UC San Diego

DSC 102
Systems for Scalable Analytics

Haojian Jin

Topic 4: Dataflow Systems

Spark Book; Chapter 2.2 of MLSys Book

Tentative Course Schedule

Week	Topic
	Basics of Machine Resources: Computer Organization
	Basics of Machine Resources: Operating Systems
4	Basics of Cloud Computing
4-5	Parallel and Scalable Data Processing: Parallelism Basics
	Midterm Exam on TBD
6-7	Parallel and Scalable Data Processing: Scalable Data Access
7-8	Parallel and Scalable Data Processing: Data Parallelism
9	Dataflow Systems
10	ML Model Building Systems
11	**Final Exam on Dec 15**

Systems Principles

Scalability Principles

Scalable Analytics Systems

Remaining quarter

- Last lecture: Review for final exam.
- Final exam.
- Triton Test Center
- 80%+ CAPE response for class yields => 1% collective boost to final score.
- 90%+ CAPE response for class yields => 2% collective boost to final score.

Outline

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

Q: *How to shield users from needing to think about moving raw pages between disk/RAM/network to scale data-intensive programs?*

Parallel RDBMSs

- Parallel RDBMSs are highly successful and widely used
- Typically shared-nothing data parallelism
- Optimized **runtime performance** + enterprise-grade features:
  - ANSI SQL & more
  - Business Intelligence (BI) dashboards/APIs
  - Transaction management; crash recovery
  - Indexes, auto-tuning, etc.

Q: *So, why did people need to go beyond parallel RDBMSs?*

Ad: Take CSE 132C for more on parallel RDBMSs

Beyond RDBMSs: A Brief History

- DB folks got blindsided by the rise of Web/Internet giants

- 4 new concerns of Web giants vs RDBMSs built for enterprises:
  - **Developability:** Custom data models and computations hard to program on SQL/RDBMSs; need for simpler APIs
  - **Fault Tolerance:** Need to scale to 1000s of machines; need for graceful handling of worker failure
  - **Elasticity:** Need to be able to easily upsize or downsize cluster size based on workload
  - **Cost:** Commercial RDBMSs licenses too costly; hired own software engineers to build custom new systems

A new breed of parallel data systems called **Dataflow Systems** jolted the DB folks from being smug and complacent!

What is MapReduce?

- A programming model for parallel programs on **sharded data** + **distributed system** architecture
- **Map** and **Reduce** are terms from functional PL; software/data/ML engineer implements logic of Map, Reduce
- System handles data distribution, parallelization, fault tolerance, etc. under the hood
- Created by Google to solve “simple” data workload: index, store, and search the Web!
- Google’s engineers started with MySQL! Abandoned it due to reasons listed earlier (developability, fault tolerance, elasticity, etc.)

https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

Programming Language Background

- Declarative?
- While coding, you will not be interested in how you want the job done. The focus is on what result you want to obtain. (e.g., SQL*)
- Imperative?
- The micromanaging boss who gives instructions down to the final detail. (e.g., Python*)
- Object-Oriented Programming?
- Organizes data and the software structure based on the concept of classes and objects.
- Functional?
- A paradigm of building computer programs using expressions and functions without mutating state and data

What is MapReduce?

- **Standard example**: count word occurrences in a doc corpus
- **Input**: A set of text documents (say, webpages)
- **Output**: A dictionary of unique words and their counts

