Contents

1 Introduction to Distributed Systems **5**
- 1.1 Properties and Architectures . . . 5
- 1.2 Communication and Coordination . . . 6
- 1.3 Synchronous and Asynchronus Distributed System . . . 6
- 1.4 Distributed Computing . . . 6
- 1.5 Logical Clocks . . . 6
- 1.6 Vectorial Clocks . . . 7

2 Distributed data systems and shared nothing architectures **8**
- 2.1 Distributed data system . . . 8
- 2.2 Reference Architecture . . . 8
- 2.3 Share-nothing architectures and data distribution : why and how? 9

3 Partitioning **12**
- 3.1 How to partition . . . 13
- 3.2 What kind of requests . . . 13
- 3.3 Different type of partitioning . . . 13
- 3.4 Rebalancing . . . 14
- 3.5 Request Routing . . . 15
- 3.6 Intra-operator parallelism and partitioning : examples . . . 16

4 Replication **16**
- 4.1 Replication protocols . . . 17

5 Fault tolerance in distributed system **22**
- 5.1 Failure Classification . . . 22
- 5.2 Consistency . . . 24
  - 5.2.1 Consistency protocols . . . 24
- 5.3 Distributed Consensus . . . 25
- 5.4 Atomic Broadcast . . . 26
- 5.5 Chandra-Toueg's Consensus Algorithm . . . 26
- 5.6 Chandra-Toueg's Atomic Broadcast Algorithm . . . 27

6 CAP Theorem **27**
- 6.1 The CAP theorem illustrated . . . 28
- 6.2 CA Systems . . . 28
- 6.3 The CAP theorem revisited . . . 29
- 6.4 Consistency / Latency tradeoff . . . 29

7 Cluster in Cloud **29**
- 7.1 Definition of Cloud Computing . . . 29
- 7.2 Deployment Models . . . 30
- 7.3 Types of Cloud Services . . . 30
  - 7.3.1 Infrastructure as a Service (IaaS) . . . 30
  - 7.3.2 Platform as a Service (PaaS) . . . 31

7.3.3 Software as a Service (SaaS) . . . 31
7.3.4 Function as a Service(Faas) . . . 31
7.4 Cloud Economics : For Users and Providers . . . 31
7.5 Google Cloud Platform . . . 31
7.5.1 Storage . . . 31
7.5.2 Data Analysis . . . 32
7.5.3 Data Ingestion and Messaging . . . 32
7.6 Azure . . . 32
7.7 Containers in the Cloud . . . 32
7.7.1 Hub and spoke . . . 32
7.7.2 Kubernetes in the Cloud . . . 32

8 Hadoop **33**
8.1 Reference Scenario . . . 33
8.2 Hadoop . . . 33
8.2.1 The core of Hadoop . . . 33
8.2.2 Hadoop Ecosystem . . . 33
8.3 Introduction to HDFS . . . 34
8.3.1 Data Distribution . . . 34
8.3.2 System Architecture . . . 34
8.3.3 File Read . . . 35
8.3.4 File write(append) . . . 35
8.3.5 Recovery Management . . . 36
8.3.6 Behaviour with respect to the CAP theorem . . . 36

9 Map_Reduce **36**
9.1 An example : Word Count . . . 36
9.2 MapReduce Programming Paradigm . . . 38
9.3 Let's go back to the example : Word Count . . . 39
9.4 MapReduce Phases . . . 40
9.5 Back to the example : Word Count, alternative input representation 41
9.6 An example : Word Count - pseudocode 2 . . . 41
9.7 An example : Word Count - pseudocode 1 . . . 42
9.8 Four MapReduce variants for wordcount . . . 42
9.9 Different factors to balance . . . 42
9.10 MapReduce data structures . . . 42
9.11 MapReduce usage . . . 43

10 MapReduce - Relation Algebra Operators **43**
10.1 Selection . . . 43
10.1.1 Selection in MapReduce . . . 43
10.2 Projection . . . 44
10.2.1 Projection in MapReduce . . . 44
10.3 Union . . . 44
10.3.1 Union in MapReduce . . . 44
10.4 Intersection . . . 45

10.4.1 Intersection in MapReduce . . . 45
10.5 Difference . . . 45
10.5.1 Difference in MapReduce . . . 45
10.6 Join . . . 46
10.6.1 Join (reduce-side join) . . . 46
10.7 Group By . . . 47
10.7.1 Group By and aggregation in MapReduce . . . 47

11 Design Patterns **48**
11.1 Filtering Pattern . . . 48
11.2 Summarization pattern . . . 48
11.3 Join patterns . . . 49
11.4 Sorting pattern . . . 49
11.5 Two stage MapReduce . . . 50
11.5.1 Example of two stage MapReduce . . . 50
11.6 Exercise 1 : Top-k . . . 51
11.7 Exercise 2 : Inverted Index . . . 51

12 MapReduce Runtime Environment **52**
12.1 MapReduce Program . . . 52
12.2 MapReduce Input . . . 52
12.3 MapReduce Architecture . . . 52
12.4 Processing a MapReduce job . . . 52
12.5 Partitioning and task assignment . . . 54
12.6 Partitioners . . . 54
12.6.1 Issues with default partitioners . . . 55
12.6.2 Customization of partitioners . . . 55
12.6.3 Sorting(reprise) . . . 55
12.7 Combiners . . . 55
12.7.1 Combine function . . . 55
12.7.2 Word Count without combiners . . . 56
12.7.3 Word Count with combiners . . . 56
12.8 Questions and Answers . . . 56

13 Large Scale Data Processing Frameworks : Spark **57**
13.1 Introduction and Motivation . . . 57
13.1.1 MapReduce limitations . . . 57
13.1.2 Spark Computing Framework . . . 59
13.2 Spark Main Components . . . 59
13.3 RDDs . . . 60
13.4 Operators over RDDs . . . 61
13.4.1 Creation . . . 61
13.4.2 Transformations . . . 61
13.4.3 Actions . . . 61
13.5 Persistence . . . 62
13.6 Spark Programs . . . 62

13.6.1 Spark Context 62
13.6.2 RDD Transformations 62
13.6.3 Persitency and Caching of RDDs 62
13.7 Spark program : Word Count 63
13.8 Spark Transformations and Actions (Low-Level API) 64

Large Scale Computing

Riccardo Caprile

April 2021

1 Introduction to Distributed Systems

1.1 Properties and Architectures

A distributed system is a network that consists of autonomous computers that are connected using a distributed middleware. They help in sharing different resources and capabilities to provide users with a single and integrated coherent network. The main features are :

- Several autonomous nodes
- Each node has its own local memory
- Nodes communicate with each other by message passing

The properties are :

- Location Transparency : resource-centric
- Concurrent computations on different nodes
- No shared memory
- Absence of a global state
- Fault tolerance → The system must continue operating properly in case of failure of some of its components
- Heterogeneity in hardware and software

Let's see some architectures examples :

- Client-Server Architectures → Clients contact the server for data; results are commited back to the server when they represent a permanent change
- Three-tier → Architectures that move the client intelligence to a middle tier so that stateless clients can be used (Web applications)
- N-tier Architectures → They forward requests to other enterprise services
- Peer to Peer → There are no special machines that provide a service or manage the network resources. Responsibilities are divided among all machines known as Peers

1.2 Communication and Coordination

- Master/slave → Processes communicate directly with one another
- Database-Centric → Communication via shared database

1.3 Synchronous and Asynchronus Distributed System

The properties of the synchronous are : upper bound on message delivery , ordered message delivery, notion of globally synchronised clocks and Lock-step based execution. Whereas the asynchronous : Clock may not be accurate and can be delayed for arbitrary period of times.

What we want from these systems is : Interoperability , Integration , Flexibility, Modularity , Scalability , Quality of service and availability.

Some examples are : Network File System , Distributed Objects , Distributed database , Pub/Sub architectures.

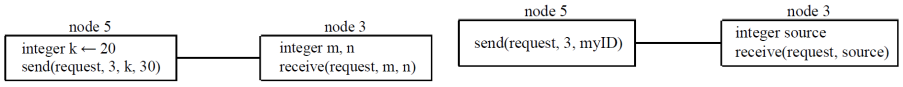
1.4 Distributed Computing

Distributed Algorithms

Distributed algorithms are often formulated as algorithms working on graphs. Network is represented by the graph , The nodes are the processes and the edge is the communication link via message passing. Classical problems for distributed systems are : Leader election , Mutual exclusion , Clock synchronisation , Global snapshot , Consensus in presence of failures, Routing/route maintenance

To illustrate some examples of message passing, let's consider the following type of message commands. send(TYPE,RECEIVER,D_1,....,D_n) receive(TYPE,D_1,...,D_n)

Sending and Receiving a message - Sending a message and Expecting a Reply

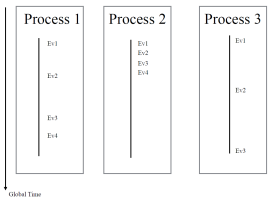


1.5 Logical Clocks

Algorithms based on logical clocks make use of timestamps to sort operations on different nodes of a network. We need to adjust individual clock in order to use them a global timestamps. (Scalar clocks and Vectorial Clocks)

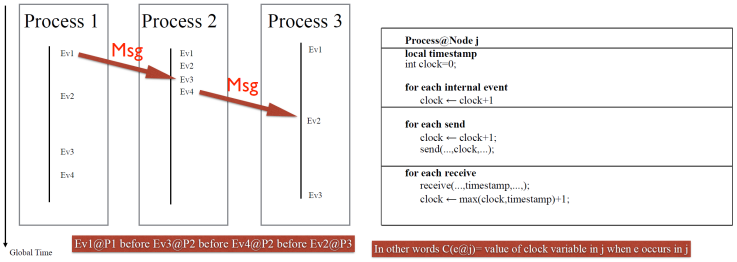
The relation a →b defines the event "a happened before b" in a set of distributed nodes.

- if a and b are events in the same node/process then a →b

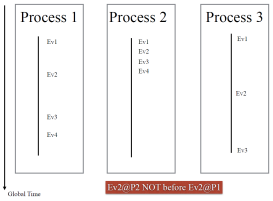


- if a is the event of sending message "m" and b is the event of its reception in a different node then a →b
- if a → b and b → c, then a → c

We would like to find a way to assign to each event "e" a timestamp C(c) such that if a → b then C(a) ¡ C(b)



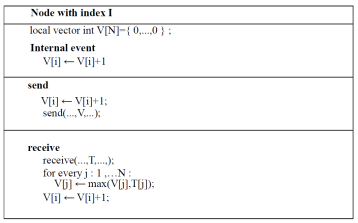
In general C(a) ¡ C(b) does not imply a → b, in other words scalar timestamp cannot be used to identify concurrent events.



