UC San Diego

DSC 102
Systems for Scalable Analytics

Rod Albuyeh

Topic 4: Dataflow Systems

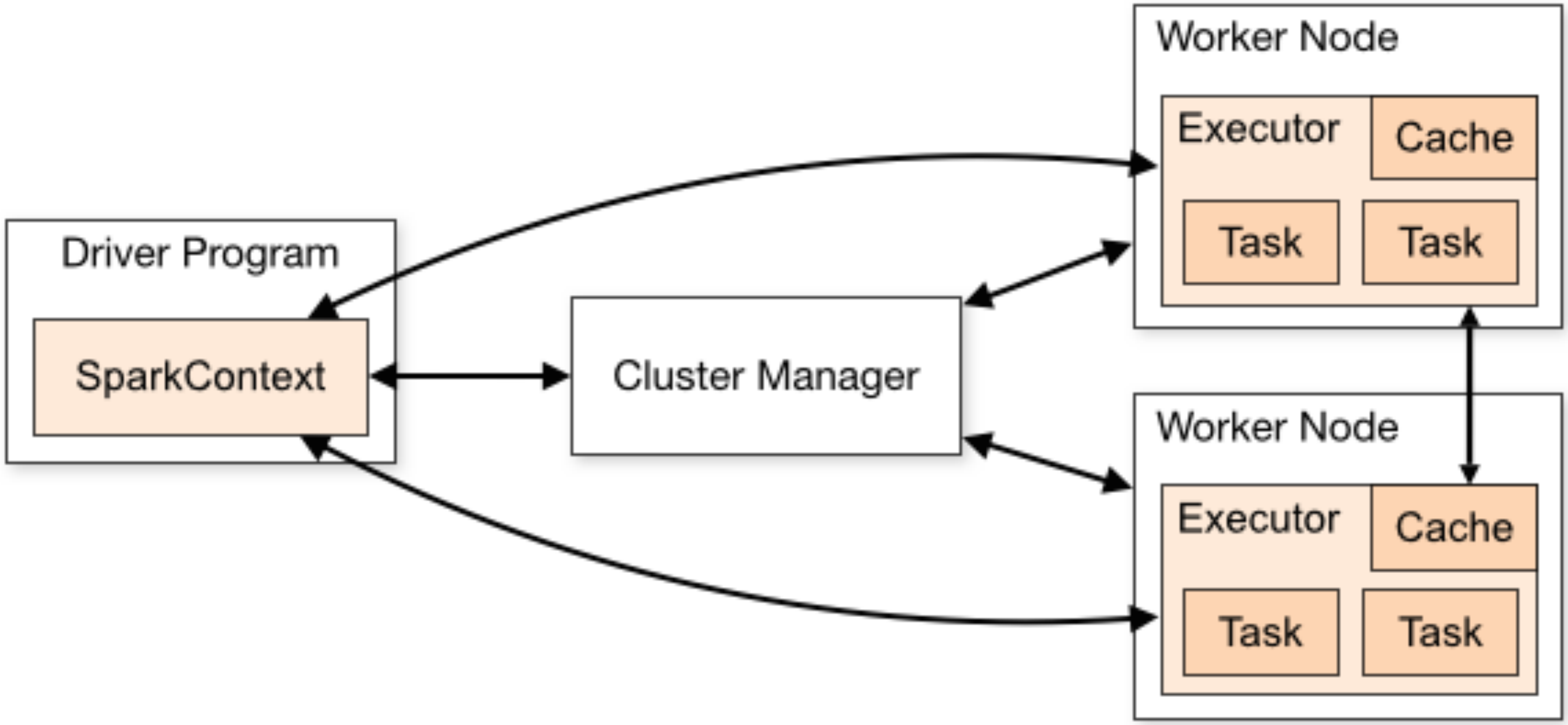
Outline

- Beyond RDBMSs: A Brief History
- MapReduce/Hadoop Craze
- Spark and Dataflow Programming
- Scalable BGD with MapReduce/Spark
- Dataflow Systems vs Task-Parallel Systems

Apache Spark

- **Dataflow programming** model (subsumes most of Relational Algebra; MR)
  - Inspired by Python Pandas style of chaining functions
  - Unified storage of relations, text, etc.; custom programs
  - Custom design (and redesign) from scratch
- Tons of sponsors, gazillion bucks, unbelievable hype!
- **Key idea vs Hadoop**: exploit distributed memory to cache data
- **Key novelty vs Hadoop**: lineage-based fault tolerance
- Open-sourced to Apache; commercialized as Databricks

Distributed Architecture of Spark



https://spark.apache.org/docs/latest/cluster-overview.html

Resilient Distributed Datasets

Key concept in Spark.

