

CSE 132C

Database System Implementation

Arun Kumar

Topic 6: Parallel RDBMSs and Dataflow Systems

Chapter 22 till 22.5 of Cow Book; extra references listed

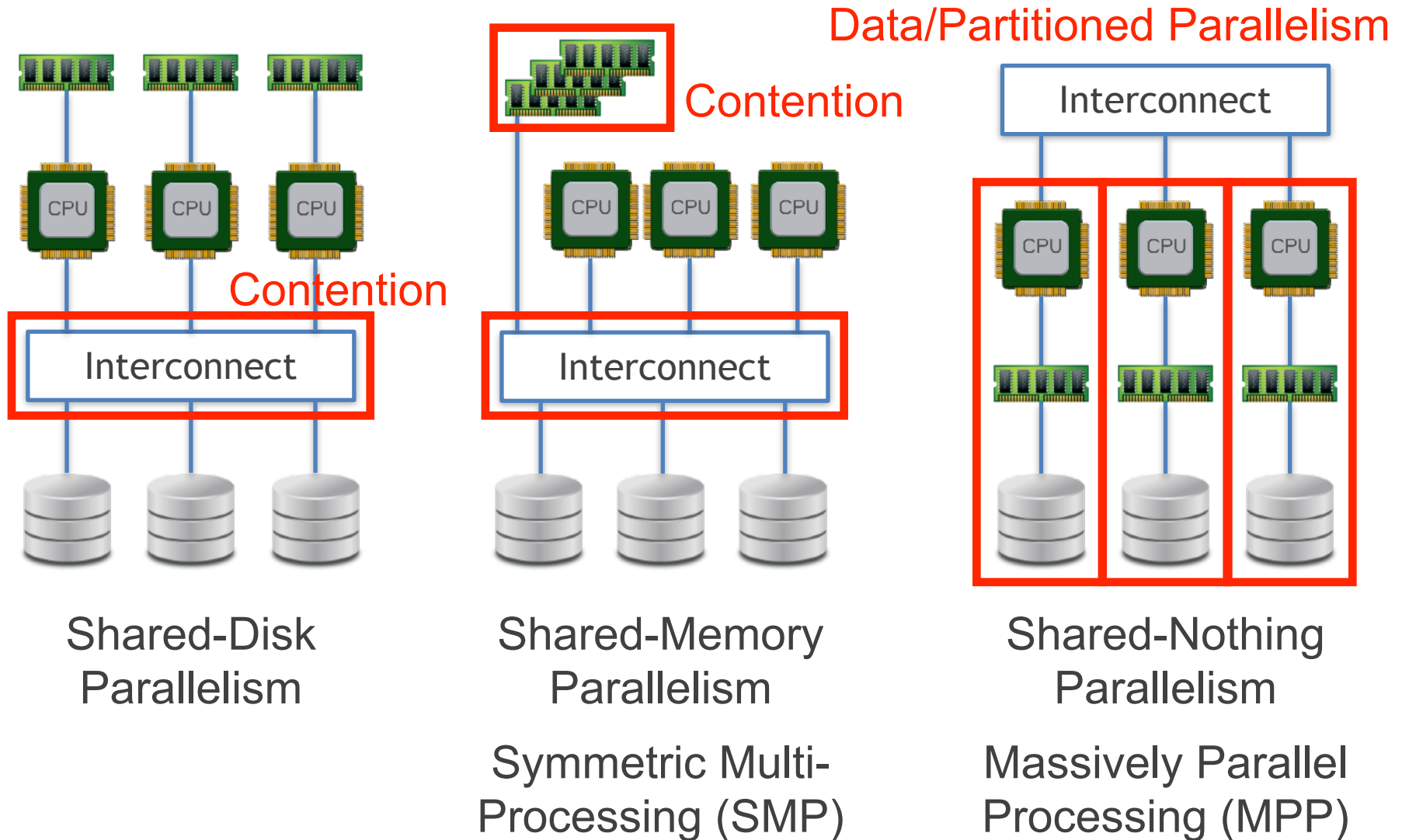
Outline

- ❖ Parallel RDBMSs
- ❖ Cloud-Native RDBMSs
- ❖ Beyond RDBMSs: A Brief History
- ❖ “Big Data” Systems aka Dataflow Systems

Parallel DBMSs: Motivation

- ❖ **Scalability:** Database is too large for a single node's disk
- ❖ **Performance:** Exploit multiple cores/disks/nodes
- ❖ ... while maintaining almost all other benefits of (R)DBMSs!

Three Paradigms of Parallelism



Shared-Nothing Parallelism

- ❖ Followed by almost all parallel RDBMSs (and “Big Data” sys.)
- ❖ 1 *manager* node orchestrates multiple *worker* nodes
- ❖ Need partitioned parallel implementations of all the relational operators we saw, parallel query proc., parallelism-aware QO

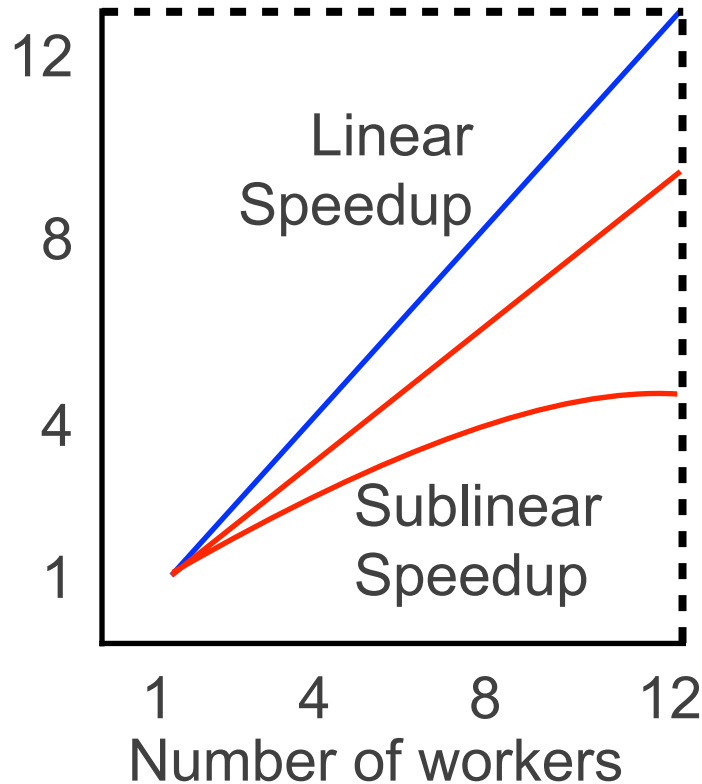
Q: If we give 10 workers (CPUs/nodes) for processing a query in parallel, will its runtime go down by a factor of 10?

It depends!

(Access patterns of the query's operators, communication of intermediate data, relative startup overhead, etc.)

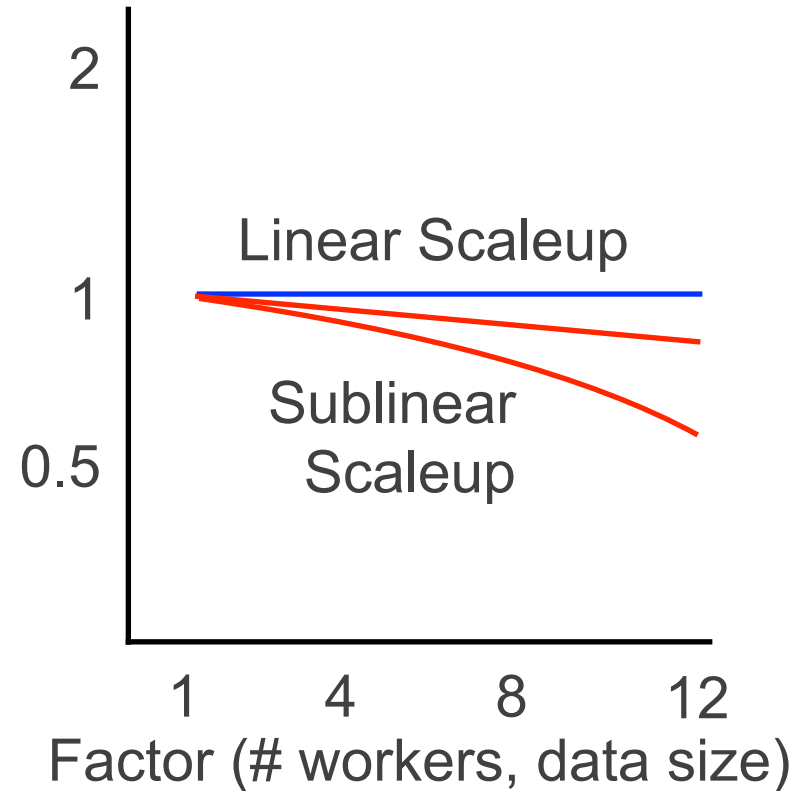
Shared-Nothing Parallelism

Runtime speedup (fixed data size)



Speedup plot / Strong scaling

Runtime speedup



Scaleup plot / Weak scaling

Q: Is superlinear speedup/scaleup possible?