1.6 Vectorial Clocks

Every node maintains a local representation of the logical clocks of all other nodes (for N nodes a vector of N elements). V[i] in node j = what node j knows about the logical clock of process i.

Ordering



$V \leq W$ iff $V[i] \leq W[i]$ for all $i : 1, \ldots, N$
$V < W$ iff $V \leq W$ there exists i s.t. $V[i] < W[i]$
$V = W$ iff $V \leq W$ and $W \leq V$.

Let C(e) be the vector clock when e occurs

Then the following property holds

$C(a) < C(b)$ if and only if $a \rightarrow b$

2 Distributed data systems and shared nothing architectures

2.1 Distributed data system

Any distributed system that provide commonly needed functionalities for storing, accessing , and processing data, by distributing data,load and control over the network. Distributed data systems are standard building blocks for developing data-intensive applications.

Compute-Intensive Computing applications which devote most of their execution time to computational requirements. CPU cycles are the bottleneck.

Data-Intensive Computing applications which require large volumes of data and devote most of their processing time to I/O and data manipulation. The quantity and the complexity of data as well as the speed at which they are changing are the bottleneck. The development of data-intensive applications rely on distributed data systems

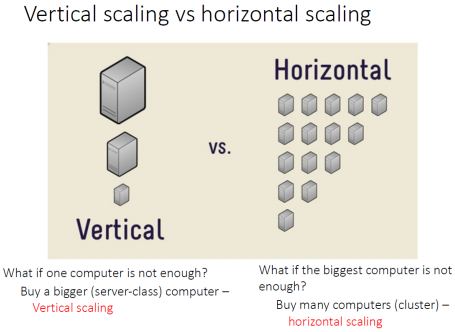
2.2 Reference Architecture

Big data applications require huge amounts of processing and data. Scaling is an issue

Key-ingredients for computing at scale

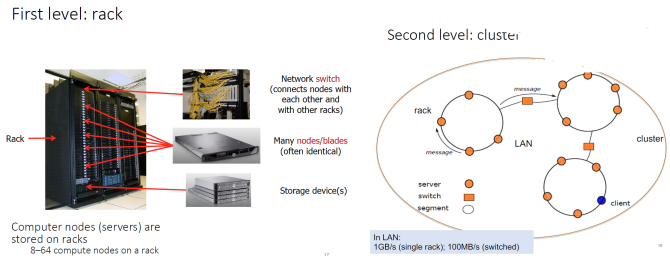
- Networking Infrastructure Clusters of computers that work together
- Distributed data for parallel processing
- Distributed processing

Computer clusters Set of loosely or tightly connected computers that work together so that, in many aspects, they can be viewed as a single system.



Rely on **Shared-nothing architecture** designed for data intensive computing. Close interconnection. Each node set to perform the same tasks. A cluster can be owned and used by a single organization or be available in the cloud.

Many levels. Rack scale out in Groups of racks - one cluster , then to groups of cluster - data center and the to groups of data centers.



Bandwidth : Amount of information transmitted over a channel in a given time unit.

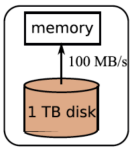
Third Level → Data center. If the cluster is too big to fit into an office building , you should build a separate building for the cluster and the result is DATA CENTER. It has hundreds of thousands of racks and it can be owned by a single organization and used by several organizations. Fourth Level → Network of data centers. If even a data center is not big enough , you need to build additional data centers.

2.3 Share-nothing architectures and data distribution : why and how?

What is the best architecture for developing data-intensive applications? Comparison with respect to performance for data access. Centralized architecture → read/write from disks Distributed architectures → read/write from disk + data exchange

Why data distribution???

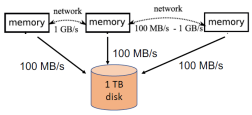
Centralized Architecture. No data distribution, sequential access. It takes 166 minutes to read 1 TB from disk



a. Single CPU, single disk

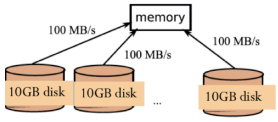
All of the next are analytical application. Sequentially read a large dataset from disks. Batch operations on the whole collection. Under this assumption the seek time is negligible regarding the transfer time

Shared disk Architecture No data distribution, distributed (parallel) processing. With 100 computers but a shared disk, this works as long as the task is CPU intensive, but becomes unsuited if large data exchanges are involved. The single disk becomes a bottleneck.



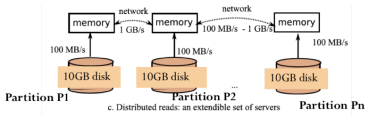
Shared memory and processor architecture

Data distribution , distributed (parallel) data access. With 100 disks , assuming the disks work in parallel and sequentially : about 1 min and 30s. **Data partitioning** : Splitting big database into smaller subsets called partitions so that different partitions can be assigned to different nodes. When the size of the data set increases, the CPU of the computer is typically overwhelmed at some point by the data flow and it is slowed down.



b. Parallel read: single CPU, many disks

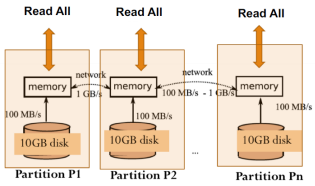
Shared nothing architecture Distributed data , distributed (parallel) data access and processing. With a cluster of 100 computers , each disposing of its own local disk : each processes its own Dataset. Data partitioning and bandwidth is a shared resource.



c. Distributed reads: an extendible set of servers

Side effect of data partitioning in Shared nothing architecture : task parallelization

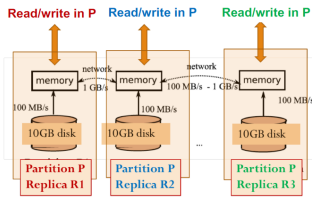
Data partitioning favoutrs intratask parallelization which means that the same batch operation executed in parallel over distinct partitions, by different nodes.



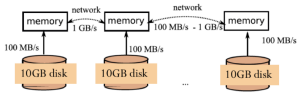
Now we have to consider an alternative scenario (Transactional application)

Workload consisting of lots of reads and writes operations, each one randomly accessing a small piece of data in a large collection. Distribute the load through **replication**

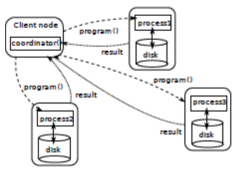
The Load distribution it comprehends concurrent random access operations executed in parallel over the same partition by different nodes.



The control distribution. P2P architecture (all nodes play the some role) and the control is distributed.



One master and many slaves , centralized control.



Principle 1

Disk transfer rate is a bottleneck for batch processing of large scale data sets. Typical of analytical processing. It is possible with shared nothing architectures.

Principle 2

Disk seek time is a bottleneck for transactional applications that submit a high rate of random accesses. Typical of transaction processing. It is possible with share nothing architectures.

Principle 3

Data locality. Bandwidth is a scarce resource, and program should be punished near the data they must access to. In this way, we rely on more fast communications.

Additional motivations for data distribution

Reliability denotes the ability of a distributed system to deliver its services even when one or several of its software or hardware components fail. **Faults** : system components , deviating from their specification program bugs , human errors, hardware or network problems. System that anticipate faults and can cope with them are called **Fault-Tolerant**. Fault tolerance is a based on the assumption that a participating machine affected by a failure can always be replaced by another one, and not prevent the completion of a requested task.

Availability : is the capacity of a system to limit as much as possible its latency. Involves several aspects : Failure detection and a quick recovery procedure.

Scalability : is the ability of a system to cope and continuously evolve in order to support increased load. Performance can decrease while increasing the number of nodes. It may also happen that some tasks are not distributed, either because of their inherent atomic nature. A node dedicated to some administrative tasks that is really negligible or that does not increase proportionally to the global workload is acceptable

Load Parameters : depend on the specific system (data volume, number of works). Some systems are elastic , they can automatically add computing resources when they detect a load increase.

3 Partitioning

Splitting a big database into smaller subsets called **partitions** so that different partitions can be assigned to different nodes.

3mm Assumptions

Shared nothing architecture. Scalable distributed data system : spread the data and the query load evenly across nodes. Data placement is a critical performance issue. Data as a set of n records R with attributes (K,A,B,C).

Load : set of frequent read/write accesses with respect to attribute K. K can be chosen as a **partition key**

3.1 How to partition

Each dataset is divided in p partitions where p depends on dataset size and access frequency. Favour **balanced partitioning**. Avoid **hot spots** : partitions with disproportionately high load.

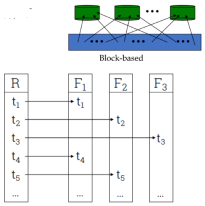
3.2 What kind of requests

- Batch query → read all data items and typical of analytical processing
- Point query → Selection of a single record and typical of transactional processing
- Multipoint/range query → selection of all the records satisfying a given condition, typical of transactional processing

3.3 Different type of partitioning

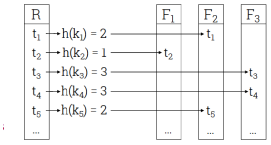
Block-based partitioning 3mm

Arbitrarily partition the data such that the same amount of data n/p is placed at each node. Use a Round-Robin approach or place the first n/p into the first node, the second n/p into the second node and so. No hot spots. Distributes data evenly, good for scanning full relation and not good for point or range queries.



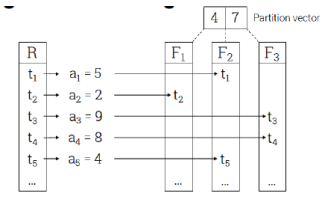
Hash-based partitioning 3mm

Applies a hash function to some attribute that yields the partition number in (1,...,p). Distributes data evenly if hash function is good. Good for point queries on key and joins. Not good for range queries and point queries.



Range-based partitioning 3mm

Partition the data according to a specific range of an attribute. Find separating points k1,...,kp. Send to the first node the tuples such that k1 ¡ t.K ¡ k2. Send to the n-th node the tuples such that kp ¡ t.K. Partition boundaries might be manually chosen by an administrator or automatically chosen by the system. It might generate hot spots



Each distributed data system supports a specific partitioning scheme. The partition key can be selected for any dataset. The selection of the partition key might favour some requests and make some others very inefficient. Depends on the reference workload.

Hash and range based partitioning help in determining the partition containing a given key and can reduce hot spots when the load is evenly distributed with respect to all the key values. When all r/w are for the same key , you still end up with all requests being routed to the same partition.