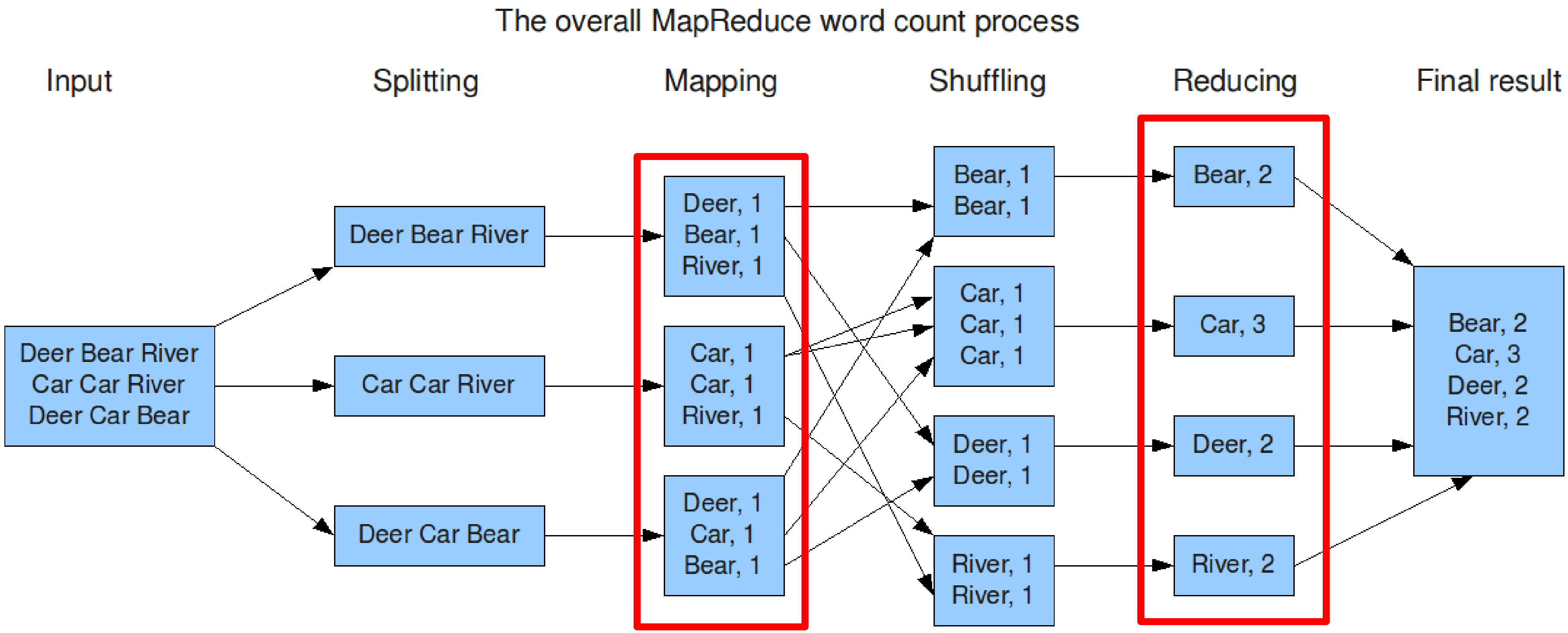
```
function map (String docname, String doctext) :
        for each word w in doctext :
                emit (w, 1)
function reduce (String word, Iterator partialCounts) :
        sum = 0
        for each pc in partialCounts :
                sum += pc
        emit (word, sum)
```

Hmmm, sounds suspiciously familiar! :)

Part of MapReduce API

How MapReduce Works

Parallel flow of control and data during MapReduce execution:



Under the hood, each **Mapper** and **Reducer** is a separate process; Reducers face barrier synchronization (BSP)

Fault tolerance achieved using **data replication**

Benefits and Catch of MapReduce

- **Goal:** High-level *functional* ops to simplify data-intensive programs
- **Key Benefits:**
  - Map() and Reduce() are highly general; any data types/structures; great for ETL, text/multimedia
  - Native scalability, large cluster parallelism
  - System handles fault tolerance automatically
  - Decent FOSS stacks (Hadoop and later, Spark)
- **Catch:** Users must learn "art" of casting program as MapReduce
  - Map operates record-wise; Reduce aggregates globally
  - But MR libraries now available in many PLs: C/C++, Java, Python, R, Scala, etc.

Abstract Semantics of MapReduce

- **Map():** Process one “record” at a time *independently*
  - A record can physically *batch* multiple data examples/tuples
  - Dependencies across Mappers *not* allowed
  - *Emit* 1 or more key-value pairs as output(s)
  - Data types of input vs. output can be different
- **Reduce():** Gather all Map outputs across workers sharing same key into an Iterator (list)
  - Apply *aggregation* function on Iterator to get final output(s)
- **Input Split:**
  - Physical-level shard to batch many records to one file “block” (HDFS default: 128MB?)
  - User/application can create *custom* Input Splits

Emulate MapReduce in SQL?