- RDD has been the primary user-facing API in Spark since its inception. At the core an RDD is an immutable distributed collection of elements of your data,
  - partitioned across nodes in your cluster
  - that can be operated in parallel with a low-level API that offers transformations and actions.
- Good for dataset low-level transformation, actions and control.
- Good for unstructured data.
- Good for functional programming data manipulation.
- Not recommended for imposing a schema on your data.
- Lacks some optimization and performance benefits

Spark's Dataflow Programming Model

Transformations are relational ops, MR, etc. as functions

Actions are what force computation; aka ***lazy evaluation***

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$sample(fraction : \mathrm{Float})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$ (Deterministic sampling)
	$groupByKey()$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, Seq[V])]}$
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$union()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[T]}) \Rightarrow \mathrm{RDD[T]}$
	$join()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (V, W))]}$
	$cogroup()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (Seq[V], Seq[W]))]}$
	$crossProduct()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[U]}) \Rightarrow \mathrm{RDD[(T, U)]}$
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, W)]}$ (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
Actions	$count()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Long}$
	$collect()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Seq[T]}$
	$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{T}$
	$lookup(k : \mathrm{K})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{Seq[V]}$ (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.*, HDFS

[Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In NSDI 2012]

Word Count Example in Spark

Spark Resilient Distributed Dataset (RDD) API

available in Python, Scala, Java, and R

```python
text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

```scala
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

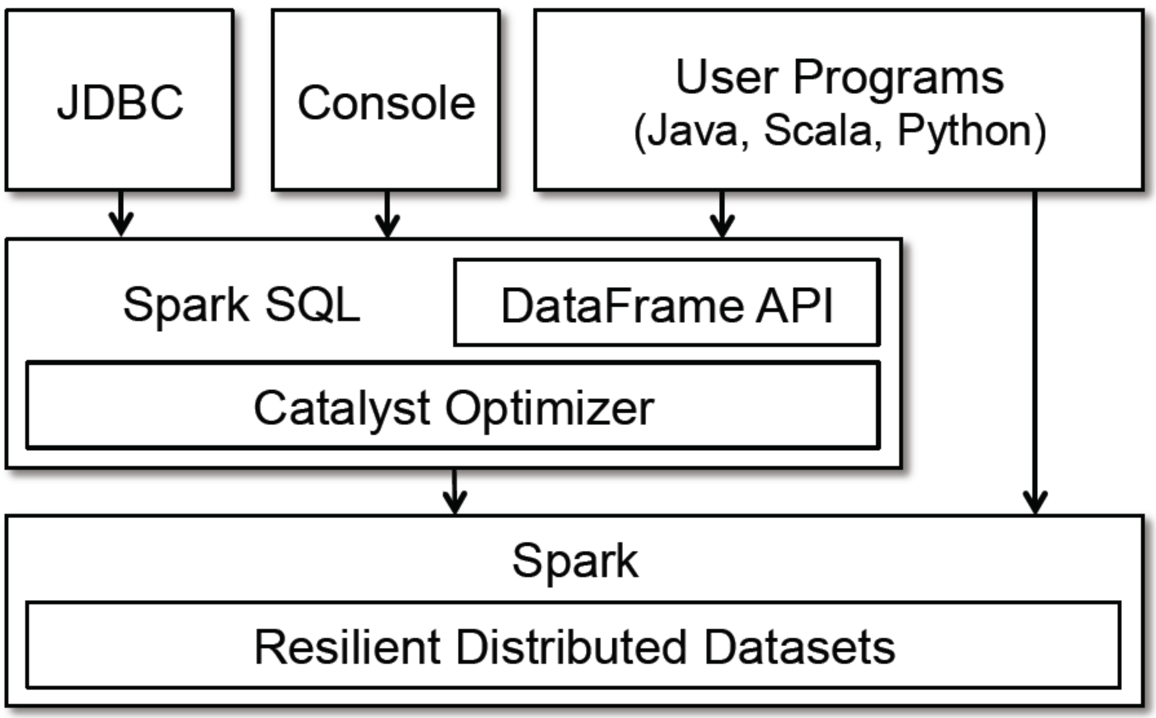
```java
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaPairRDD<String, Integer> counts = textFile
    .flatMap(s -> Arrays.asList(s.split(" ")).iterator())
    .mapToPair(word -> new Tuple2<>(word, 1))
    .reduceByKey((a, b) -> a + b);
counts.saveAsTextFile("hdfs://...");
```