3.4 Rebalancing

The load may change. The number of request or the dataset size increases, so you want to add more nodes to handle the increased load. Rebalancing → process of moving load from one node in the cluster to another

Example

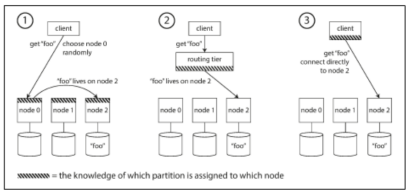
Suppose a hashed partitioning is applied : H(v) = v mod p, where p is the number of nodes. If the number of nodes changes, most of the data must be moved. Before p = 10, H(10) = 0 , H(11) = 1 , H(12) = 2. After p = 11 , H(10) = 10 , H(11) = 0 , H(12) = 1

Rebalancing is an expensive operation , because it requires moving a large amount of data from one node to another. **Fully automatic rebalancing** → the system decides automatically when to move partitions from one node to another, without any admin interaction. Less operational work to do for normal maintenance **Fully manual rebalancing** → The assignment of partitions to nodes is explicitly configured by an admin, and only changes when the admin explicitly reconfigures it. It's slower than a fully automatic process, but it can help preventing operational surprises.

3.5 Request Routing

When a client wants to make a request, how does it know which node to connect to???? As partitions are rebalanced , the assignment of partitions to nodes changes. Somebody needs to stay on top of those changes in order to answer the question. Three main approaches :

- **Allow clients to contact any node** → if that node coincidentally owns the partition to which the request applies, it can handle the request directly; otherwise, it forwards the request to the appropriate node, receives the reply, and passes the reply along the client
- **Send all requests from clients to a routing tier first** → which determines the node that should handle each request and forwards it accordingly. This routing tier does not itself handle any requests; it only acts as a partition-aware load balancer
- **Require that clients be aware of the partitioning and the assignment of partitions to nodes** → In this case, a client can connect directly to the appropriate node without any intermediary



How does the component making the routing decision (which may be one of the nodes , or the routing tier , or the client) learn about changes in the assignment of partitions to nodes???

1. Protocols for achieving consensus in a distributed system, but they are hard to implement correctly. 2. Rely on a separate coordination service (ZooKeeper) to keep track of this cluster metadata

Some systems rely on a **gossip protocol** among the nodes to disseminate any changes in a cluster state. Requests can be sent to any node, and that node forwards them to the appropriate node for the requested partition.

3.6 Intra-operator parallelism and partitioning : examples

Intra-operator parallelism → The read operator can be decomposed in many sub-operators, each of them executed on a given position, in an independent way The execution is more efficient if the processing is limited to nodes containing relevant data. Partitioning can make the execution of operator supporting intra-operator parallelism more efficient.

Full scan Retrieve all tuples from relation R(A,B,C). All nodes should be accessed no matter of the selected partition key and used partitioning scheme

Projection Retrieve column A from R(A,B,C). Assume we admit duplicates. pi_A(R).Parallelizable : each node can independently execute the operation for its own partition. Any partition key and scheme.

Selection Retrieve the tuples from R(A,B,C) that satisfy a given condition C. sigma_C(R). Data have to be filtered. Efficient parallelization depends on the selection condition and on the partitioning scheme.

Join

R(A,B,C) S(C,D,E) R natural join S,R —x— S. Many alternative algorithms for executing it in a parallel way

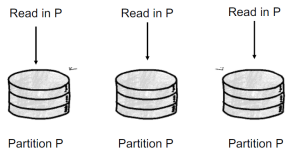
4 Replication

Replication means keeping a copy of the same data on multiple machines that are connected via network. Each node that stores a copy of the database is called **replica**. Most of the properties required by a distributed system depend on the replication data. Side effects in terms of performance and data consistency.

Pros : Latency

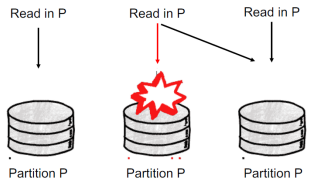
Keep data geographically close to your users (and thus reduce latency)

Pros : distribution of read operations (performance)



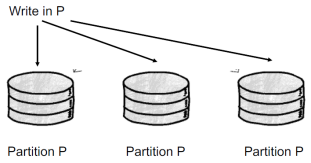
Scale out the number of machines that can serve r/w queries

Pros: Availability



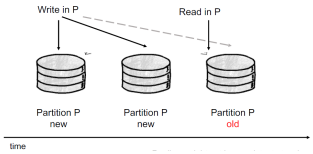
Allow the system to continue working even if some of its part have failed.

Cons: Performance of write operations



Writing several copies of an item takes more time, which may effect the throughput of the system.

Cons: Inconsistency



Replicas might not be consistent at a given instant of time.

4.1 Replication protocols

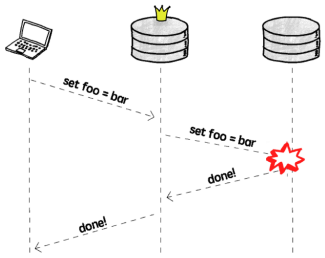
1.How writes are propagated to all the replicas.

Leader : nodes receiving first the write request. Follower : any other node storing a replica.

Synchronous replication

The leader waits until the follower has confirmed that it received the write before reporting success to the user, making the write visible to other clients.

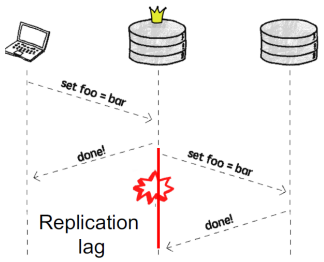
Pros : The follower is guaranteed to have an up-to-date copy of the data (STRONG CONSISTENCY). Write requests are sequentially executed by the leader : no concurrent writes but lower throughput and latency. Cons : if the synchronous follower does not respond, the write cannot be processed(limited



availability). Applications have to wait for the completion of other client's requests.

Asynchronous replication

The leader sends the message, but does not wait for a response from the followers.

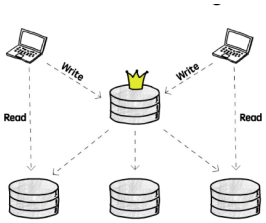


Pros : The leader can continue processing writes, even if all of its followers have fallen behind. At some point the replicas will become consistent . Cons : In a certain interval, some of the replicas may be out of date. Replication lag, delay between a write happening on the leader and being reflected on a follower. If the leader fails and is not recoverable, any writes that have not yet been replicated to followers are lost.

2.Number of nodes in which we write

Single leader replication

Just one leader, storing the primary copy of data

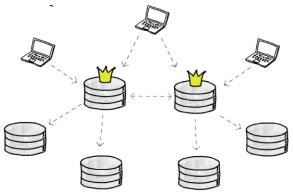


READ : clients can read from either the leader or any of the followers. Followers are read-only from the client's point of view

WRITE : Clients must send their write requests to the leader (it first writes the new data to its local storage and then it sends the data change to all of its followers). Each follower takes the log from the leader and updates its local copy of the database accordingly.

Pros : simple implementation , no concurrent writes. Cons : Limited throughput (all write operations are sequentially executed), limited availability, under an asynchronous protocol, read conflicts may arise

Multi-leader replication More than one node can accept writes (more than one leader). Replication still happens in the same way : each leader node that processes a write must forward that data change to all the other nodes. Typically asynchronous in order not to loose benefits of multiple leaders.



Pros : Increased write throughput , increased availability : in case of server failure , writes can be sent to other leaders.

Cons : under an asynchronous protocol read conflicts. The same data may be concurrently modified through different leaders , and those write confilicts must be resolved.

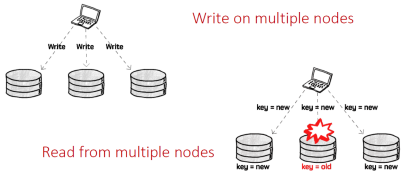
Usually one leader in each datacenter. Between datacenters , each leader replicates its changes to the leaders in other datacenter.

Leader-less replication

The client sends this write request concurrently to several replicas , and as soon as it gets a confirmation from some of them it consider that write a success and move on.

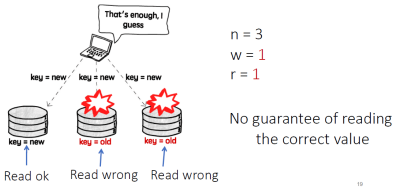
Pros : no leader , high throughput and availability

Cons : reads and write conflicts.

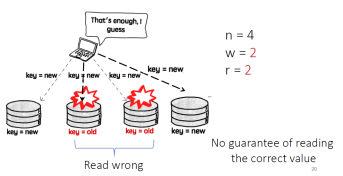


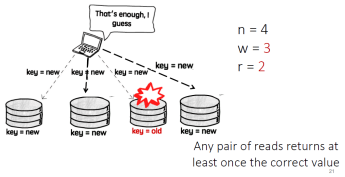
How to be sure that at least one value is up to date ???

n : number of replicas. **w** : number of successful - Write quorum read request sent to **r** nodes in parallel - read quorum



Read requests are sent to r nodes in parallel

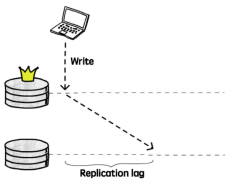




w + r ¿ n , at least one read value is up-to-date.

Let's demonstrate the power of quorum. w and r are usually configurable. w ¡ n : we can still process writes if a node is unavailable. r ¡ n : we can still process reads if a node is unavailable. n = 3 , w = 2 , r = 2 : we can tolerate one unavailable node. n = 5 , w = 3 , r = 3 : we can tolerate two unavailable nodes. w = n , r = 1 : fast read but just one failed node causes all database writes to fail.

3.Conflicts that may arise and how they are addressed.



If a client reads from a replica during the replication lag , it will receive outdated information , because the latest update were not applied yet. In normal operations , the replication lag may be only a fraction of a second , and not noticeable in practice. If the system is operating near capacity or if there is a problem in the network , the lag can easily increase to several seconds or even minutes , a real problem for applications. It is typical of asynchronous single-leader and multi-leader replication protocols.

During the replication lag , data are not consistent , they will become consistent later. Eventual consistency is not always a problem, it depends on the

replication lag and the reference application. (Post a picture on ig , your friend is able to see it after 30 seconds and it's not a big problem).

There are cases where the replication lag and related read conflicts are a real problem and generate specific types of conflicts. For each type of conflict , a related consistency level is defined if the system is able to avoid it.

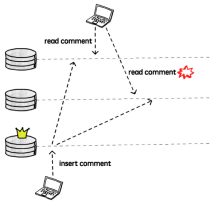
Read your write conflict

If the user views the data shortly after making a write , the new data may not yet have reached the replica.

A client never reads the database in a state it was before it performed a write. 1. When reading something that the user may have modified , read it from the leader; otherwise, read it from a follower. 2. Read from the leader for a certain time window after the update. 3. More sophisticated solutions based on timestamps.