Shared-Nothing Parallelism: Outline

- ❖ Data Partitioning
- ❖ Parallel Operator Implementations
- ❖ Parallel Query Optimization
- ❖ Parallel vs “Distributed” DBMSs

Data Partitioning

- ❖ A part of ETL (Extract-Transform-Load) for database
- ❖ Typically, record-wise/horizontal partitioning (aka “sharding”)
- ❖ Three common schemes (given k machines):
 - ❖ **Round-robin**: assign tuple i to machine $i \text{ MOD } k$
 - ❖ **Hashing-based**: needs partitioning attribute(s)
 - ❖ **Range-based**: needs ordinal partitioning attribute(s)
- ❖ **Tradeoffs**: Round-robin makes all queries touch all workers; hashing-based and range-based for range queries can target workers better but can face skew; hashing-based most common
- ❖ **Replication** often used for more availability, performance

Parallel Scans and Select

- ❖ **Intra-operator parallelism** is our primary focus
 - ❖ Inter-operator and inter-query parallelism also possible
- ❖ **Filescan:**
 - ❖ Trivial; worker simply scans its partition and streams it
 - ❖ Apply selection predicate (if any)
- ❖ **Indexed:**
 - ❖ Depends on data partitioning scheme and predicate
 - ❖ Same tradeoffs: Hash index vs. B+ Tree index
 - ❖ Each worker can have its own (sub-)index
 - ❖ Manager *routes* query based on matching workers

Parallel Sorting

❖ Naive algorithm:

1. Each worker sorts local partition (EMS)
2. Manager merges all locally sorted runs

❖ **Issue:** Parallelism is limited during merging phase!

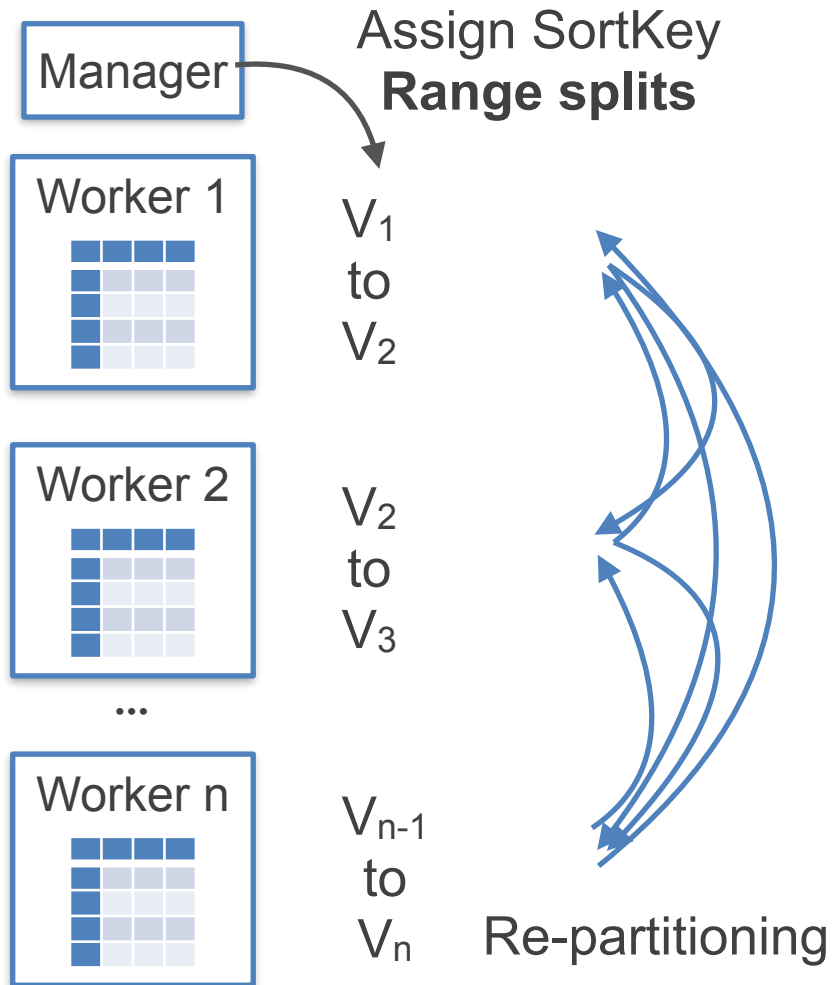
❖ Faster algorithm:

1. Scan in parallel and *range partition* data (most likely a repartitioning) based on SortKey
2. Each worker sorts local allotted range (EMS); result is globally sorted and conveniently range-partitioned

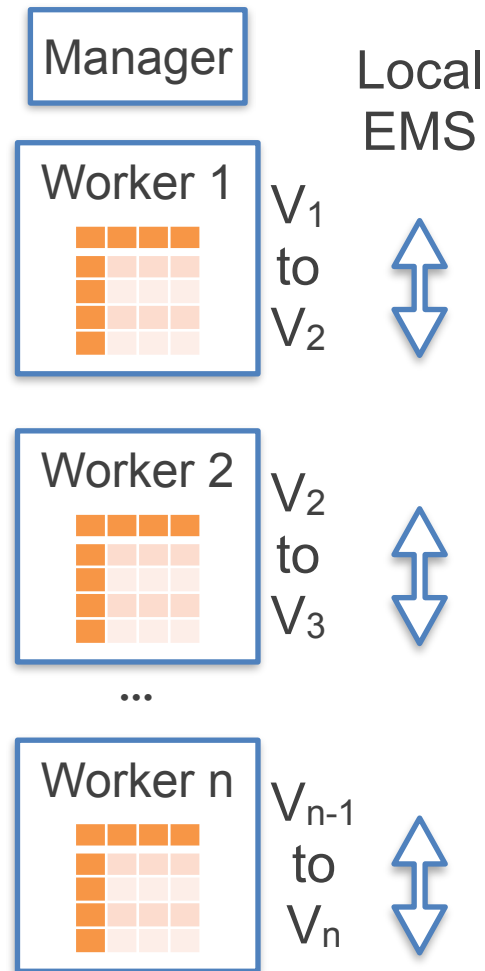
❖ **Potential Issue:** Skew in range partitions; handled by roughly estimating distribution using sampling