Q: How would you do the word counting in RDBMS / in SQL?

- First step: **Transform** text docs into relations and load:

  Part of the ETL stage

  Suppose we pre-divide each doc into words w/ schema:

  DocWords (DocName, Word)

- Second step: a single, simple SQL query!

```
SELECT      Word, COUNT (*)
FROM        DocWords
GROUP BY    Word
[ORDER BY   Word]
```

Parallelism, scaling, etc. done by RDBMS under the hood

More MR Examples: Select Operation

- **Input Split**:
  - Shard table tuple-wise
- **Map():**
  - On tuple, apply selection condition; if satisfies, emit KV pair with dummy key, entire tuple as value
- **Reduce():**
  - Not needed! No cross-shard aggregation here
- These kinds of MR jobs are called "**Map-only**" jobs

More MR Examples: Simple Agg.

- Suppose it is *algebraic* aggregate (SUM, AVG, MAX, etc.)
- **Input Split**:
  - Shard table tuple-wise
- **Map():**
  - On agg. attribute, compute incr. stats; emit pair with single global dummy key and incr. stats as value
- **Reduce():**
  - Since only one global dummy key, Iterator has *all* suff. stats to unify into global agg.

More MR Examples: GROUP BY Agg.

- Assume it is *algebraic* aggregate (SUM, AVG, MAX, etc.)
- **Input Split**:
  - Shard table tuple-wise
- **Map():**
  - On agg. attribute, compute incr. stats; emit pair with *grouping attribute as key* and stats as value
- **Reduce():**
  - Iterator has all suff. stats *for a single group*; unify those to get result for that group
  - Different reducers will output different groups' results

More MR Examples: Matrix Norm

- Assume it is *algebraic* aggregate ($L_{p,q}$ norm)
- Very similar to simple SQL aggregates
- **Input Split**:
  - Shard table tuple-wise
- **Map():**
  - On agg. attribute, compute incr. stats; emit pair with single global dummy key and stats as value
- **Reduce():**
  - Since only one global dummy key, Iterator has *all* suff. stats to unify into global agg.

What is Hadoop then?

- FOSS system implementation with MapReduce as prog. model and HDFS as filesystem
- MR user API; input splits, data distribution, shuffling, fault tolerances handled by Hadoop under the hood
- Exploded in popularity in 2010s: 100s of papers, 10s of products
- A “revolution” in scalable+parallel data processing that took the DB world by surprise
- But nowadays Hadoop (API) largely supplanted by Spark
- HDFS is still common in many companies.

NB: Do not confuse MR for Hadoop or vice versa!

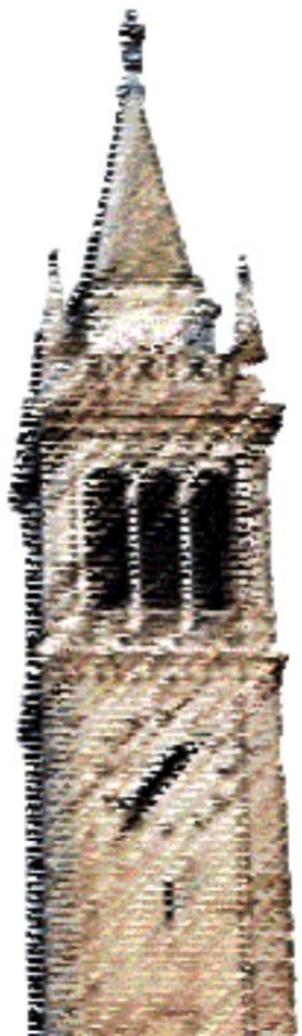
Outline

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

Apache Spark

- **Dataflow programming** model (subsumes most of RA; MR)
  - Inspired by Python Pandas style of chaining functions
  - Unified storage of relations, text, etc.; custom programs
  - System impl. (re)designed 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

An Architecture for Fast and General Data Processing on Large Clusters



Matei Zaharia

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-12
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-12.html