Monotonic read conflict and consistency

After users have seen the data at one point in time , they should not later see the data from some earlier point in time.

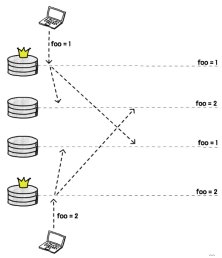


If you make several reads to a given value , all the successive reads will be at least as recent as the previous one. Time never moves backwards.

1.Each user always make reads from the same replica (different users can read from different replicas) 2.More sophisticated solutions based on timestamps.

Read conflicts might happen also under leader-less replication. Quorum appear to guarantee that a read returns at least once the latest written value. In practice this is not so simple because if a write happens concurrently with a read , the write may be reflected on only some of the replicas and in this case the read may return the old or the new value. Guarantees like read your writes and monotonic reads are not always achieved.

Write conflicts (concurrent writes) and consistency



Multi-leader and leaderless replication protocols allow several clients to concurrently write to the same item through two different leaders. Events may arrive in a different order at a different nodes, due to variable network delays and partial failures. Write conflicts may occur even if quorums are used.

Consistency avoid multiples values for the same replica , due to concurrent writes.

1. No concurrent writes 2. More sophisticated solutions based on timestamps. 3. In leader-less replication protocols , write some system - level custom conflict resolution code , when a w/r operation is executed , execute the code and choose one of the replicas as the correct one.

5 Fault tolerance in distributed system

A system is called **Fault Tolerant** if it will continue to perform (at some level) even when some component , process , link fails.

5.1 Failure Classification

- **Crash Failures**→ Occurs when a process halts forever. It is irreversible. In synchronous systems a crash failure can be detected using timeouts. In asynchronous system crash detection becomes much more difficult. Fail-stop failure is a simple abstraction that mimics crash failure when process behavior becomes arbitrary. Implementations of fail-stop behavior help detect which processor has failed. If a system cannot tolerate fail-stop failure, then it cannot tolerate crash.

- **Omission Failures**→ Message lost in transit. May happen due to various causes like : buffer overflow , receiver out of range ecc..

- **Transient Failures**→ Hardware : arbitrary perturbation of the global state. May be induced by weak batteries ecc. Software : Heisenbugs are a class of temporary internal and intermittent faults. They are essentially permanent faults whose conditions of activation occur rarely and are not easily reproducible, so they are harder to detect during the testing phase.

- **Software Failures**→ Coding error or human error, design flaws or inaccurate modeling, memory leaks.
- **Temporal Failures**→ Inability to meet deadlines. May be caused by poor algorithms or poor design strategy
- **Byzantine Failures** → Includes every conceivable form of erroneous behavior. Includes malicious behaviors

General properties of a system

A property should be true for every possible execution of the system. Two main classes : Safety (Nothing bad happens) and Liveness (Something good eventually happens)

Fault Tolerance

- Masking → neither safety nor liveness is violated
- Non-Masking → safety property may be temporarily violated but not liveness
- Fail-safe tolerance → a given safety predicate is preserved , but liveness may be affected.
- Graceful degradation → Application continues , but in a degraded mode. Much depends on what kind of degradation is acceptable

Failure detection

Depends on the system model , and the type of failures. If processors speed and channel delays have a known upper bound , failure can be detected using heartbeat messages and timeouts.

Omission detection

For FIFO channels we can use sequence numbers with messages. Message 5 was received but not message 4 so message 4 is missing. For Non-FIFO bounded delay channels we can use timeouts.

Detection of transient failures

The detection of an abrupt change of state from S to S requires the periodic computation of local or global snapshots of the distributed systems. The failure is locally detectable when a snapshot of the distance-1 neighbors reveals the violation of some invariant. In general we need global snapshots.

Recovery

- Backward Recovery → When a safety property is violated , the computation rolls back and resumes from a previous correct state

- Forward Recovery → Computation does not care about getting the history right, but moves on , as long as eventually the safety property is restored

Tolerating Omission Failures

Routers may drop messages , but reliable end-to-end transmission is an important requirement. If the sender does not receive an ack within a time period, it retransmits. This implies , the communication must tolerate Loss, Duplication and Re-ordering messages. There different types of ideal retransmission protocols using unbounded sequence numbers to reorder messages. If the communication channels are non-FIFO , and the message propagation delays are arbitrarily large, then , it is impossible to design a window protocol that can withstand loss , duplication , and message reordering by using bounded sequence numbers only.

5.2 Consistency

There are two main reasons for replication : Reliability and Performance. The reliability switch to another replica in the case of the current replica failure , provide multiple copies of data on different replicas. Performence divide the work.

Multiple copies are consistent if

1. A read operation returns the same value from all copies
2. A write operation as a single atomic operation updates all copies before any other operation takes place

Replication can cause consistency problem between multiple copies of data

5.2.1 Consistency protocols

Primary-based protocols

For each data item in a data store there is an associated primary that coordinates write operations on that item.

- Primary-backup protocol → Write operations are forwarded to a single server and read operations can be performed locally
- Local-write protocols → Primary copy moves between processes willing to perform an update. To update a data item , a process first moves it to its location. As a result, in this approach , successive write operations can be performed locally while each process can read their local copy of data items. After the primary finishes its update , the update is forwarded to other replicas and all perform the update locally

Replicated-write protocols All updates are carried out to all replicas.

- Active Replication → Updates are sent to each replica in the form of an operation in order to be executed. All updates need to performed in the same order in all replicas
- Quorum-based protocols → A client request and receives permission from multiple servers in order to read and write a replicated data

Ensuring consistency is a very difficult problem in distributed systems. How to select a primary leader? How to ensure that a sequence of operations is executed in the same order in all replicas? How to ensure that a sequence of operations is executed atomically in a replica? How to ensure that concurrent writes maintain our data store consistent?

The distributed consensus can be viewed as a generalization of all these problems

5.3 Distributed Consensus

Each process chooses a unique initial vale (in 0,1). Each process proposes a value to all other processes . Every node must agree on the same final value. Every node must agree on the same final value. The final value must be in the set of initial values.

Properties

- Termination → Every non-fault process must eventually decide
- Validity → If every non-faulty process begins with the same initial value v, then their final decision must be v
- Integrity → Every process decides only once
- Agreement → The final decision of every non-faulty process must be identical

If there is no failure , then reaching consensus is trivial. All-to-all broadcast followed by applying the same choice function. Consensus in presence of failure can however be complex. The complexity depends on the system model and the type of failures.

Impossibility result

In a purely Asynchronous distributed system , the consensus problem is impossible to solve with a deterministic algorithm if a even a single process crashes.

The impossibility result brings a stronger assumptions on the system , weaker properties and randomized algorithms.

Consensus algorithms are used in order to maintain consistent copies of logs in replicated nodes. For synchronous systems , there exist exact solutions even in presence of byzantine faults. In general replication need messages to be delivered in a consistent manner, otherwise replicas may diverge.

5.4 Atomic Broadcast

An atomic broadcast is a broadcast where all correct processes in a system of multiple receive the same set of messages in the same order; that is the same sequence of messages. The broadcast is termed atomic because it either eventually completes correctly at all participants , or all participants abort without side effects.

Properties

- Validity → If a correct participant broadcasts a message , then all correct participants will eventually receive it
- Uniform Agreement → If one correct participant receives a message, then all correct participants will eventually receive that message
- Uniform Integrity → A message is received by each participant at most once , and only if it was previously broadcast
- Uniform Total Order → The message are totally ordered , if any correct participant receives message 1 first and message 2 second , then every other correct participant must receive message 1 before 2

A value can be proposed by a process for consensus by atomically broadcasting it, and a process can decide a value by selecting the value of the first message which it atomically receives. Thus , consensus can be reduced to atomic broadcast.

Conversely , a group of participants can atomically broadcast messages by achieving consensus regarding the first message to be received , followed by achieving consensus on the next message , and so forth until all the messages have been received. Thus, atomic broadcast reduces to consensus.

5.5 Chandra-Toueg's Consensus Algorithm

The Chandra Toueg consensus algorithm solves in a network of unreliable processes equipped with an eventually strong failure detector and reliable channels. The failure detector is an abstract version of timeouts; it signals to each process when other processes may have crashed. An eventually strong failure detector is one that never identifies some specific correct process as having failed and that eventually identifies all faulty processes as failed. The Chandra consensus algorithm assumes that the number of faulty processes , denoted by f , is less than n/2

The algorithm goes through three asynchronous epochs , each of which spans several asynchronous rounds. In the first epoch several decision values are possible. In the second epoch a values gets locked : no other decision values are possible. In the third epoch processes decide the locked value. The algorithm uses a rotating coordinator : in each round r , the process whose identity is given by r mod n is chosen as the coordinator. Each process keeps track of its

current preferred decision value and the last round where it changed its decision value.

Initialization
Select an initial preference (value). Start executing the different phases until a value is decided. Any process that receives decide(preference) for the first time, decides preference and terminates

Phase 1
Let r = r + 1 be the current round and c = (r mod n) + 1 be the current coordinator. send (r,preference, timestamp) to current coordinator c

Phase 2
The coordinator waits to receive messages from at least half of the processes. It then chooses as its preference a value with the most recent timestamp among those sent, and then sends (r,preference) to all processes.

Phase 3
Each process waits to receive (r,preference) from the coordinator, or its failure detector to identify the coordinator as suspected to be faulty. In this first case , it sets its own preference to the coordinator's preference and responds with ack(r), in the second case, it sends nack(r) to the coordinator

Phase 4
Coordinator waits to receive ack(r)/nack(r) from a majority of processes. If it received ack(r) from a majority, it broadcasts decide(preference) to all processes.

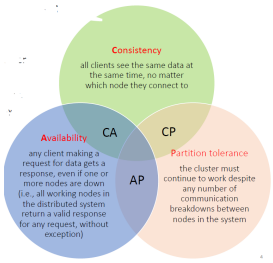
5.6 Chandra-Toueg's Atomic Broadcast Algorithm

The difficulty here is that faulty processes may crash at any moment , even when they are Coordinators and message delivery can be delayed. The failure detector is used to avoid to block in a wait statement. If the current coordinator is correct and not suspected to be faulty, then it will succeed. The strong failure detector ensures that there is a time after with correct processes will not be suspected to be faulty. Since decisions are taken via majority voting, if a coordinator decides value v at round r , then coordinators in round r' ¿ r will maintain the same value. Since decisions are taken using majority and there is a majority of correct processes , eventually the algorithm terminates.

6 CAP Theorem

Is it possible to build a distributed system that simultaneously satisfy all the expected properties? Always be available with high efficiency , provide strong consistency guarantees , provide high reliability with respect to network communication faults.