Parallel Sorting

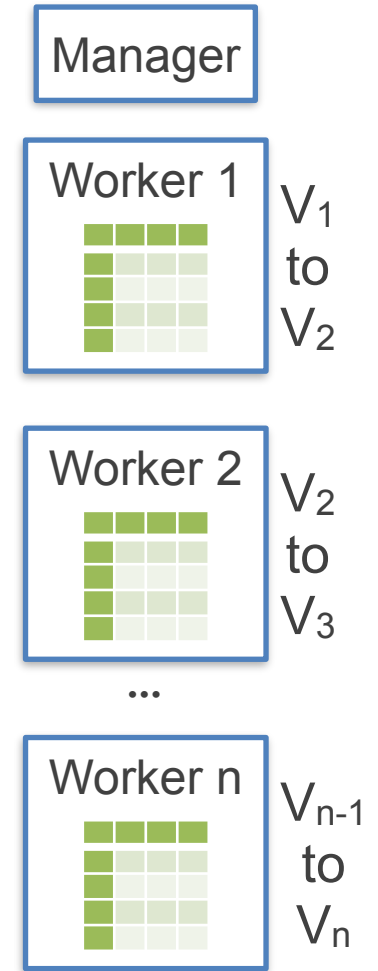
Original Partitions



Range-partitioned



Globally Sorted



Parallel Aggregates and Group By

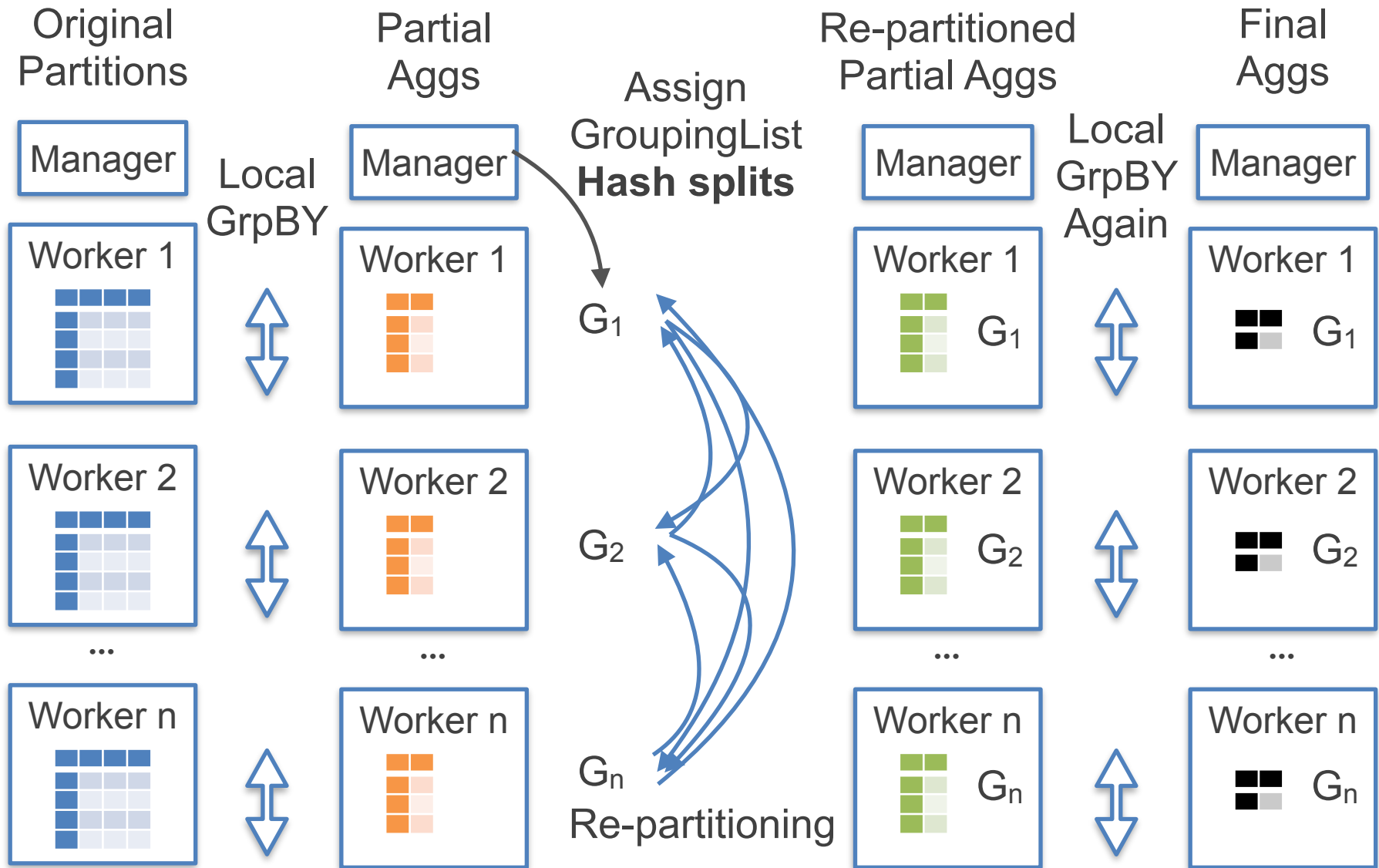
❖ Without Group By List:

- ❖ Trivial for MAX, MIN, COUNT, SUM, AVG (why?)
- ❖ MEDIAN requires parallel sorting (why?)

❖ With Group By List:

1. If AggFunc allows, pre-compute partial aggregates
2. Manager assigns each worker a set of groups (hash partition)
3. Each worker communicates its partial aggregate for a group to that group's assigned worker (aka "shuffle")
4. Each worker finishes aggregating for all its assigned groups

Parallel Group By Aggregate



Parallel Project

❖ Non-deduplicating Project:

- ❖ Trivial; pipelined with Scans/Select

❖ Deduplicating Project:

1. Each worker deduplicates its partition on ProjectionList
2. If estimated output size is small (catalog?), workers communicate their results to Manager to finish dedup.
3. If estimated output size is too large for Manager's disk, similar algorithm as Parallel Aggregate with Group By, except, there is no AggFunc computation

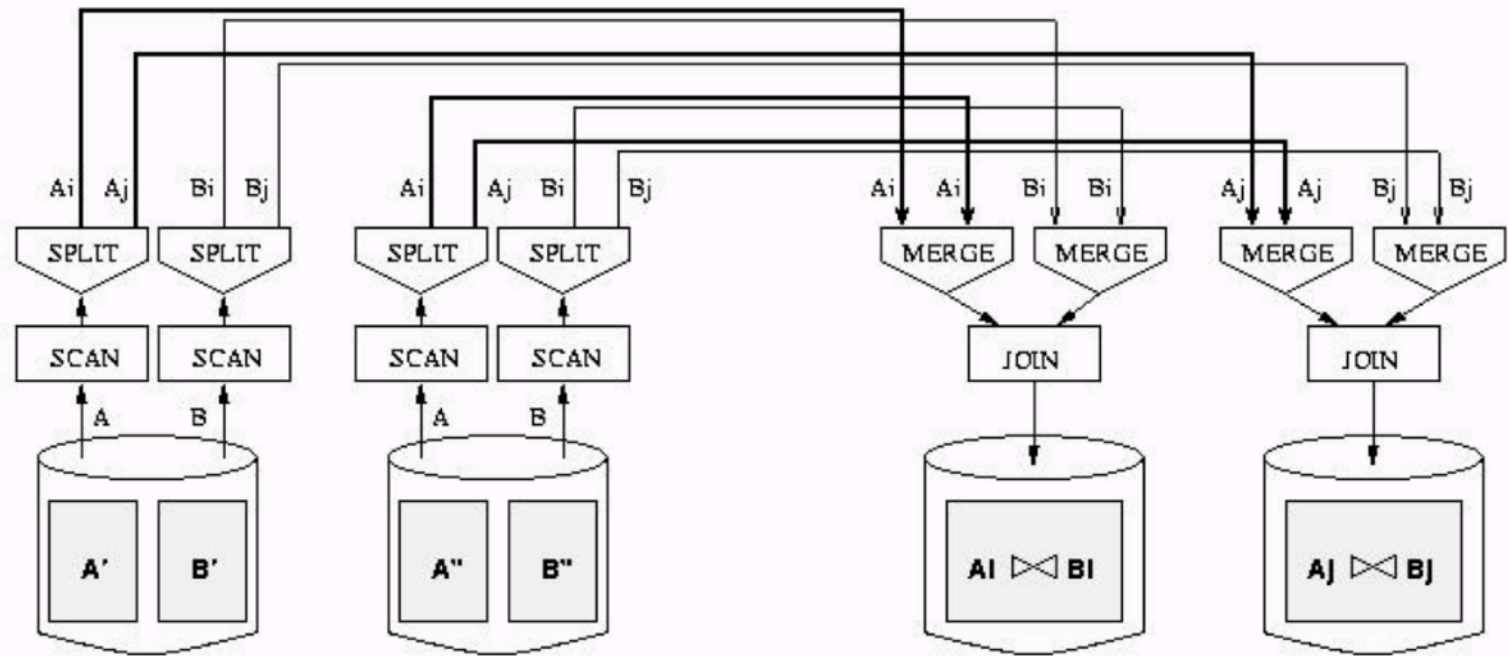
Parallel Nested Loops Join

- ❖ Given two tables A and B and JoinAttribute for equi-join
 1. Manager assigns range/hash splits on JoinAttribute to workers
 2. Repartitioning of A and B separately using same splits on JoinAttribute (unless pre-partitioned on it!)
 3. Worker i applies BNLJ locally on its partitions A_i and B_i
 4. Overall join output is just collection of all n worker outputs
- ❖ If join is not equi-join, there might be a lot of communication between workers; worst-case: all-to-all for cross-product!