Spark DataFrame API of SparkSQL offers an SQL interface

Can also interleave SQL with DF-style function chaining!

Spark DF API and SparkSQL

- Databricks now recommends SparkSQL/DataFrame API; avoid RDD API unless really needed!
- **Key Reason:** Automatic query optimization becomes more feasible



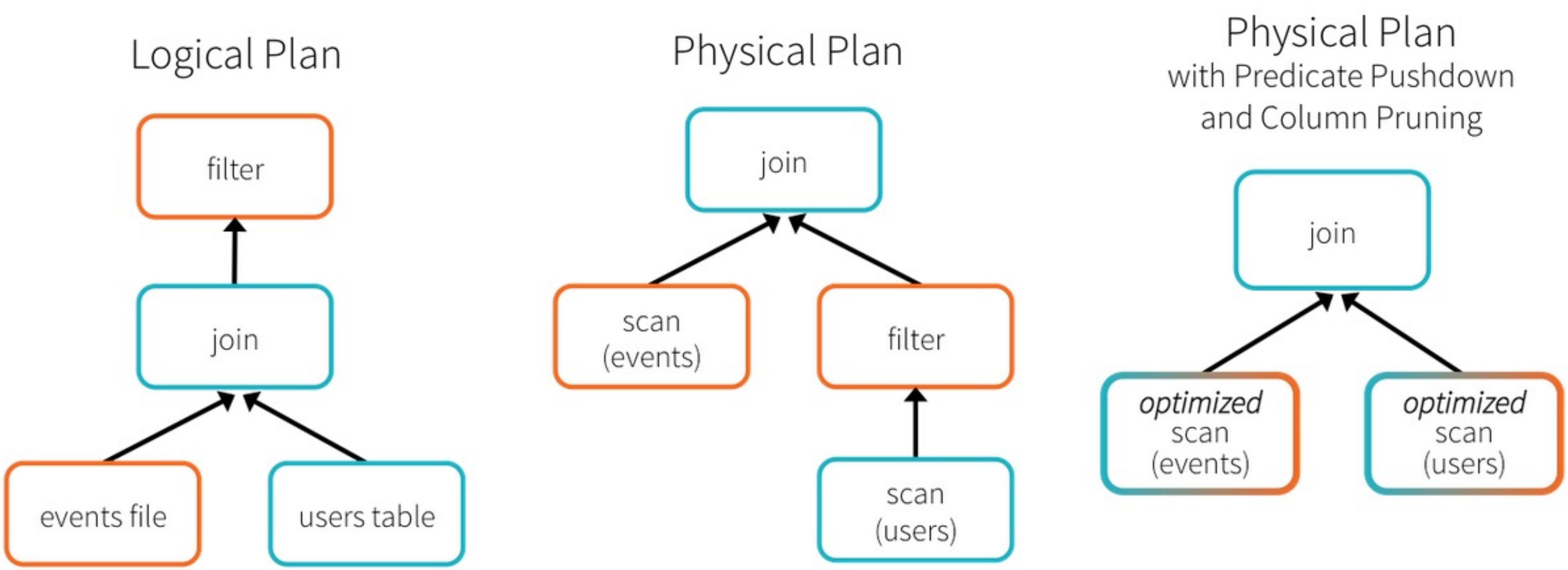
Query Optimization in Spark

- Common automatic query optimizations (from RDBMS world) are now performed in Spark's Catalyst optimizer:
- **Projection pushdown:**
  - Drop unneeded columns early on
- **Selection pushdown:**
  - Apply predicates close to base tables
- **Join order optimization:**
  - Not all joins are equally costly
- Fusing of aggregates
- …