February 3, 2014

As a result, a wide range of new programming models have been designed for clusters. At first, Google's MapReduce [36] presented a simple and general model for batch processing that automatically handles faults. However, MapReduce was found poorly suited for other types of workloads, leading to a wide range of *specialized* models that differed significantly from MapReduce. For example, at Google, Pregel [72] offers a bulk-synchronous parallel (BSP) model for iterative graph algorithms; F1 [95] runs fast, but non-fault-tolerant, SQL queries; and MillWheel [2] supports continuous stream processing. Outside Google, systems like Storm [14], Impala [60], Piccolo [86] and GraphLab [71] offer similar models. With new models continuing to be implemented every year, it seems that cluster computing is bound to require an array of point solutions for different tasks.

we show that a simple extension to MapReduce called Resilient Distributed Datasets (RDDs), which just adds efficient data sharing 1 primitives, greatly increases its generality. The resulting architecture has several key advantages over current systems:

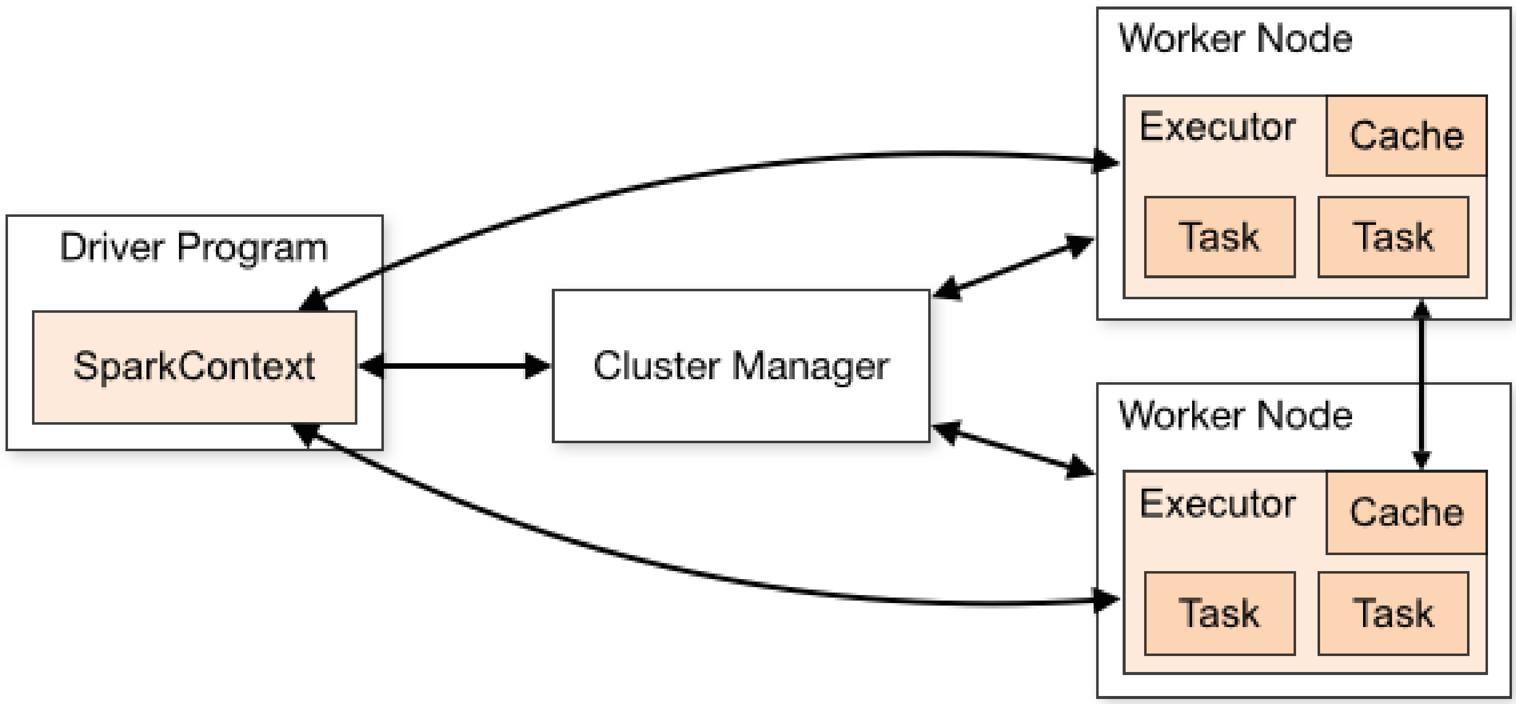
Thesis Statement: *A common execution model based on resilient distributed datasets can efficiently support diverse distributed computations.*