Parallel “Split” and “Merge” for Joins

- ❖ Repartitioning quite common for parallel (equi-)joins
- ❖ Functionality abstracted as two new physical operators:
 - ❖ **Split:** each worker sends a subset of its partition to another worker based on Manager’s command (hash/range)
 - ❖ **Merge:** each worker unions subsets sent to it by others and constructs its assigned (re)partitioned subset
- ❖ Useful for parallel BNLJ, Sort-Merge Join, and Hash Join

Parallel Sort-Merge and Hash Join



- ❖ For SMJ, split is on ranges of (ordinal) JoinAttribute; for HJ, split is on hash function over JoinAttribute
- ❖ Worker i does local join of A_i and B_i using SMJ or HJ

Improved Parallel Hash Join

- ❖ 2-phase parallel HJ to improve performance
- ❖ **Idea:** Previous version hash partitions JoinAttribute to n (same as # workers); instead, decouple the two and do a 2-stage process: partition phase and join phase
- ❖ **Partition Phase:** Say $|A| < |B|$; divide A and B into k (can be $> n$) partitions using $h1()$ s.t. each $F \times |A_i| < \text{Cluster RAM}$
- ❖ **Join Phase:** Repartition an A_i into n partitions using $h2()$; build hash table on new A_{ij} at worker j as tuples arrive; repartition B_i using $h2()$; local HJ of A_{ij} and B_{ij} on worker j in parallel for $j = 1$ to n ; repeat all these steps for each $i = 1$ to k
- ❖ Uses all n workers for join of each subset pair $A_i \bowtie B_i$

Parallel Query Optimization

- ❖ Far more complex than single-node QO!
- ❖ I/O cost, CPU cost, and communication cost for each phy. op.
- ❖ Space of PQPs explodes: each node can have its own different local sub-plan (e.g., filescan vs. indexed)
- ❖ Pipeline parallelism and partitioned parallelism can be interleaved in complex ways
- ❖ Join order enumeration affected: bushy trees can become better!
- ❖ ... (we will skip more details)

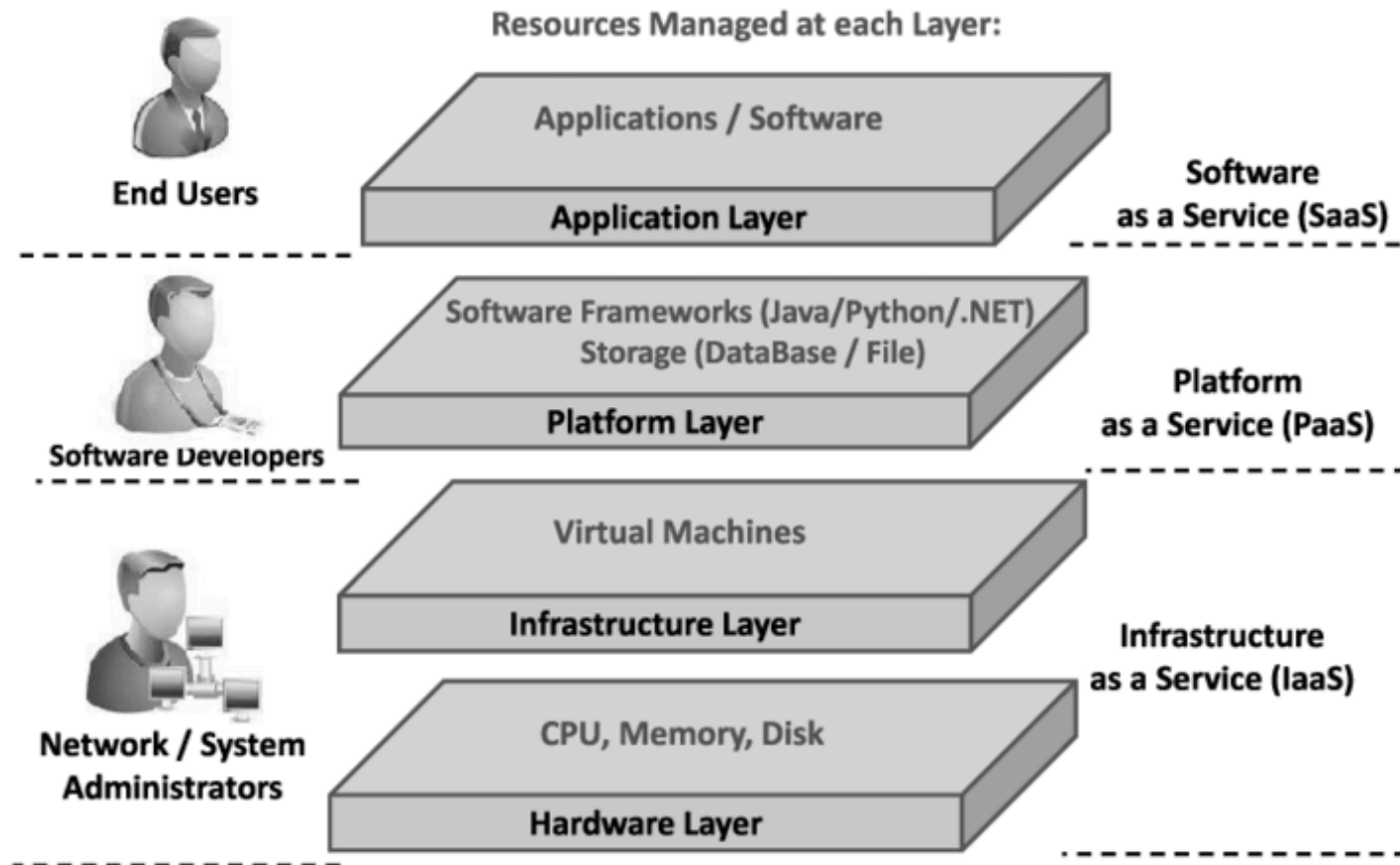
Outline

- ❖ Parallel RDBMSs
- ❖ Cloud-Native RDBMSs
- ❖ Beyond RDBMSs: A Brief History
- ❖ “Big Data” Systems aka Dataflow Systems

Cloud Computing

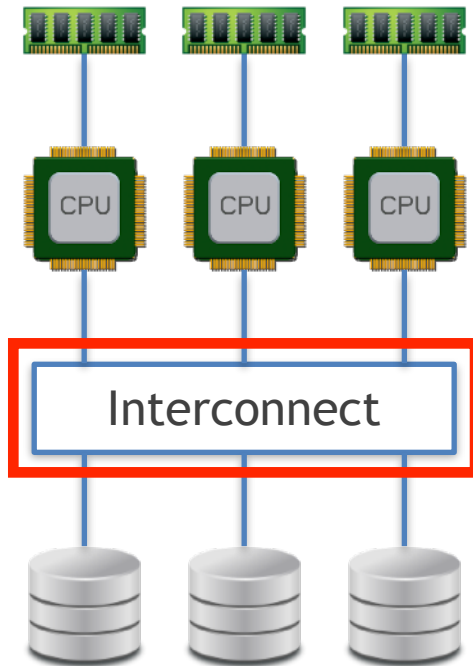
- ❖ Compute, storage, memory, networking are virtualized and exist on remote servers; rented by application users
 - ❖ **Manageability**: Managing hardware is not user's problem!
 - ❖ **Pay-as-you-go**: Fine-grained pricing economics based on actual usage (granularity: seconds to years!)
 - ❖ **Elasticity**: Can dynamically add or reduce capacity based on actual workload's demand
- ❖ Infrastructure-as-a-Service (IaaS); Platform-as-a-Service (PaaS); Software-as-a-Service (SaaS)

Cloud Computing



Q: How to redesign a parallel RDBMS to best exploit the cloud's capabilities?

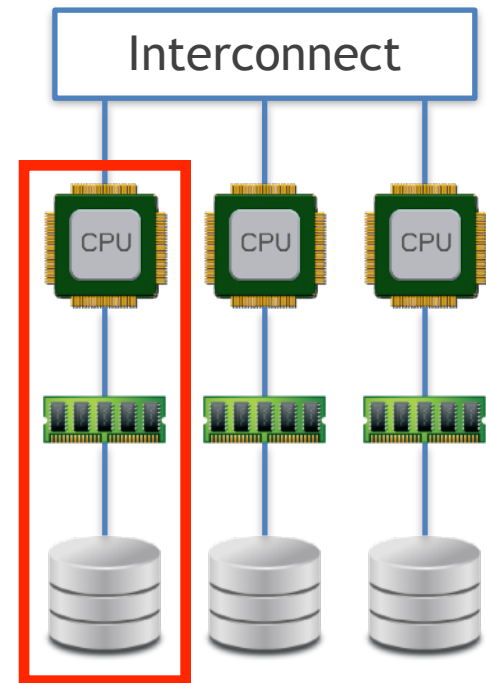
Revisiting Parallelism in the Cloud



Shared-Disk
Parallelism

Networks have
become much
faster: 100GbE to
even TbE!