CSE 132C has more on relational query optimization

Query Optimization in Spark

```
def add_demographics(events):
   u = sqlCtx.table("users")                      # Load partitioned Hive table
   events \
     .join(u, events.user_id == u.user_id) \      # Join on user_id
     .withColumn("city", zipToCity(df.zip))       # Run udf to add city column
events = add_demographics(sqlCtx.load("/data/events", "parquet"))
training_data = events.where(events.city == "New York").select(events.timestamp).collect()
```



Databricks is building yet another parallel RDBMS! :)

Reinventing the Wheel?

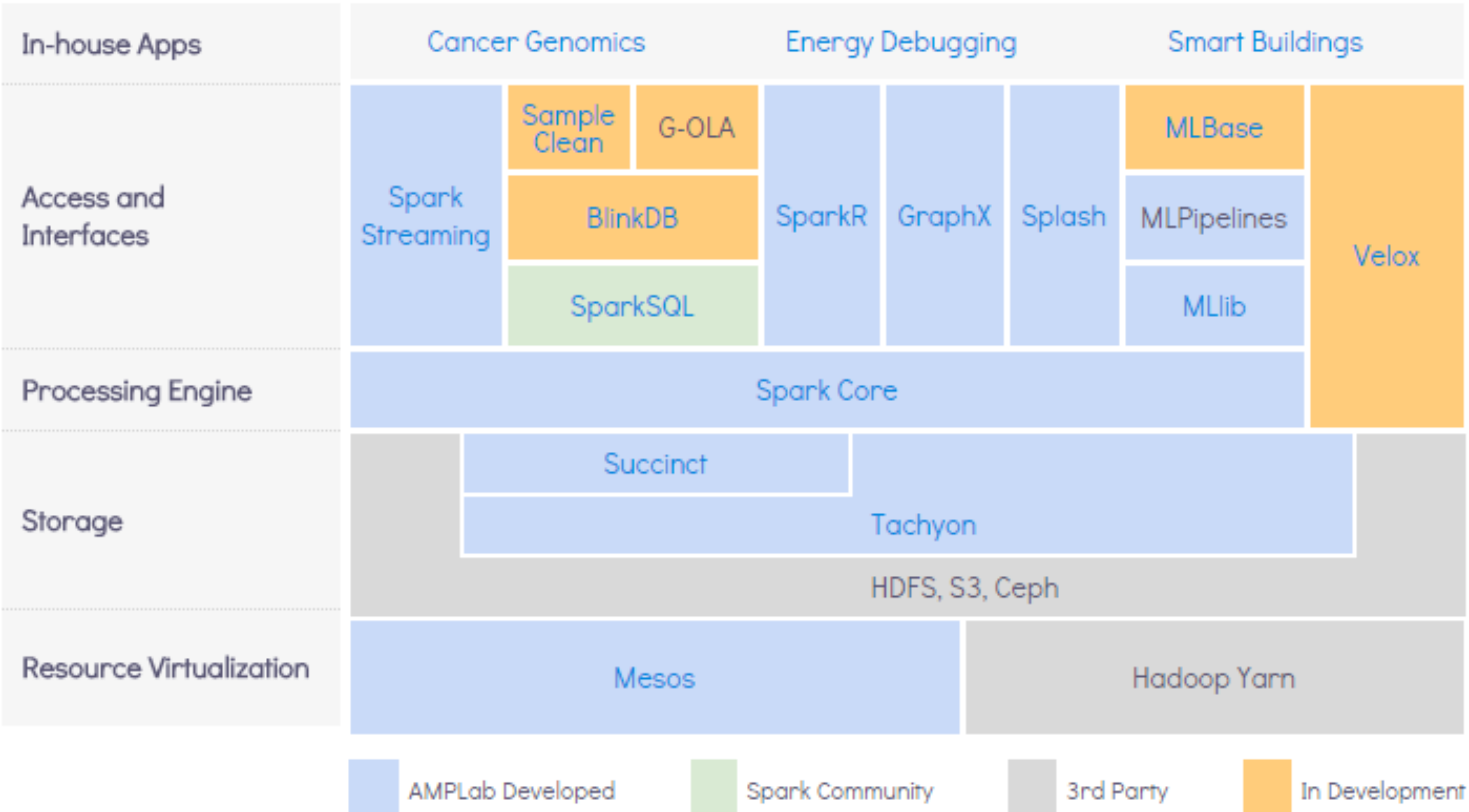
Comparing Spark's APIs

A rough comparison of
RDD, DataFrames and Koalas (databricks pandas-like module)