Network partition is a communication break within a distributed system which means a lost or temporarily delayed



The problem is that no distributed system can simultaneously provide consistency , availability and partition tolerance.

- CP System → A CP system delivers consistency and partition tolerance at the expense of availability. When a partition occurs between any two nodes , the system has shut down the non-consistent node until the partition is resolved
- AP System→ An AP system delivers availability and partition tolerance at the expense of consistency. When a partition occurs, the system remains available but some nodes might return an older version of data. When the partition is resolved , the AP database typically resync the nodes to repair all inconsistencies in the system
- CA System → A CA system delivers consistency and availability across all nodes. The system is not fault tolerant with respect to network partitions. When partition between any two nodes occurs , CA cannot be guaranteed at the same time

6.1 The CAP theorem illustrated

Ann is trying to book a room of the Ace Hotel in New York on a node located in London of a booking system. Pathin is trying to do the same on a node located in Mumbai There is only one room available. The network link breaks.

CA : Neither user can book any hotel room CP : Designate one node as the leader for Ace Hotel and rely on a synchronous protocol (Pathin can make the reservation and Ann cannot book the room until all the replicas are updated) AP : both nodes accept the hotel reservation (Overbooking!)

6.2 CA Systems

A single server system is the obvious example of CA system. CA cluster can also be built to ensure availability , if a partition occurs , all the nodes would go down so that no client can talk to any node. Availability in CAP : every request received by a non failing node in the system must result in a response. If all

nodes are failing , availability trivially satisfied. Need to ensure that partitions happen only rarely and completely , otherwise your system will be down most of the time. Not a reasonable assumptions for distributed data systems

6.3 The CAP theorem revisited

Misleading → network partitions are a kind of fault , so they aren't something about which you have a choice. It can be revised as follows. During normal periods both C and A can be achieved. When you have a partition in the network , you cannot have both C and A and a decision among C and A can be taken. When the partition is resolved the system takes corrective action coming back to work in normal situation. The decision between Consistency and Availability is not a binary decision. AP systems relax consistency in favour of availability but are not inconsistent. CP systems sacrifice availability for consistency but are not unavailable. This suggest both AP and CP systems can offer a degree of consistency , and availability

6.4 Consistency / Latency tradeoff

CAP does not force designers to give up A or C but there exists a lot of systems trading C. Why ? LATENCY. CAP does not explicitly talk about latency and performance in general. PACELC → if there is a partition (P), how does system trade off availability and consistency (A and C)? else (E) , when the system is running normally in the absence of partitions , how does the system trade off latency(L) and consistency (C)?

7 Cluster in Cloud

7.1 Definition of Cloud Computing

Cloud Computing is a general term used to describe a new class of network based computing that takes place over the Internet,

- A collection/group of integrated and networked hardware , software and Internet infrastructure
- using the Internet for communication and transport provides hardware , software and networking services to clients

These platforms hide the complexity and details of the underlying infrastructure from users and applications by providing very simple graphical interface or API. In addition , the platform provides on demand services , that are always on , anywhere , anytime and any place. The hardware and software services are available to general public , enterprises and corporations.

Cloud Computing is an umbrella term used to refer to Internet based development and services. A number of characteristics define cloud data , applications services and infrastructure :

- Remotely Hosted → Services or data are hosted on remote infrastructure
- Ubiquitous → Services or data are available from anywhere
- Commodified → The result is utility computing model similar to that of traditional utilities.

Cloud are transparent to users and applications , they can be built in multiple ways. In general, they are built on clusters of PC servers and off-the-shelf components plus open Source software.

7.2 Deployment Models

Public Cloud describes cloud computing where resources are dynamically provisioned on an on-demand , self-service basis over the Internet , via web applications , open API , from a third-party provider who bills on a utility computing basis.

A **Private Cloud** environment is often the first step for a corporation prior to adopting a public cloud initiative. Corporations have discovered the benefits of consolidating shared services on virtualized hardware deployed from a primary datacenter to serve local and remote users.

A **Hybrid Cloud** environment consists of some portion of computing resources on-site and off-site. By integrating public cloud services , users can leverage cloud solutions for specific functions that are too costly to maintain on-premise such as virtual server disaster recovery , backups and test/development environments.

A **Community Cloud** is formed when several organizations with similar requirements share common infrastructure. Costs are spread over fewer users than a public cloud but more than a single tenant.

Polynimbus is the term used to refer to the strategy of an organization utilizing multiple Cloud Providers. This enables organizations to utilize the best features and pricing of each cloud provider for different solutions where they fit the solutions , data , and workloaded best. It is an extremely common patter in use by all major corporations as they migrate to the Cloud to replace their on-premises datacenters.

7.3 Types of Cloud Services

7.3.1 Infrastructure as a Service (IaaS)

IaaS is the delivery of technology infrastructure as an on demand scalable service : processing , storage , networks , and other fundamental computing resources where the consumer is able to deploy and run arbitrary software , which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems.

7.3.2 Platform as a Service (PaaS)

Provides all of the facilities required to support the complete life cycle of building and delivering applications and services entirely from the Internet using programming languages and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure. He has control over the deployed applications and possibly application hosting environment configurations.

7.3.3 Software as a Service (SaaS)

It is a model of software deployment where an application is hosted as a service provided to customers across the Internet . The applications are accessible from various client devices through a thin client interface such as a web browser.

7.3.4 Function as a Service(Faas)

Serverless computing executes code that developers write using only the precise amount to compute resources needed to complete the task. When a pre-defined event occurs that triggers that , the serverless platform executes the task. The end user does not need to tell serverless provider how many times these events or functions will occur.

7.4 Cloud Economics : For Users and Providers

Pay as you go → Most services charge per minute Elasticity → Using 1000 servers for 1 hour costs the same as 1 server for 1000 hours Economies of Scale → Purchasing, powering and managing machines at scale gives lower per-unit costs than customers Speed of iteration → Software as a service means fast time-to-market, updates and detailed feedback

7.5 Google Cloud Platform

Compute Engines are customizable Virtual Machines interconnected by the inter/intra-datacenter Google Fiber. This is the most low level way to use GCP but is the fundamental building block of other services. Cloud Storage is a global , distributed , persistent filesystem, it is essentially a blob storage , capable of storing raw data in any format

7.5.1 Storage

GCP offers a series of data management solutions : Cloud SQL , Datastore , BigTable , BigQuery.

7.5.2 Data Analysis

Cloud DataLab allows you to analyse data using Google Cloud resources. Cloud Dataflow is an execution framework capable of parallel and auto-scaling processing. Cloud Dataproc is a cloud service that essentially provides Google-managed Hadoop clusters.

7.5.3 Data Ingestion and Messaging

Cloud IoT Core is a fully managed service on GCP to easily connect , manage and ingest data from devices. Cloud Pub/Sub is a publish/subscribe service provided by the GCP that provide a serverless global message queue. It is an asynchronous messaging service that decouples services that produce events from services that process events. It can be used as messaging-oriented middleware or event ingestion and delivery for analytics pipelines

7.6 Azure

Azure services are designed for large scale data ingestion , from either an IoT enabled device or from an application

7.7 Containers in the Cloud

Kubernetes is a platform and container orchestration tool for automating deployment, scaling , and operations of application containers.

In Kubernetes , there is a master node and multiple worker nodes , each worker node can handle multiple pods. Pods are just a bunch of containers clustered together as a working unit. You can start designing your applications using pods. Once your pods are ready , you can specify pod definitions to the master node , and how many you want to deploy.

7.7.1 Hub and spoke

Kubernetes has a hub-and-spoke API pattern. All API usage from nodes terminates at the apiserver. The apiserver is configured to listen for remote connections on a secure HTTPS port with one or more forms of client authentication enbled. The master controls the cluster. Kubelets manage worker node. The worker nodes run pods. A popd holds a set of containers. Pods are bin-packed as efficiently as configuration and hardware allows. Different pods can be clustered to form service with a stable external IP address.

7.7.2 Kubernetes in the Cloud

Kubernetes orchestrates containers across a fleet of machines , with support for : Automated deployment and replication of containers. Online scale-in and scale-out of containers clusters.....

8 Hadoop

8.1 Reference Scenario

- Batch Processing → large amount of input data : modest number of huge files. Runs a job to process it. Produces some output data
- (Few) long lasting jobs → often scheduled to run periodically
- Read-intensive scenarios → Files are written once, mostly appended to. High number of read requests of the whole file

We want to store and process large files , rely on a reliable storage system on cheap commodity hardware , achieve a high throughput when reading data and be able to parallelize the access to the content of a file And we don't have to : modify existing data , store a lot of small files and access to your data with a low latency.

8.2 Hadoop

Today Hadoop is used as a general-purpose storage and analysis platform for large scale data intensive computing.

8.2.1 The core of Hadoop

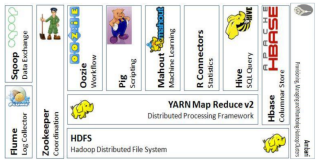
Storage Layer

HDFS → Hadoop Distributed File System. A distributed file system , based on master-slave architecture providing distributed data storage and Fault-Tolerant.

Data Processing Infrastructure Map Reduce : A programming model to facilitate the development and execution of distributed tasks. Provides a high level abstraction view . Fault Tolerant.

MapReduce abstracts away the distributed part of the problem (Programmers focus on what). The distributed part of the problem is handled by the framework (The Hadoop infrastructure focuses on how)

8.2.2 Hadoop Ecosystem



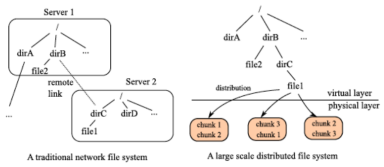
Pig → a data flow language and execution environmnet , based on MapReduce for exploring very large datasets Hive → a distributed data warehouse , based on MapReduce, for querying data stored in HDFS by means of a query language SQL-like. HBase → a distributed column-oriented database, which stores data relying on HDFS Sqoop → A tool for efficiently moving data between relational database ZooKeeper → A distributed coordination service

8.3 Introduction to HDFS

Simplest approach for large-scale distributed data storage.

The problem

Standard Network File System does not meet scalability requirements. Distributed File System storage is based on a virtual file namespace, and partitioning of files in "chunks"



8.3.1 Data Distribution

Partitioning : block-based partitioning , the architecture works best for very large files partitioned in large chunks. This limits the metadata information served by the Master. Based on record position in the files. Replication : Synchronous replication protocol. No concurrent writes. Usually , each partition is replicated 3 times. Nodes holding copies of one chunk are located on different racks.