https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-12.pdf

2.2.4 Applications Not Suitable for RDDs

As discussed in the Introduction, RDDs are best suited for batch applications that apply the same operation to all elements of a dataset. In these cases, RDDs can efficiently remember each transformation as one step in a lineage graph and can recover lost partitions without having to log large amounts of data. RDDs would be less suitable for applications that make asynchronous fine-grained updates to shared state, such as a storage system for a web application or an incremental web crawler. For these applications, it is more efficient to use systems that perform traditional update logging and data checkpointing, such as databases, RAMCloud [81], Percolator [85] and Piccolo [86]. Our goal is to provide an efficient programming model for batch analytics and leave these asynchronous applications to specialized systems. Nonetheless, Chapter 5 covers a few possible approaches for integrating these types of applications with the RDD model, such as batching updates.

Distributed Architecture of Spark



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

Spark's Dataflow Programming Model

Transformations are relational ops, MR, etc. as functions

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

Transformations	$\mathit{map}(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] ⇒ RDD[U]
	$\mathit{filter}(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] ⇒ RDD[T]
	$\mathit{flatMap}(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] ⇒ RDD[U]
	$\mathit{sample}(\mathit{fraction} : \mathrm{Float})$	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	$\mathit{groupByKey}()$	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	$\mathit{reduceByKey}(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	$\mathit{union}()$	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	$\mathit{join}()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	$\mathit{cogroup}()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	$\mathit{crossProduct}()$	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	$\mathit{mapValues}(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	$\mathit{sort}(c : \mathrm{Comparator[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	$\mathit{partitionBy}(p : \mathrm{Partitioner[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	$\mathit{count}()$	:	RDD[T] ⇒ Long
	$\mathit{collect}()$	:	RDD[T] ⇒ Seq[T]
	$\mathit{reduce}(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] ⇒ T
	$\mathit{lookup}(k : \mathrm{K})$	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	$\mathit{save}(\mathit{path} : \mathrm{String})$	:	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 RDD API available in Python, Scala, Java, and R

```
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://...")
```

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

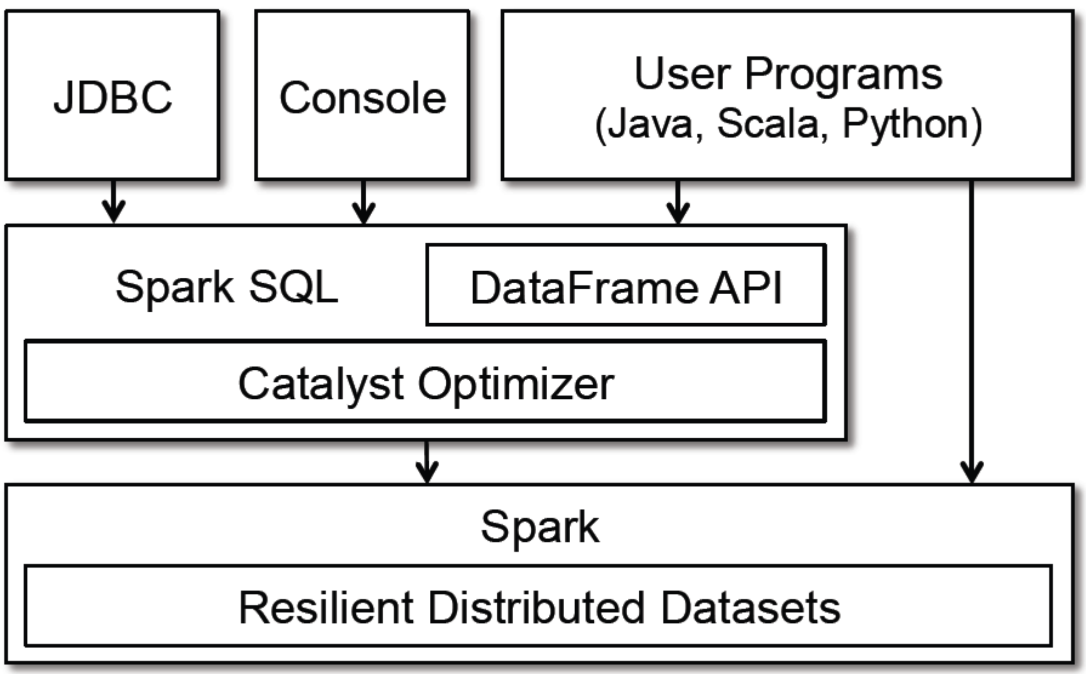
```
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaPairRDD<String, Integer> counts = textFile
    .flatMap(s -> Arrays.asList(s.split(" ")).iterator())
    .mapToPair(word -> new Tuple2<>(word, 1))
    .reduceByKey((a, b) -> a + b);
counts.saveAsTextFile("hdfs://...");
```