	RDD	DataFrame	Koalas
Abstraction Level	Low	High	High
Named Columns	No	Yes	Yes
Support for Query Optimization	No	Yes	Yes
Programming Mode	map-reduce	Dataflow, SQL	Pandas-like
Best suited for	Unstructured data Low-level ops Folks who like func. PLs and MapReduce	Structured data High-level ops Folks who know SQL, Python, R	Structured data Lower barrier to entry for folks who only know Pandas or Dask

DSC 291 offers more about Spark programming

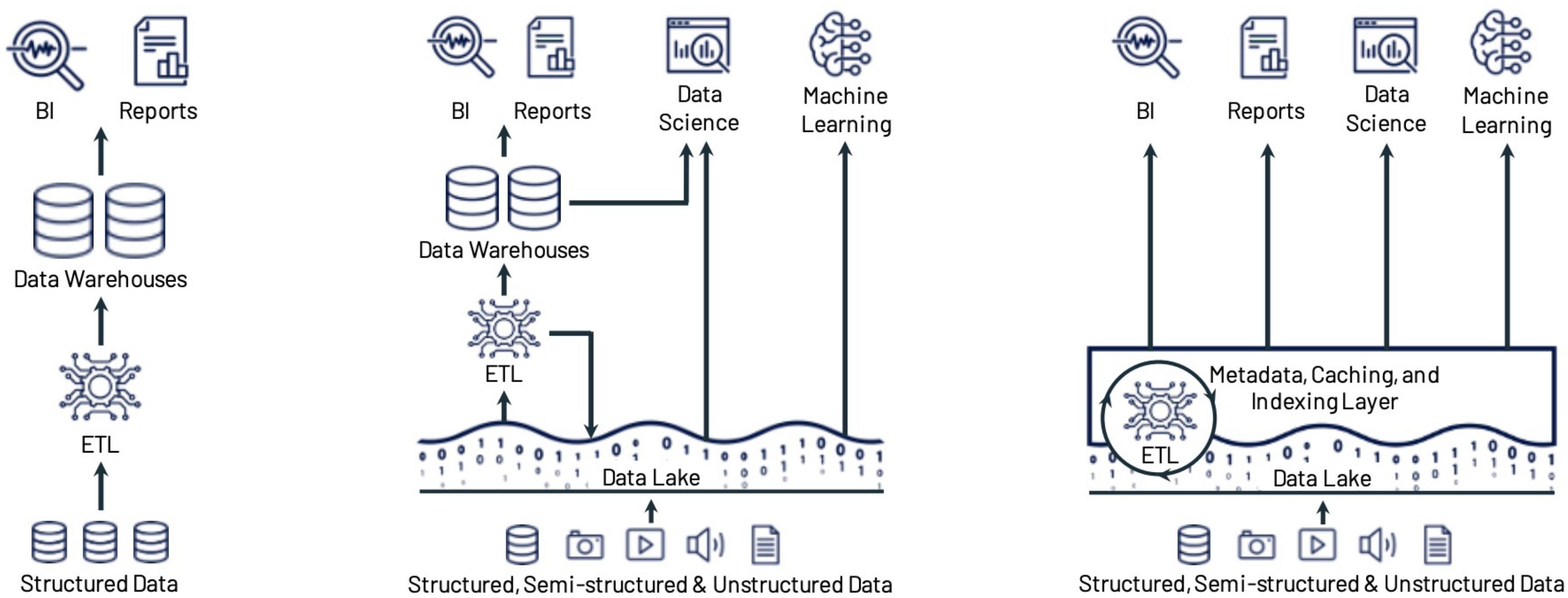
Spark-based Ecosystem of Tools



The Berkeley Data Analytics Stack (BDAS)

New Paradigm of Data "Lakehouse"

- **Data "Lake":** *Loose coupling* of data file format and data/query processing stack (vs RDBMS's tight coupling); many frontends



(a) First-generation platforms. (b) Current two-tier architectures. (c) Lakehouse platforms.