8.3.2 System Architecture

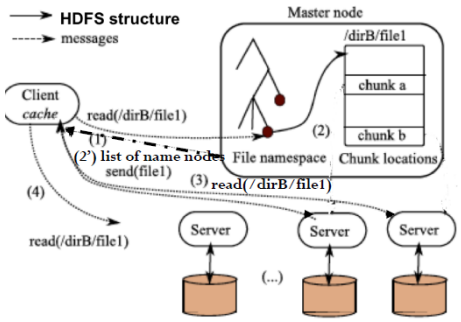
One **Master node (NameNode)**.

- Administrative tasks : replication and rebalancing ; garbage collection.
- Manages the file system namespace : holds file/directory structure , metadata , file-to-block mapping
- Regulate access to files by clients for reads and writes.
- Might be replicated for fault tolerance

Multiple **Slave servers(DataNodes)**. They store chunks/partitions , store and retrieve the blocks when they are told to. Block are themselves stored on standard single-machine file systems.

8.3.3 File Read

1. A client wishing to read a file must first contact the Master to determine where the actual data is stored
2. In response to the client request the Master returns : the relevant blocks ids and the location where the blocks are held.
3. The client send a read request to each data node holding a chunk of the file.
4. For improving scalability , the client keeps in its cache the addresses of the nodes accessed in the past , this knowledge can be used for subsequent accesses.

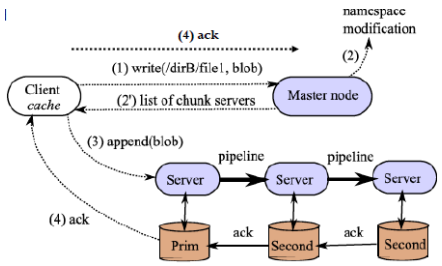


Data is never moved through the Master. All data transfer occurs directly between clients and DataNodes. Communications with the Master only involve transfer of metadata.

8.3.4 File write(append)

1. An application client wishing to append some records to a file must first contact the Master to determine the nodes where the partition to be updated is stored.
2. In response to the client request the Master: modifies namespace information , returns the locations where the chunks are stored, the file is locked until the write operation is concluded.
3. The client then contacts the datanode , among those storing the chunk , closest to it and sends the append request (append is synchronously executed
4. An ack is sent to the client only at the end of the process , the client ack the end of the write operation to the Master

Multiple clients cannot write into an HDFS file at the same time. No concurrent writes = no write conflicts. Synchronous updates = no read conflicts



8.3.5 Recovery Management

Logging at both Master and Server sites. The Master sends heartbeat message to servers , and initiates a replacement when a failure occurs. The Master is a potential single point of failure; its protection relies on distributed recovery techniques for all changes that affect the file namespace. Hadoop replicates it with an additional node , storing the same data and supporting the same task

8.3.6 Behaviour with respect to the CAP theorem

Due to Synchronous replication protocol , HDFS is consistent. Due to replication and additional recovery management activities , it is partition tolerant. It is not completely available : No answer to a write when a node storing one replica is down. No final answer to a read when all the replicas storing a partition are down. No answer to any request when the master node are down.

9 Map_Reduce

Let's see a typical Large-Data Problem :

1. Iterate over a large number of records in parallel
2. Extract something of interest from each iteration
3. Shuffle and sort intermediate results of different concurrent iterations
4. Aggregate intermediate results
5. General final output

Map will think about 1,2 whereas 4,5 reduce. The **key idea** is to provide a functional abstraction for these two operations

9.1 An example : Word Count

- Input → A large textual file of words
- Problem → Count the number of times each distinct word appears in the file

- Output → A list of pairs : < word , number of occurrences in the input file >

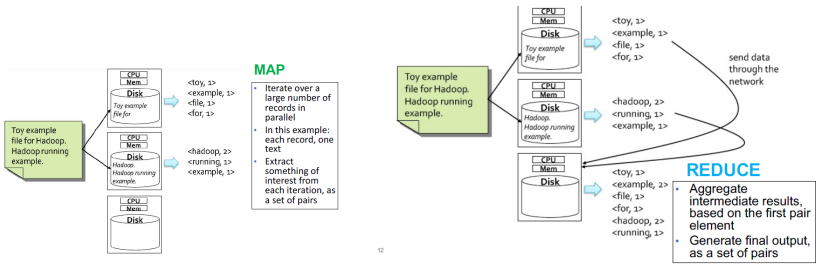
What if The entire file fits in main memory ? A traditional single node approach is probably the most efficient solution in this case. The complexity and overheads of a distributed system impact negatively on the performance when files of few GBs are analyzed. The following examples are proposed for illustrating the paradigm and do not represent use cases.

Case of interest : file too large to fit in main memory

Distributed file system useful in this case : file partitioning and replication, but how can we split problem in a set of (almost) **independent sub-tasks** and execute them in **parallel on a cluster of servers?**. A single analytical process to whole dataset and no need for logical design on the basis of the workload, you do not want to execute many distinct request against a single logical level

Let's start with the example

Suppose that we have : **The cluster with 3 Servers** and the content of the file in input is **(Toy example file for Hadoop. Hadoop running example)**



The problem can be easily parallelized. Each server process its chunk of data and counts the number of times each word appears in its chunk and can perform it independently (Synchronization is not needed in this phase). Each server sends its local(partial) list of pairs <word , number of occurrences in its chunk> to a server is in charge of aggregating local results and computing the global list/global result. The server in charge of computing the global result needs to receive all the local(partial) result to compute and emit the final list

A more realistic case

Let's suppose that the file size is 100 GB and the number of distinct words occuring in it is at most 1000. The cluster has 101 servers.

The file is optimally spread across 100 servers and each of these servers contains one (different) chunk of the input file (1 GB, corresponding to 1/100 of the original file), stored in a distributed file system.

Each server reads 1GB of data from its local hard drive. Each local list is composed of at most 1000 pairs (because the number of distinct words is 1000). The maximum amount of data sent on the network is 100 x size of local list (number of servers x local list size)

Weak scalability

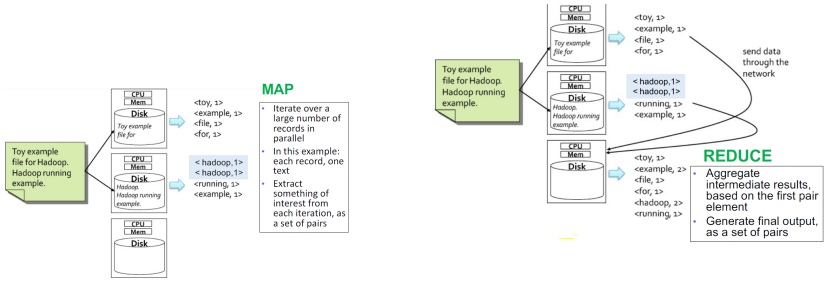
Given twice the amount of data, the word count algorithm takes approximately no more than twice as long to run. Because each server processes 2 x data -> 2 x execution time to compute local list. Given twice the amount of data and the number of servers , the execution time does not change.

Weak Scalability → Keep the load per node fixed and add more nodes, to cope with a load increase (the overall data increase)

Strong scalability Given twice the number of servers , the word count algorithm takes approximately no more than half as long to run. Because server processes 1/2 x data -> 1/2 x execution time to compute local list

Strong scalability → Keep the total load fixed and add more nodes, with the aim of increasing performance, for example the THROUGHPUT (the load per server will decrease)

The time needed to send local results to the node in charge of computing the final result and the computation of the final result are considered negligible in this running example but , frequently , this assumption is not true and depends on the complexity of the problem and on the ability of the developer to limit the amount of data sent on the network. **Variant 2 of Word count**



Both variant are correct and split the load map among map nodes , but variant 1 sends less data on the network and anticipate some work at map nodes

9.2 MapReduce Programming Paradigm

The MapReduce programming paradigm is based on the basic concepts of **functional programming**. The programmer defines the program logic as two functions :

- **Map** → Do something to everything in the a list

- **Reduce** → Combine/aggregate results of a list in some way

A complex program can be decomposed as a succession of Map and Reduce tasks

Map function

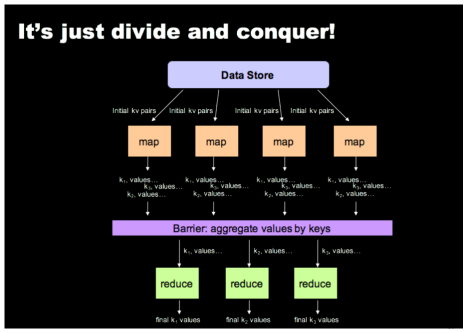
The input data set is a list of key-value pairs (k1,v1).

(k1,v1) -> [(k2,v2]). Applied over each pair of an input data set , for each input pair(k1,v1) emits a list of (key,value) pairs [(k2,v2)]. The application of the map function to each input can be parallelized in a straightforward manner, since each function application happens in isolation.

Reduce function

(k2,[v2] -> [(k3,v3)]

The reduce function applied over the list of pairs(k2,value) pairs (emitted by the map function) with the same key and emits a list of (k3,v3) pairs which is the **final result**. Elements in the input list must be "brought together" (leading to (k2,[v2] pairs) before the function Reduce can be applied, this is automatically performed by the system. The reduce aggregations can be proceed in parallel if different keys are sent to different reduce functions.