Such bundling
could under-utilize
some resources



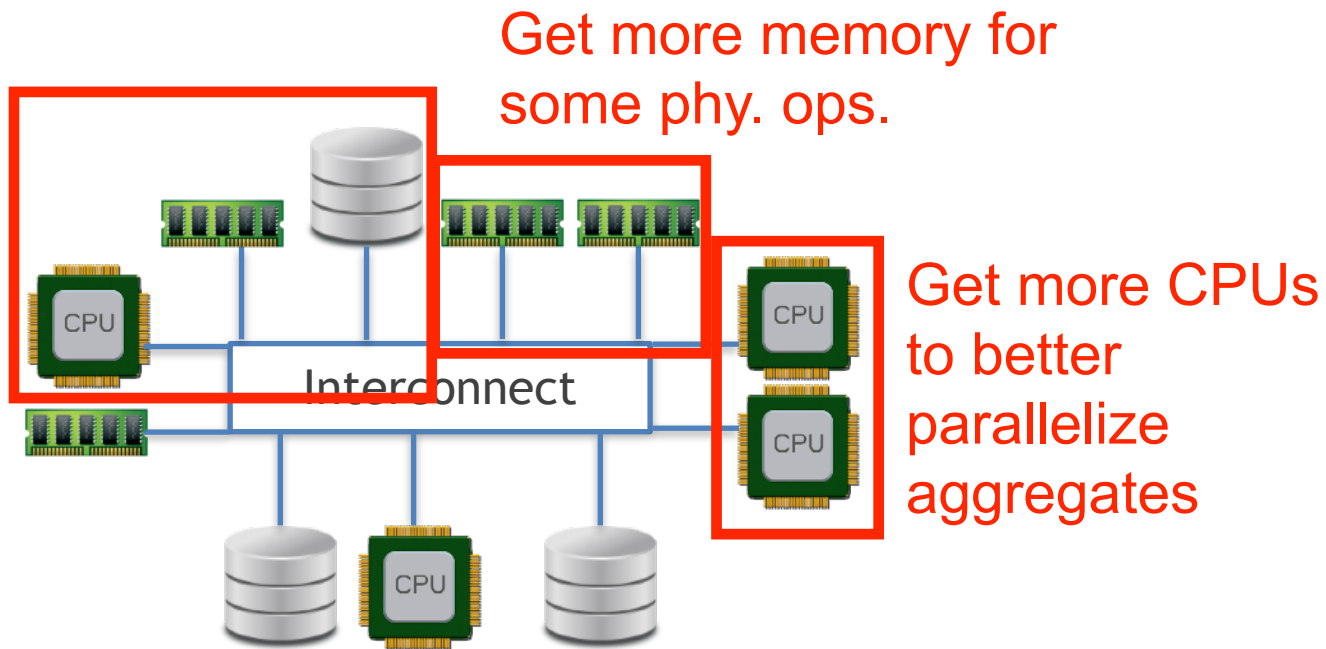
Shared-Nothing
Parallelism

Q: How to exploit cloud's virtualization of compute, memory, and storage resources to improve speed and utilization?

Revisiting Parallelism in the Cloud

The promise of full serverless / resource disaggregation:

All resources (compute, memory, storage) are network-attached and can be elastically added/removed



Q: How to fulfill the promise with minimal added latency?

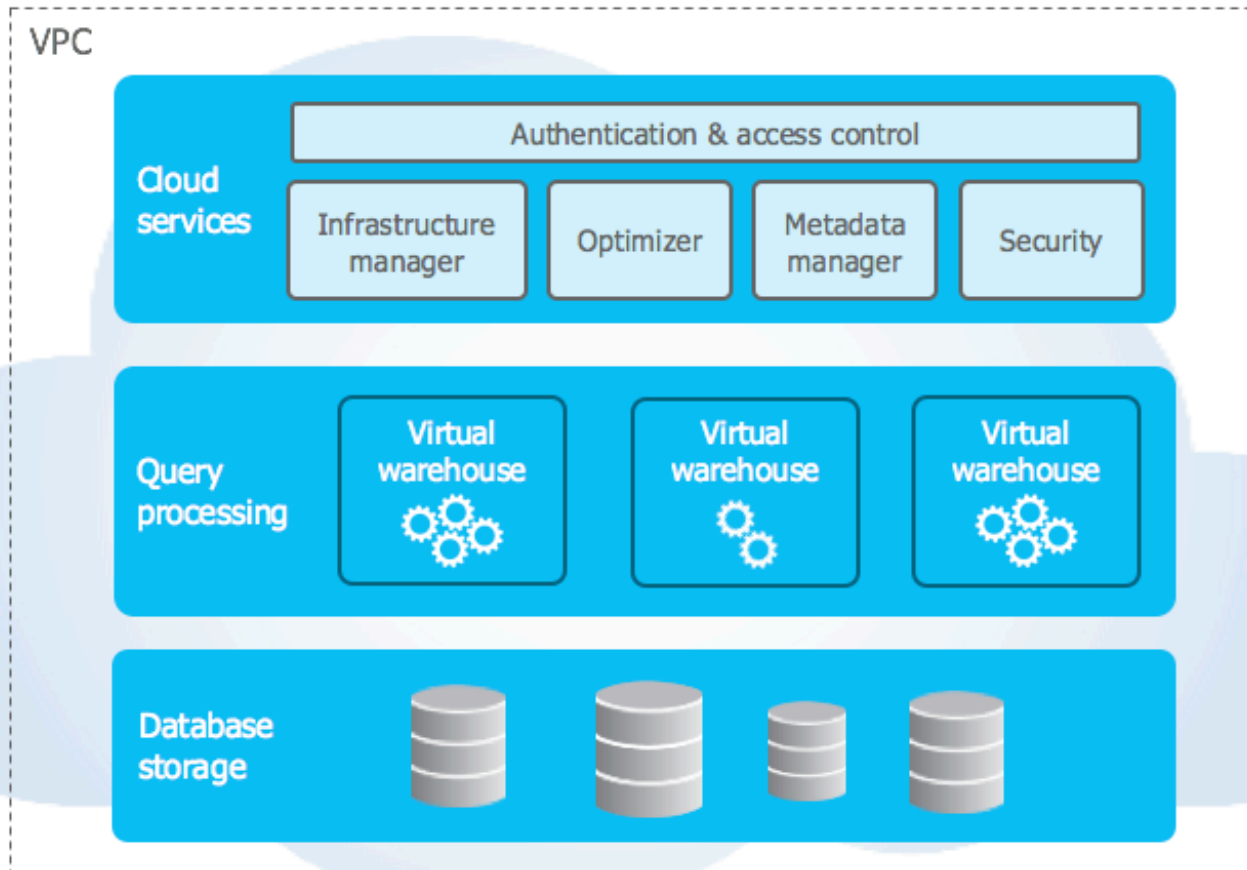
Cloud-Native Parallel RDBMSs

- ❖ Many cloud vendors, traditional RDBMS companies, startups



- ❖ Not just running a regular parallel RDBMS on IaaS!
 - ❖ Higher levels (data model, SQL, parser, etc.) preserved
 - ❖ Need to revisit, redesign, and reimplement all internals: storage subsystem, memory management, query processing and optimization, transaction management, and more

Key Example:



Shared-disk style
+ elastic compute

Each virtual
warehouse is an
independent MPP
compute cluster

Compressed
columnar format

Outline

- ❖ Parallel RDBMSs
- ❖ Cloud-Native RDBMSs
- ❖ Beyond RDBMSs: A Brief History
- ❖ “Big Data” Systems aka Dataflow Systems

Beyond RDBMSs: A Brief History

❖ Relational model and RDBMSs are **too restrictive**:

1. “Flat” tables with few data/attribute types

Object-Relational DBMSs: UDT, UDFs, text, multimedia, etc.

2. Restricted language interface (SQL)

PL/SQL; recursive SQL; embedded SQL; QBE; visual interfaces

3. Need to know schema first

“Schema-later” **semi-structured**: XML+XQuery; JSON; YAML

4. Optimized for static dataset

Stream data model; “standing” queries; time windows

Ad But the DB community has addressed these issues already
Take DSC 104, CSE 132B, CSE 135 on these DB capabilities

*So, why did people still need to look
beyond 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!

Outline

- ❖ Parallel RDBMSs
- ❖ Cloud-Native RDBMSs
- ❖ Beyond RDBMSs: A Brief History
- ❖ “Big Data” Systems aka Dataflow Systems
 - ❖ MapReduce/Hadoop
 - ❖ Spark

“Big Data”