If interested, check out this vision paper on the future of data lakes and data lakehouses: http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

References and More Material

- **MapReduce/Hadoop:**
  - MapReduce: Simplified Data Processing on Large Clusters. Jeffrey Dean and Sanjay Ghemawat. In OSDI 2004.
  - More Examples: http://bit.ly/2rkSRj8
  - Online Tutorial: http://bit.ly/2rS2B5j

- **Spark:**
  - Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. Matei Zaharia and others. In NSDI 2012.
  - More Examples: http://bit.ly/2rhkhEp, http://bit.ly/2rkT8Tc
  - Online Guide: https://spark.apache.org/docs/2.1.0/sql-programming-guide.html

Outline

- Beyond RDBMSs: A Brief History
- MapReduce/Hadoop Craze
- Spark and Dataflow Programming
- Scalable BGD with MapReduce/Spark
- Dataflow Systems vs Task-Parallel Systems

Example: Batch Gradient Descent

$$\nabla L(\mathbf{w}^{(k)}) = \sum_{i=1}^{n} \nabla l(y_i, f(\mathbf{w}^{(k)}, x_i))$$

- Very similar to algebraic SQL; vector addition
- **Input Split**: Shard table tuple-wise
- **Map():**
  - On tuple, compute per-example gradient; add these across examples in shard; emit partial sum with single dummy key
- **Reduce():**
  - Only one global dummy key, Iterator has partial gradients; just add all those to get full batch gradient

Outline

- Beyond RDBMSs: A Brief History
- MapReduce/Hadoop Craze
- Spark and Dataflow Programming
- More Scalable ML with MapReduce/Spark
- Dataflow Systems vs Task-Parallel Systems

Dataflow Systems vs Task-Par. Sys.

- **Pros:**

- **Cons:**

Specific to Spark vs Dask?

- **Pros:**

- **Cons:**

We'll talk about deployment in some detail next week, but how does this stuff fit in with model deployment?

Optional: More complex examples of MapReduce usage to scale ML

Not included in syllabus

Primer: K-Means Clustering

- **Basic Idea:** Identify clusters based on Euclidean distances; formulated as an optimization problem
- **Llyod's algorithm:** Most popular heuristic for K-Means
- **Input:** *n x d* examples/points
- **Output:** *k* clusters and their centroids

1. Initialize *k* centroid vectors and point-cluster ID assignment
2. **Assignment step:** Scan dataset and assign each point to a cluster ID based on which centroid is *nearest*
3. **Update step:** Given new assignment, scan dataset again to recompute centroids for all clusters
4. Repeat 2 and 3 until convergence or fixed # iterations

K-Means Clustering in MapReduce

- **Input Split:** Shard the table tuple-wise
  - Assume each tuple/example/point has an *ExampleID*
- Need 2 jobs! 1 for Assignment step, 1 for Update step
- 2 external data structures needed for both jobs:
  - Dense matrix A: *k x d* centroids; ultra-sparse matrix B: *n x k* assignments
  - A and B first broadcast to all Mappers via HDFS; Mappers can read small data directly from HDFS files
  - Job 1 read A and creates new B
  - Job 2 reads B and creates new A

K-Means Clustering in MapReduce

- A: *k x d* centroid matrix; B: *n x k* assignment matrix
- **Job 1 Map():** Read A from HDFS; compute point's distance to all *k* centroids; get nearest centroid; emit new assignment as output pair (PointID, ClusterID)
- No Reduce() for Job 1; new B now available on HDFS
- **Job 2 Map():** Read B from HDFS; look into B and see which cluster point got assigned to; emit point as output pair (ClusterID, point vector)
- **Job 2 Reduce():** Iterator has all point vectors of a given ClusterID; add them up and divide by count; got new centroid; emit output pair as (ClusterID, centroid vector)