9.3 Let's go back to the example : Word Count

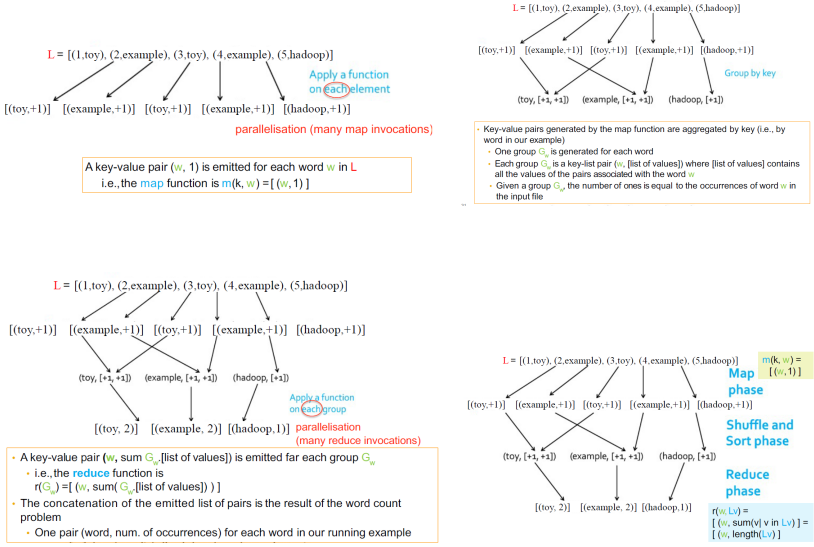
- Input → A large textual file of words
- Problem → Count the number of times each distinct word appears in the file
- Output → A list of pairs : < word , number of occurrences in the input file >

The input textual file is considered as a list of words L. There are two possible representations in terms of pairs :

1. **single pair (URL,L)**

2. **Multiple pairs (id,w)** where w is a term in L and id is a progressive number or the offset of the word in the input text, and is the one we are going to consider

L = [(1,toy),(2,example),(3,toy),(4,example),(5,hadoop)]



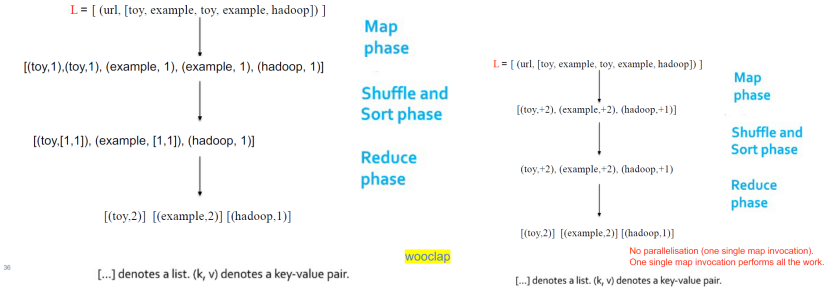
9.4 MapReduce Phases

The **Map** phase can be viewed as a transformation over each element of a data set. This transformation is a function m defined by the designer. Each application of m happens in isolation , so it can be parallelized.

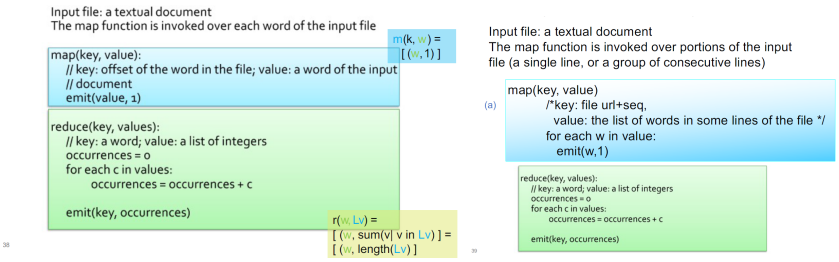
The **Reduce** phase can be viewed as an aggregate operation. The aggregate function is a function r defined by the designer. Also the reduce phase can be performed in parallel , since each group of key-value pairs with the same key can be processed in isolation

The **Shuffle and sort** phase is always the same. Group the output of the map phase by key. It does not need to defined by the designer.

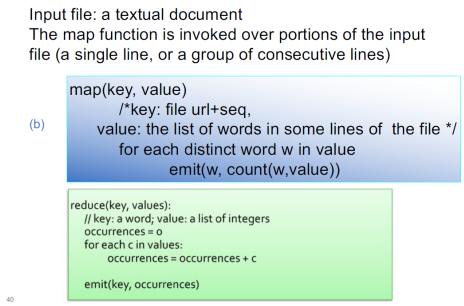
9.5 Back to the example : Word Count, alternative input representation



9.6 An example : Word Count - pseudocode 2



9.7 An example : Word Count - pseudocode 1

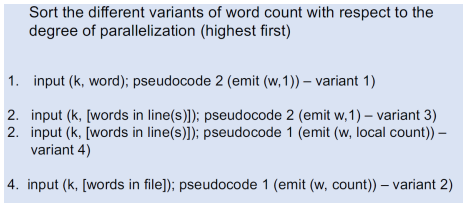


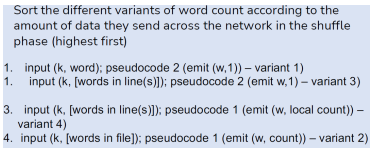
9.8 Four MapReduce variants for wordcount

1. input(k,word);pseudocode 2 (emit(w,1))
2. (k,[words in file]); pseudocode 1 (emit(w,count))
3. input(k,[words in line(s)]); pseudocode 2 (emit w , 1)
4. input(k,[words in line(s)]); pseudocode 1 (emit (w,local count))

9.9 Different factors to balance

All the four versions for wordcount work , but they differ in terms of : Parallelization , Split of work among mappers/reducers and the amount of data to be sent on the network.





9.10 MapReduce data structures

Key-value pair is the basic data structure in MapReduce. Keys and values can be : integers , float , strings..... They can also be arbitrary data structures defines by the designer. Both input and output of a MapReduce program are lists of key-value pairs. The design of MapReduce program involves imposing the key-value structure on the input and output data sets.

2mm In many applications , the keys of the input data set are ignored. The map function does not consider the key of its key-value pair argument. Some

specific applications exploit also the keys of the input data. Keys can be used to uniquely identify records/objects.

9.11 MapReduce usage

MapReduce is a **framework** not a tool. You have to fit your solution into the framework of map and reduce. It might be challenging in some situations. Need to take your algorithm and break it into filter/aggregate steps. Filter becomes part of the map function. Aggregate becomes part of the reduce function.

10 MapReduce - Relation Algebra Operators

The operators are **Selection , Projection , Union , Intersection , Difference , Join**

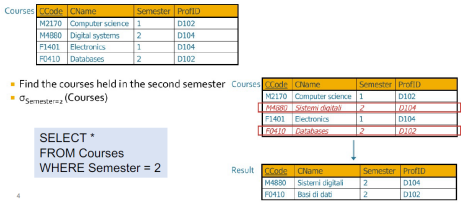
Relations/tables can be stored in the DFS.

Input pairs : for each **tuple** t of relation R , with **offset o** in the file generate **pair(o,t)**. Variant : for each tuple t of relation R , pair (t , t) of pair (kt t) where kt denotes the key value for t.

Output pairs : For each **t in the result , pair (t,t)**. The output is not exactly a relation , because it has key-value pairs. However , a relation can be obtained by using only the value components of the output.

10.1 Selection

$\sigma_C(R)$ Apply predicate (condition) C to each tuple (record) of table R. Return a relation containing only the tuples that satisfy C



10.1.1 Selection in MapReduce

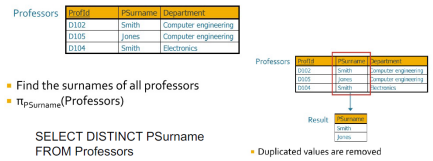
Map-only job

Map(o,t) Check if t satisfies C. if so , emit the key-value pair (t,t)
Reduce(key,value) returns(key,value).

10.2 Projection

$\pi_S(R)$

For each tuple in R , keep only the attributes in S. Return a relation with schema S. Remove duplicates , if any



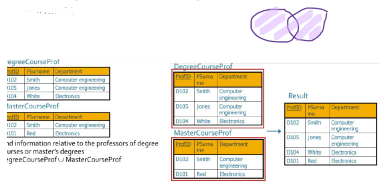
10.2.1 Projection in MapReduce

Map(o,t) constructs a tuple t' by eliminating from t those components whose attributes are not in S. Emit the key-value pair (t',t')

Reduce(key,value) For each key t' produced by any of the Map tasks, there will be one or more key-value pairs (t',t'). After sorting and shuffling , the Reduce function is called on each pair (t', [t',t',...,t']). The Reduce function then emits (t' , t'), thus aggregating [t',t',...,t'] into t' , so exactly one pair (t',t') is emitted for key t'.

10.3 Union

R U S R and S have the same schema. Return a new relation with the same schema of R and S. There is a tuple t in R U S for each tuple t in R or in S. Duplicated tuples are removed.



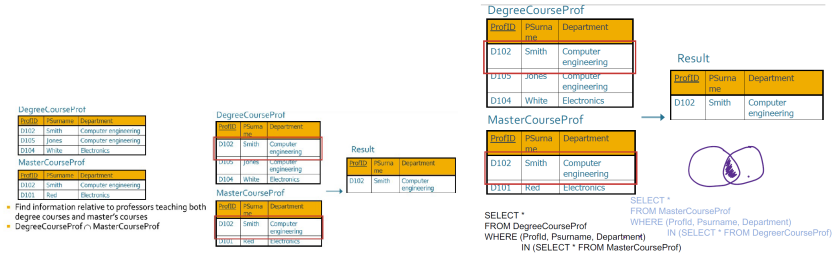
10.3.1 Union in MapReduce

Map(o,t) for tuples in R and in S Emit the pair(t,t)

Reduce(t,value) The list value may contain one or two. Values (in case one tuple belongs to both input relations) emit (t,t) in either case.

10.4 Intersection

R ∩ S R and S have the same schema. Return a new relation with the same schema of R and S. There is a tuple t in R ∩ if t appears in both R and S.



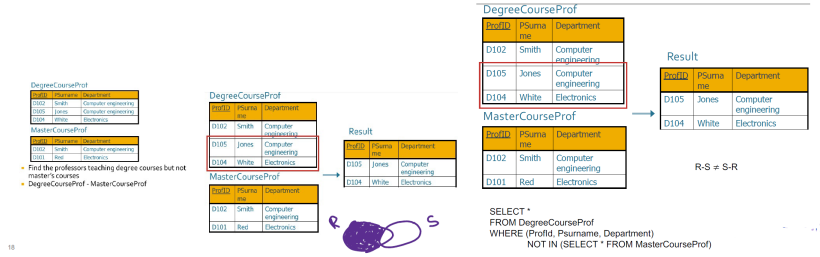
10.4.1 Intersection in MapReduce

Map(o,t) for tuples in R and in S Emit(t,t)

Reduce(t,value) If value = [t,t] emit (t,t) Otherwise , discard the pair (emit nothing).

10.5 Difference

R - S R and S have the same schema. Return a new relation with the same schema of R and S. There is a tuple in R-S if t appears in R but not in S.



10.5.1 Difference in MapReduce

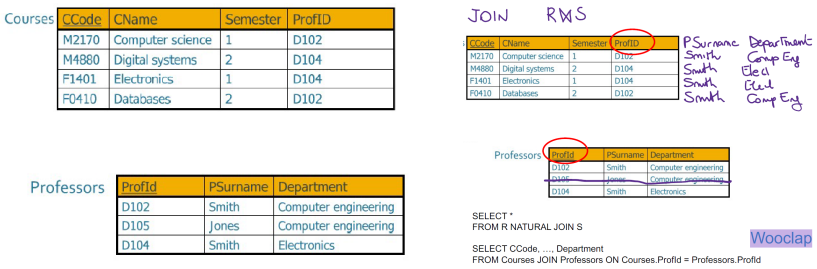
The Map function must inform the Reduce function whether the tuple came from R or S. We can use the relation as the value associated with a key t.