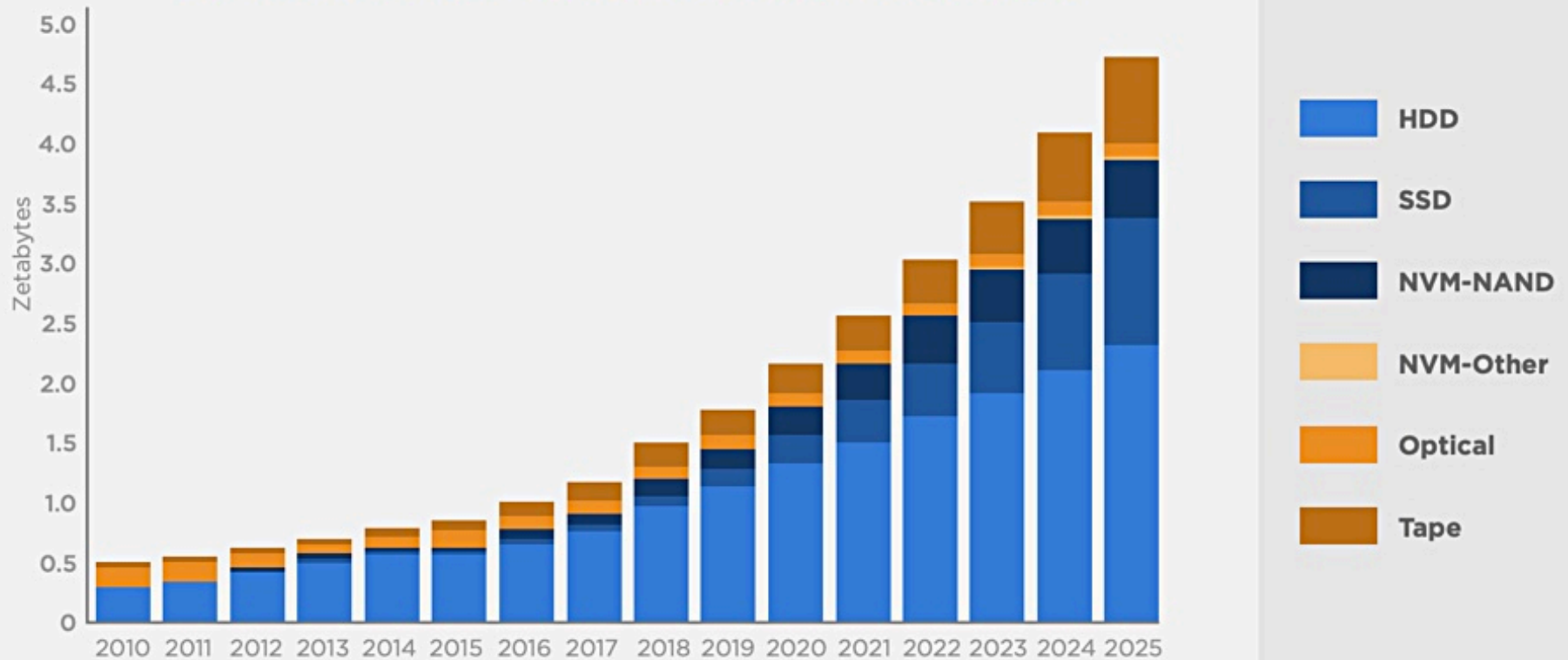
- ❖ Marketing term; think “Big” as in “Big Oil” or “Big Government”, not “big building” or “big car”
- ❖ Wikipedia says: “Data that is so large and complex that existing toolkits [read RDBMSs!] are not adequate [hah!]”
- ❖ Typical characterization by 3 Vs:
- ❖ **Volume**: larger-than-RAM; $\geq$ TBs, even Exabytes!
- ❖ **Variety**: relations, webpages, docs, tweets, multimedia, etc.
- ❖ **Velocity**: high generation rate, e.g., sensors, surveillance, etc.

Why “Big Data” now? 1. Applications

- ❖ New “data-driven mentality” in almost all applications:
- ❖ **Web**: search, e-commerce, e-mails, social media
- ❖ **Science**: satellite imagery, CERN’s LHC, document corpora
- ❖ **Medicine**: pharmacogenomics, precision medicine
- ❖ **Logistics**: sensors, GPS, “Internet of Things”
- ❖ **Finance**: high-throughput trading, monitoring
- ❖ **Humanities**: digitized books/literature, social media
- ❖ **Governance**: e-voting, targeted campaigns, NSA ☺
- ❖ ...

Why “Big Data” now? 2. Storage

Worldwide Byte Shipments by Storage Media Type



Source: Data Age 2025, sponsored by Seagate with data from IDC Global DataSphere, Nov 2018

Outline

- ❖ Parallel RDBMSs
- ❖ Cloud-Native RDBMSs
- ❖ Beyond RDBMSs: A Brief History
- ❖ “Big Data” Systems aka Dataflow Systems
 - ❖ MapReduce/Hadoop
 - ❖ Spark

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.)

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) :  
    Hmm, sounds suspiciously familiar! :)
```

```
    for each word w in doctext :
```

```
        emit (w, 1)
```

Part of MapReduce API

```
function reduce (String word, Iterator partialCounts) :
```

```
    sum = 0
```

```
    for each pc in partialCounts :
```

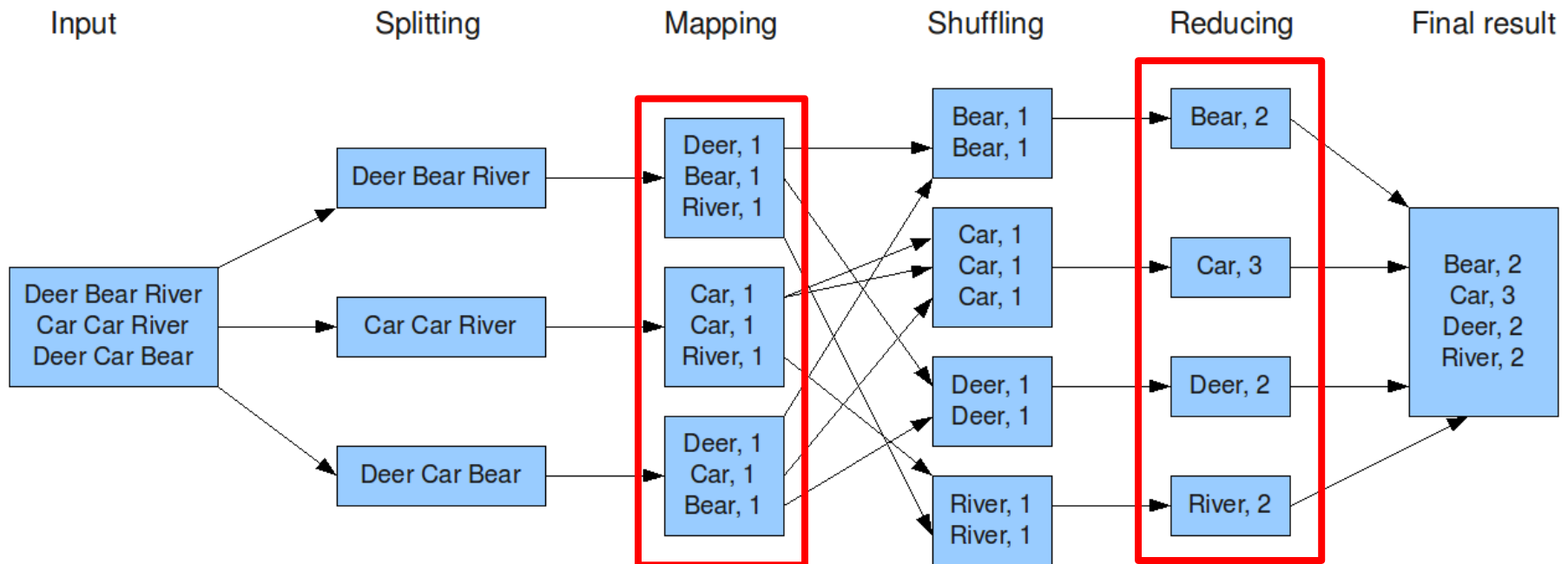
```
        sum += pc
```

