/D.csv")
    .map(new ParseRow()).cache(); // cache data in spark executors
Map<Integer,Mean> C = asCentroidMap(D.takeSample(false, k));

// until convergence
while( !equals(C, C2) & i<=maxiter ) {
    C2 = C; i++;
    // assign points to closest centroid, recompute centroid
    Broadcast<Map<Integer,Row>> bC = sc.broadcast(C)
    C = D.mapToPair(new NearestAssignment(bC))
        .foldByKey(new Mean(0), new IncComputeCentroids())
        .collectAsMap();
}

return C;
```

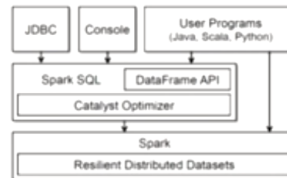
Note: Existing library algorithm

<https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/mllib/clustering/KMeans.scala>

Spark DataFrames and DataSets

Overview Spark DataFrame

- DataFrame is **distributed collection of rows** with named/typed columns
- **Relational operations** (e.g., projection, selection, joins, grouping, aggregation)
- **DataSources** (e.g., json, jdbc, parquet, hdfs, s3, avro, hbase, csv, cassandra)



example

```
logs = spark.read.format("json").open("s3://logs")
logs.groupBy(logs.user_id).agg(sum(logs.time))
    .write.format("jdbc").save("jdbc:mysql://...")
```

Dask

Overview Dask

- Multi-threaded and distributed operations for arrays, bags, and dataframes
- `dask.array`: list of numpy n-dim arrays
- `dask.dataframe`: list of pandas data frames
- `dask.bag`: unordered list of tuples (second order functions)
- Local and distributed schedulers: threads, processes, YARN, Kubernetes, containers, HPC, and cloud, GPUs



Execution

- Lazy evaluation
- Limitation: requires static size inference
- Triggered via `compute()`

```
import dask.array as da

x = da.random.random(
    (10000, 10000), chunks=(1000, 1000))
y = x + x.T
y.persist() # cache in memory
z = y[:, :2, 5000:].mean(axis=1) # colMeans
ret = z.compute() # returns NumPy array
```