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
  - AKA (painfully) re-learn 40 years of database systems research! :)



Spark SQL: Relational Data Processing in Spark. In SIGMOD 2015.

Query Optimization in Spark

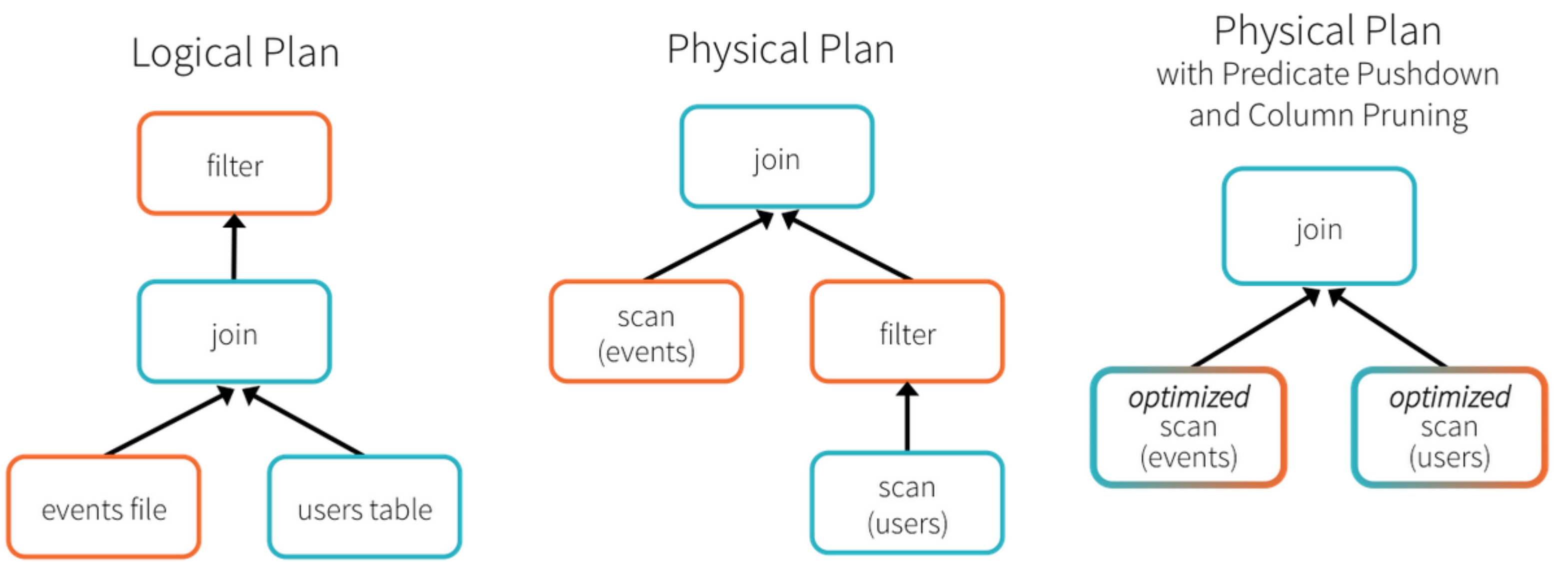
- Common automatic query optimizations (from RDBMS world) are now performed in Spark's Catalyst optimizer:
- **Projection pushdown:**
  - Drop unneeded columns early on
- **Selection pushdown:**
  - Apply predicates close to base tables
- **Join order optimization:**
  - Not all joins are equally costly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

Spark SQL: Relational Data Processing in Spark. In SIGMOD 2015.