```
    emit (word, sum)
```

How MapReduce Works

Parallel flow of control and data during MapReduce execution:

The overall MapReduce word count process



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.

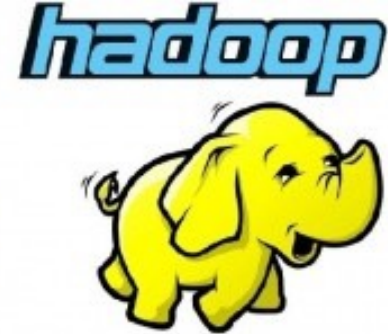
More MR Examples: GROUP BY Agg.

- ❖ 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 largely supplanted by Spark

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

A Spectacular “War of the Worlds”

[The Database Column](#)

A multi-author blog on database technology and innovation.

MapReduce: A major step backwards

By David DeWitt on January 17, 2008 4:20 PM | [Permalink](#) | [Comments \(44\)](#) | [TrackBacks \(1\)](#)

[Note: Although the system attributes this post to a single author, it was written by David J. DeWitt and Michael Stonebraker]

On January 8, a Database Column reader asked for our views on new distributed database research efforts, and we'll begin here with our recent trade press has been filled with news of the revolution of so-called "cloud computing." This paradigm entails harnessing large number of low-end servers to solve a problem. In effect, this suggests constructing a data center by lining up a large number of "jelly beans" rather than utilizing a much smaller number of high-end servers.

For example, IBM and Google have announced plans to make a 1,000 processor cluster available to a few select universities to teach students how to program such clusters using a software tool called MapReduce [1]. Berkeley has gone so far as to plan on teaching their freshman how to program using the MapReduce framework.

As both educators and researchers, we are amazed at the hype that the MapReduce proponents have spread about how it represents a paradigm shift in the development of scalable, data-intensive applications. MapReduce may be a good idea for writing certain types of general-purpose computations, but to the database community, it is:



No declarativity!

Filescan-based!

DeWitt's work on parallel DBMSs!

1. A giant step backward in the programming paradigm for large-scale data intensive applications
2. A sub-optimal implementation, in that it uses brute force instead of indexing
3. Not novel at all -- it represents a specific implementation of well known techniques developed nearly 25 years ago
4. Missing most of the features that are routinely included in current DBMS
5. Incompatible with all of the tools DBMS users have come to depend on

Cheap rip-off of RDBMSs!

“Young Turks” vs. “Old Guard”?

Swift and scathing rebuttal from MapReduce/Hadoop world!

1. A giant step backward in the programming paradigm for large-scale data intensive applications
2. A sub-optimal implementation, in that it uses brute force instead of indexing
3. Not novel at all -- it represents a specific implementation of well known techniques developed nearly 25 years ago
4. Missing most of the features that are routinely included in current DBMS
5. Incompatible with all of the tools DBMS users have come to depend on

DBMSs too high-level/hard to use for low-level text ETL

Meant for “offline” fault-tolerant workloads on cheap nodes

Google awarded a patent for MapReduce (ahem)!

MapReduce/Hadoop not meant to be an RDBMS replacement

Enter Hybrid Systems!

- ❖ **Clever DB researches:** *“Let’s get the best of both worlds!”*
- ❖ Numerous projects on hybrid systems in industry/academia:

Programming model-level: *Bring declarativity from RDBMS world to MapReduce/Hadoop world*



*SQL dialect
over Hadoop*



*Dataflow language
over Hadoop*

Systems-level: *Intermix system implementation ideas*

*HadoopDB from
Yale U.*

*Microsoft
Polybase*



Parallel Data Systems

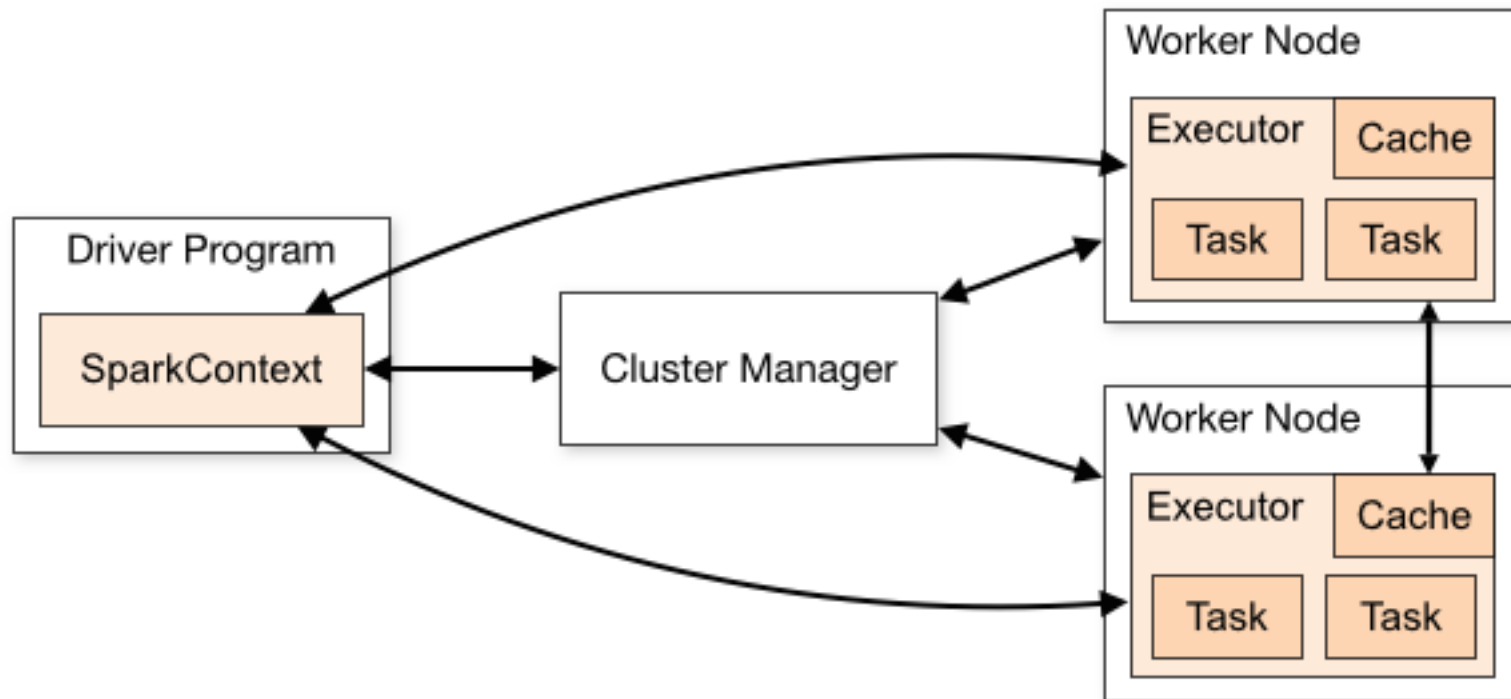
- ❖ Parallel RDBMSs
- ❖ Cloud-Native RDBMSs
- ❖ Beyond RDBMSs: A Brief History
- ❖ “Big Data” Systems aka Dataflow Systems
 - ❖ MapReduce/Hadoop
 - ❖ Spark

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

Distributed Architecture of Spark



Spark's Dataflow Programming Model

Transformations are relational ops, MR, etc. as functions

Actions are what force computation; aka *lazy evaluation*

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

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://...")
```

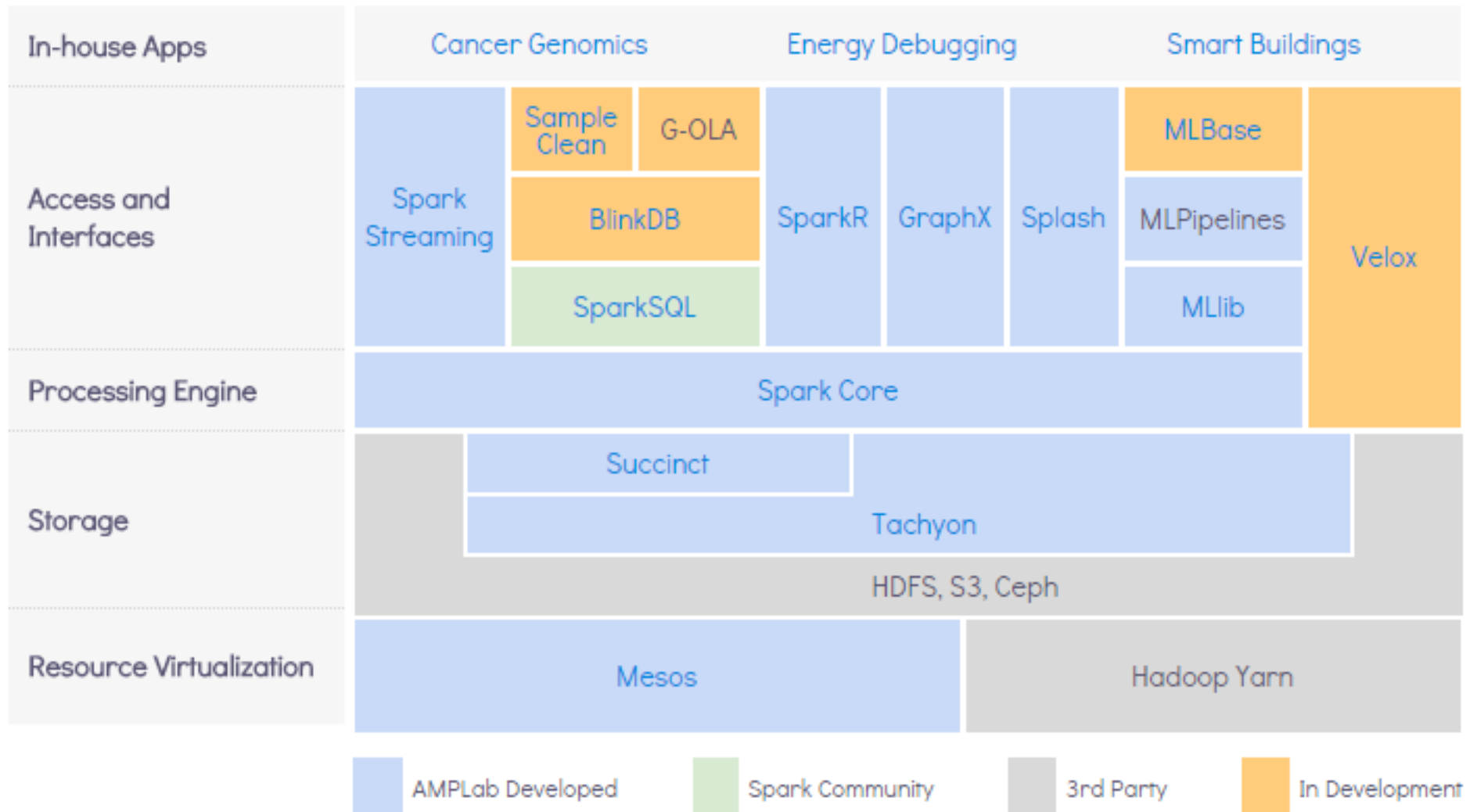
```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

```
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!