Ad: Take CSE 132C for 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! :)

[Spark SQL: Relational Data Processing in Spark](). In SIGMOD 2015.

Reinventing the Wheel?

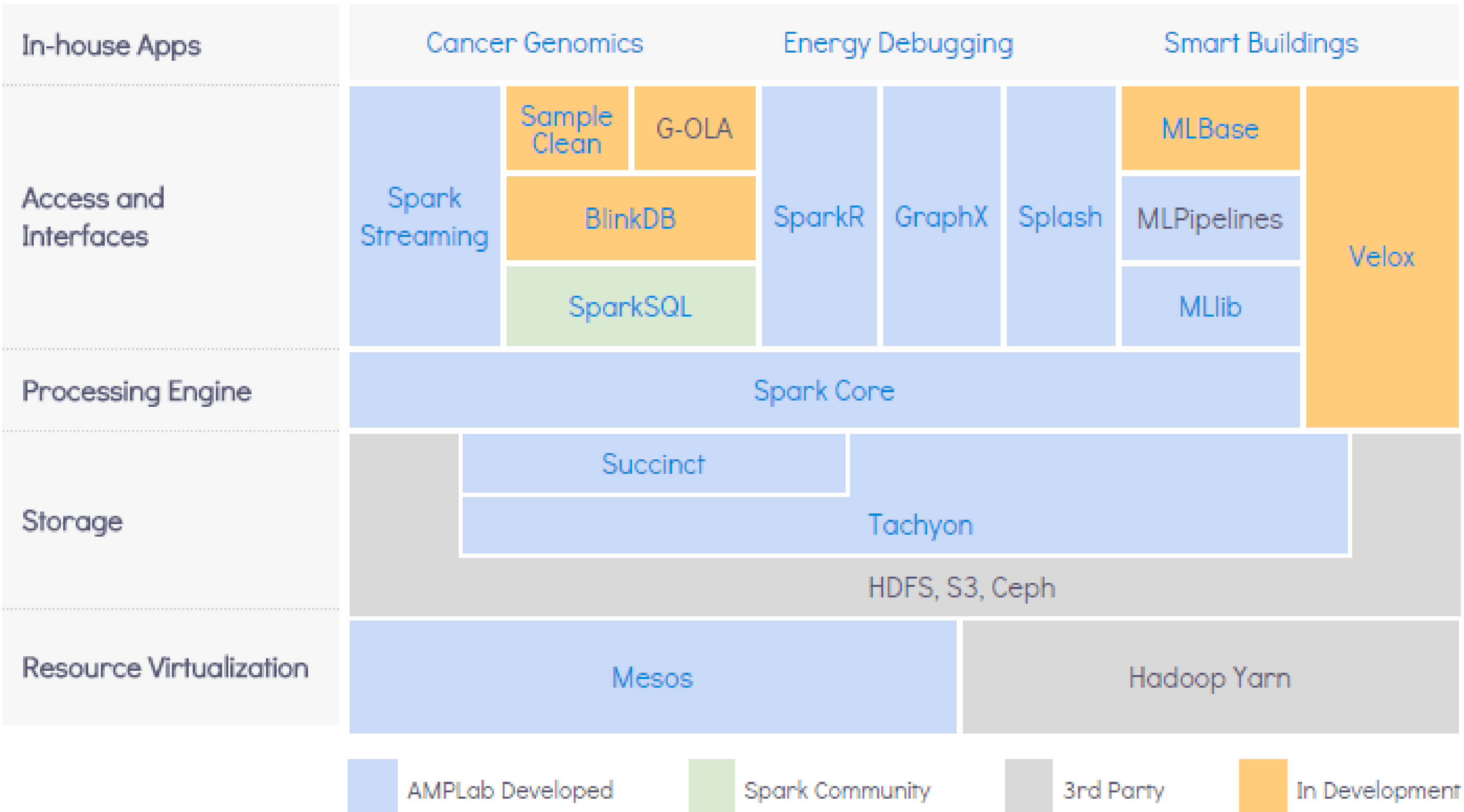
Comparing Spark's APIs

Check out TA's PA 2 slides for more on Spark APIs

	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

Ad: Take Yoav's DSC 291 to learn more Spark programming

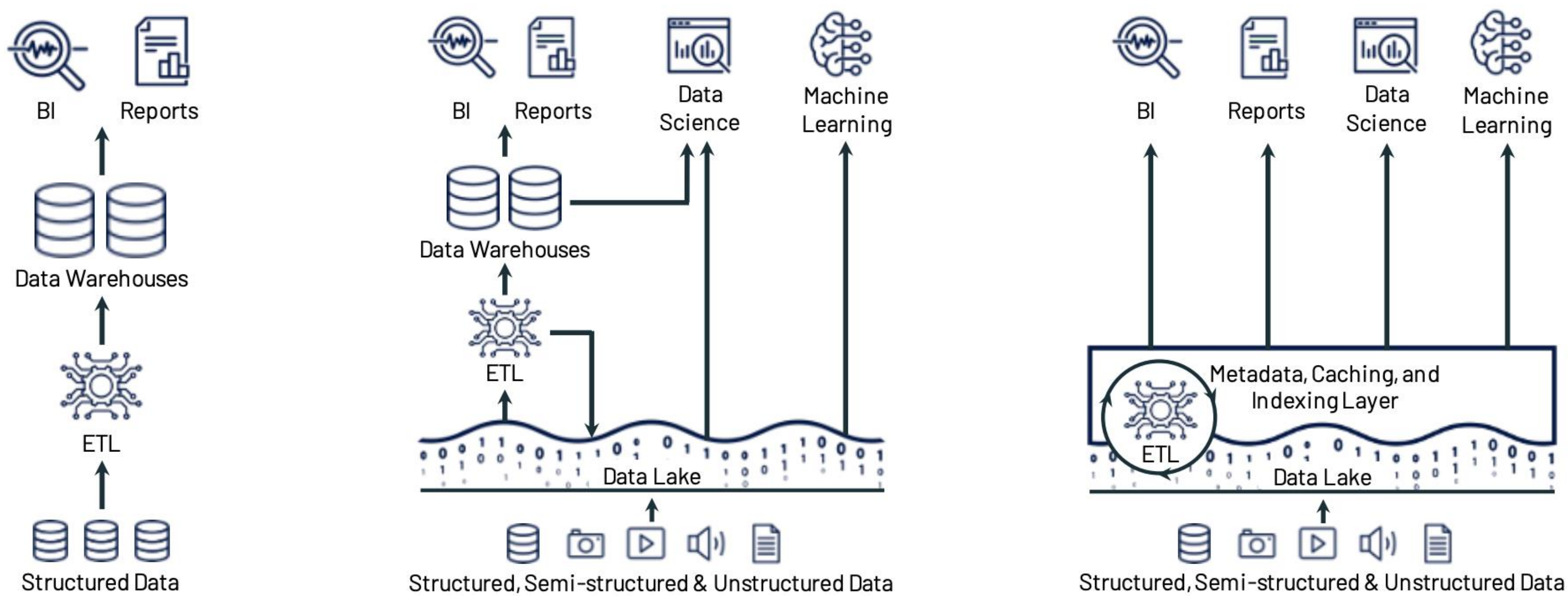
Spark-based Ecosystem of Tools



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
- Dataflow Systems vs Task-Parallel Systems

Dataflow Sys. vs. Task-Par. Sys.

- **Pros:**

  Discussion in class

- **Cons:**

  Discussion in class

More Specific to Spark vs. Dask?

- **Pros:**
  Discussion in class

- **Cons:**
  Discussion in class