Ad: Take Yoav's CSE 255 to learn more Spark programming

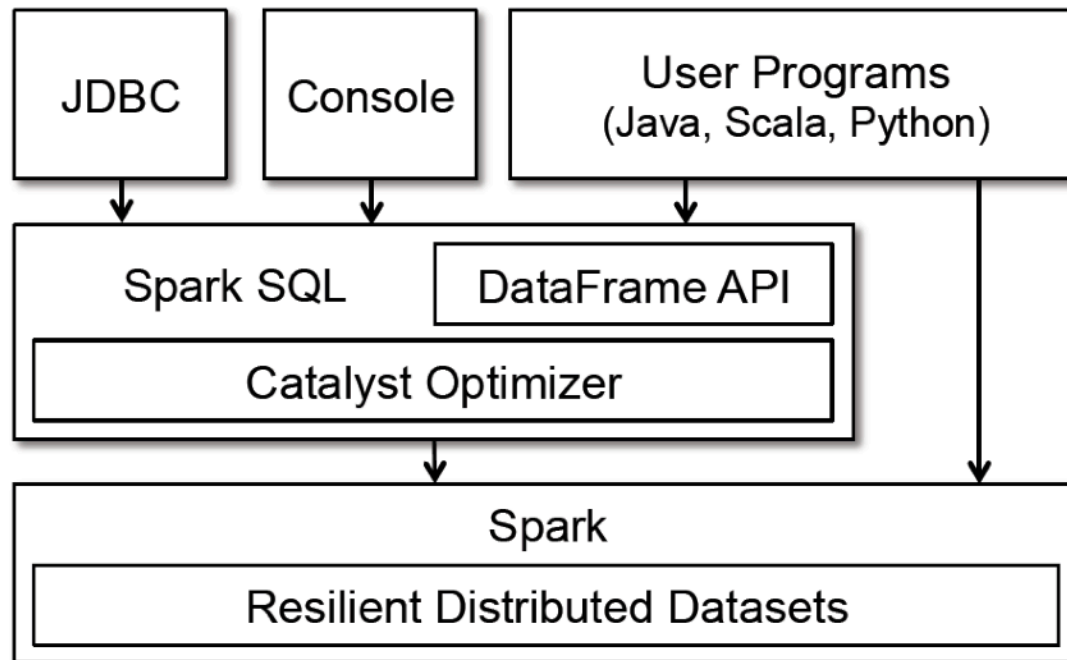
Spark-based Ecosystem of Tools



The Berkeley Data Analytics Stack (BDAS)

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! :)



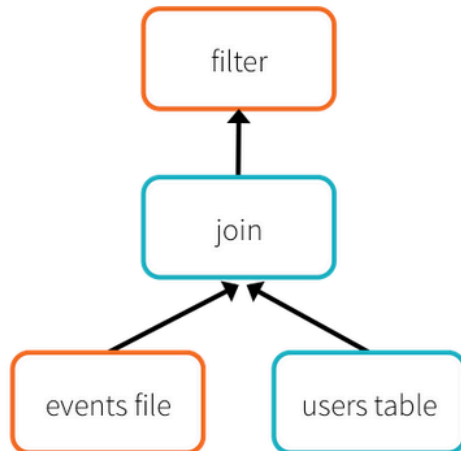
Query Optimization in Spark

- ❖ 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
- ❖ ...

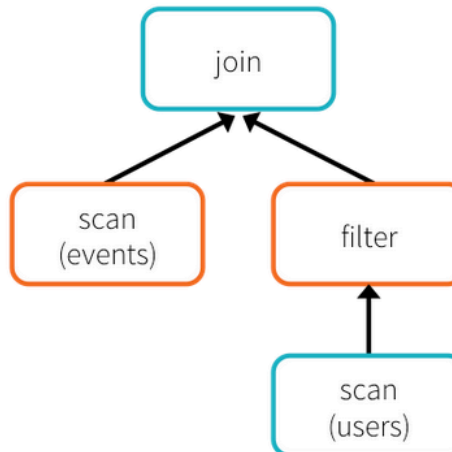
Query Optimization in Spark

```
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()
```

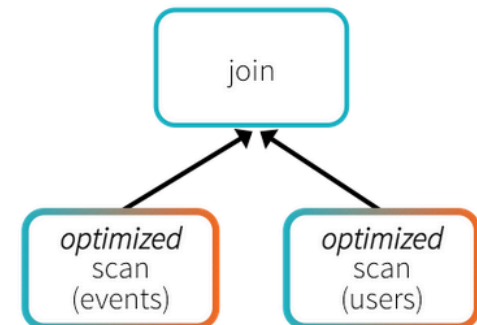
Logical Plan



Physical Plan



Physical Plan
with Predicate Pushdown
and Column Pruning



Databricks is building yet another parallel RDBMS! :)

Reinventing the Wheel?

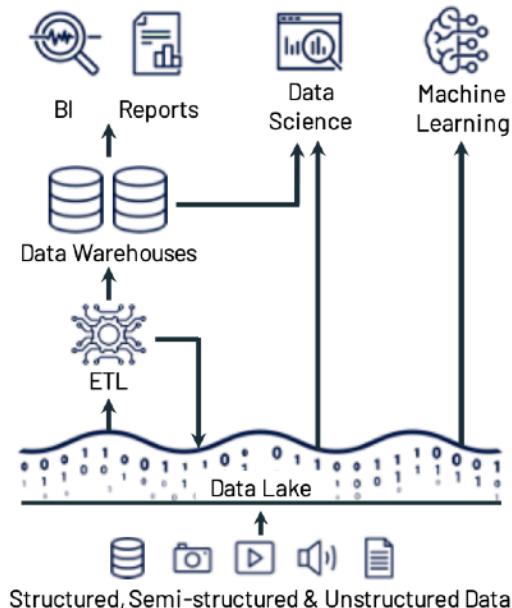


New Paradigm of Data “Lakehouse”

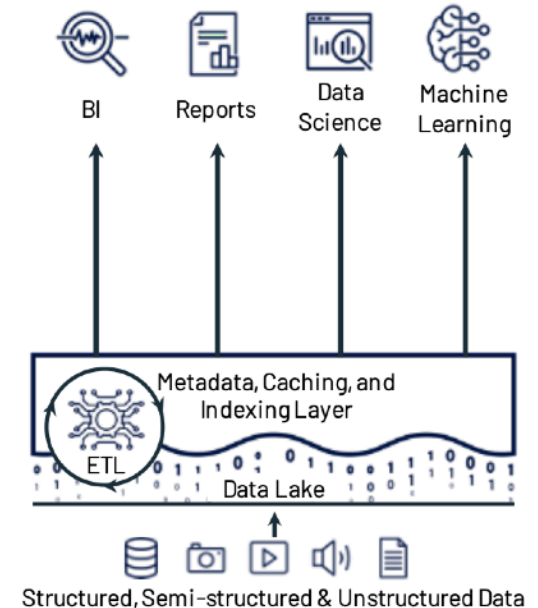
- ❖ **Data “Lake”**: *Loose coupling* of data file format and data/query processing stack (vs. RDBMS’s tight coupling); many frontends



(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

... which too is a form of DBMS! :)

 **Reynold Xin** @rxin
Replying to @TweetAtAKK

Time for
<rant>
A data
A Data
An ML
A model
A feature

While I agree with you, note that there's nothing wrong with "repeat and relearn". Many use cases' first order problems are not "data independence" (which is about change), but about being able to get the task done first. ;. 🐱

4:24 PM · Feb 12, 2022 · Twitter Web App

2 Likes

 Tweet your reply Reply

Those
doomed
</rant>

Arun Kumar @TweetAtAKK · Feb 12
Replying to @rxin
Reynold! I was wondering who'd be the first to comment b/w you and @matei_zaharia. 😊

Yes, agreed on your point. I put it more colorfully. 🙌 A wheel is a wheel but we don't use an Egyptian chariot wheels for a car nor a car's for a space shuttle. 😊

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>

Parallel Data Systems

- ❖ Parallel RDBMSs
- ❖ Cloud-Native RDBMSs
- ❖ Beyond RDBMSs: A Brief History
- ❖ “Big Data” Systems aka Dataflow Systems
 - ❖ MapReduce/Hadoop
 - ❖ Spark