Map(o,t) for tuples in R and S emit(t,R) if it is a tuple of relation R emit(t,S) if it is a tuple of relation S

Reduce(t,value) if value = [R],then emit(t,t) Otherwise, emit nothing

10.6 Join

Find for each course the course information including the surname and the department of the professor teaching it. Courses |X| Professors



R(A,B) |X| S(B,C)

Many different implementations.

How to distinguish tuples coming from the first relation R from tuples coming from the second relation S.

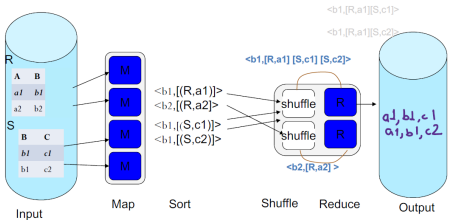
Reduce-side join implementation : use two map functions , one for each relation. each map function inserts the name of the input relation in the output value component. each map function insert the join attribute value as output key. then rely on a single reduce function for performing join , grouping is performed on join attribute values.

10.6.1 Join (reduce-side join)

R(A,B) |X| S(B,C)

Map(o,t) If t belongs to R , emit the pair (t.B , (R,t.A)) if t belongs to S , emit the pair (t.B , (S,t.C))

Reduce(b , value) Value is a list of pairs that are either of the form (R,a) or (S,c) Construct all pairs consisting of one with first component R and the other with first component S , say(R,a) and (S,c) For each such pair , emit (a,b,c)



10.7 Group By

Courses

CCode	CName	Semester	ProfID
M2170	Computer science	1	D102
M4880	Digital systems	2	D104
F1401	Electronics	1	D104
F0410	Databases	2	D102

$\gamma_{A,\theta(B)}(R)$

SELECT A, θ(B)
FROM R
GROUP BY A

Courses

CCode	CName	Semester	ProfID
M2170	Computer science	1	D102
M4880	Digital systems	2	D104
F1401	Electronics	1	D104
F0410	Databases	2	D102

Find the number of courses per semester

$\gamma_{A,\theta(B)}(R)$

SELECT A, θ(B)
FROM R
GROUP BY A

$\gamma_{Semester, count(Code)}$

Semester	COUNT
1	2
2	2

GROUP BY

Courses

CCode	CName	Semester	ProfID
M2170	Computer science	1	D102
M4880	Digital systems	2	D104
F1401	Electronics	1	D104
F0410	Databases	2	D102

$\gamma_{ProfId, count(1)}$

Prof_Id	
D104	2
D102	2

Find the number of courses per professor

$\gamma_{A,\theta(B)}(R)$

SELECT A, θ(B)
FROM R
GROUP BY A

StudentID	CCode	Grade	Date
1234	F1401	30	02-jun-2019
4567	F1401	28	02-jun-2019
8901	F1401	26	02-jun-2019
6666	F1401	24	02-jun-2019
1234	F0410	30	03-jun2019
8901	F0410	24	03-jun2019
1234	M4880	30	04-jun2019
6666	M4880	28	04-jun2019

Find the average grade per student:
SELECT StudentId, AVG(Grade)
FROM EXAMS
GROUP BY StudentId

$\gamma_{StudentId, Avg(Grade)}$ EXAMS

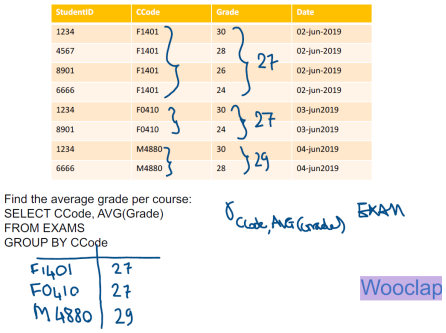
StudentID	Avg(Grade)
1234	30
4567	28
8901	25
6666	26

10.7.1 Group By and aggregation in MapReduce

R(A , B , C) Consider the operator $\gamma_A, \theta(B)$(R) corresponding to the following SQL query. SELECT θ(B) , FROM R , GROUP BY A. Map outputs tuples with the grouping attribute as key , Shuffle and sort does the grouping. Reduce does the aggregation

Map(o,t) if t = (a,b,c) emit the key-value pair(a,b)

Reduce(a,value) Each key a represents a group. let value = [b1,b2,..,bn] Apply the aggregation operator theta to [b1,b2,...,bn] let x the result.



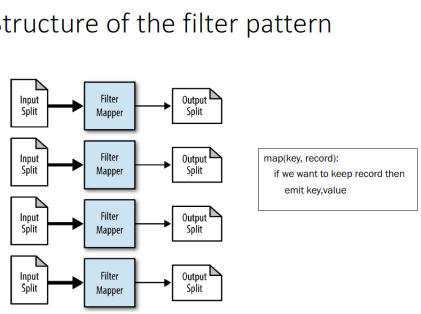
Emit (a,x)

11 Design Patterns

11.1 Filtering Pattern

Goal : find lines/files/tuples with a particular characteristics.

Map : filters and so it does most of the work Reduce : may simply be the identity



Extract some summarized information from detailed data items

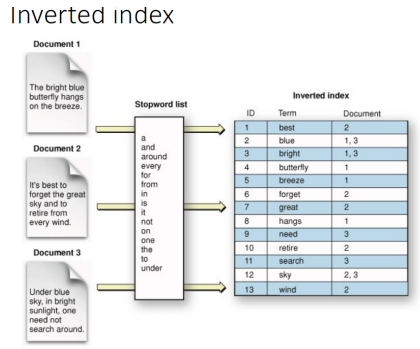
11.2 Summarization pattern

Goal : Compute the maximum , the sum , the average , ... , over a set of values Map : filters the input and emits (k,v) , where v is a summary value Reduce : takes (k,[v]) and applies the summarization to [v]

Find all documents with the words best and blue.

Map : Filters the two terms of interest (term = "best") OR (term = "blue) Emits(document , term)

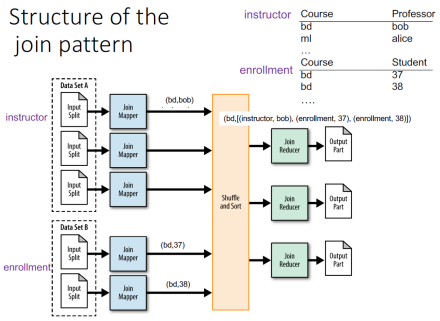
Reduce(document , list of terms) Performs the intersection (Remember that intersection is a special case of join : the natural join of two tables with identical



schemas is their intersection.

11.3 Join patterns

Goal : Combine multiple inputs according to some shared values. Map : generates a pair(k,e) for each element e in the input where k is the value to share. Reduce : generates a sequence of pairs [(k1,r1)],...,(kn,rn)] for each k[e1,...,ek] in the input

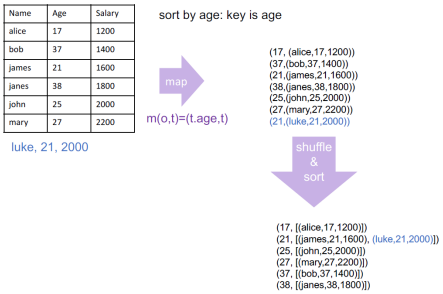


11.4 Sorting pattern

Goal : Sort input The programming model does not support this per se , but the implementations do (the Shuffle stage groups and orders The Map and the Reduce do nothing. If we have a single reducer , we will get sorted output.

Any problem with a single reducer? A single reduce node does all the work and it could get overwhelmed by lots of data

Do you see issues with multiple reducers ? No total order , partially ordered chunks at each node (each reducer receives only a part of the shuffle and sort results).

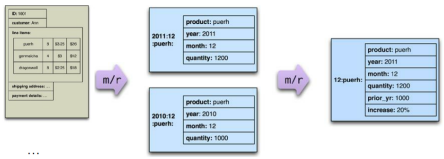


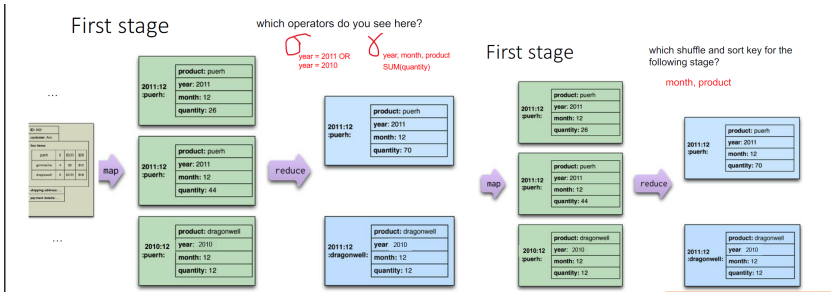
11.5 Two stage MapReduce

As MapReduce calculations get more complex , it's useful to break them down into stages , with the output of one stage serving as input to the next. Intermediate output may be useful for different outputs too , so you can get some reuse. The intermediate records can be saved in the data store , forming a materialized view. Early stages of MapReduce operations often represent the heaviest amount of data access , so building and save them once as a basis for many downstream uses saves a lot of work.

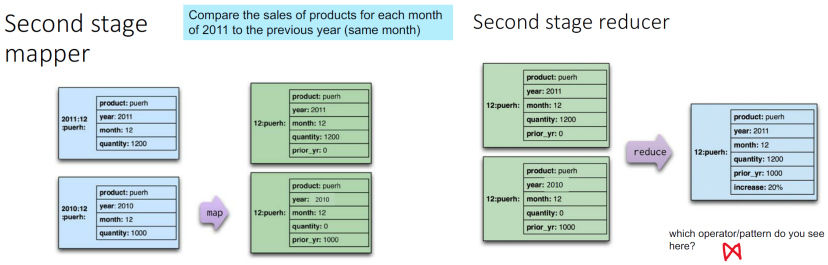
11.5.1 Example of two stage MapReduce

We want to compare the sales of products for each month in 2011 to the prior year. First stage : produces records showing the sales for a single product in a single month of the year. Second stage : produces the result for a single product by comparing one month's result with the same month in the prior year.





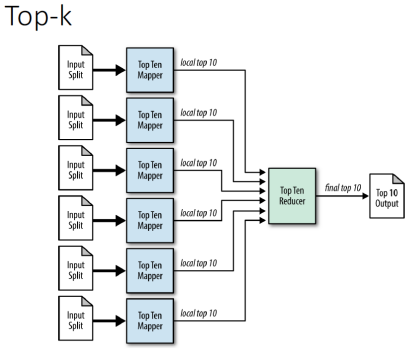
First stage : filter , summarization. The intermediate output is total sales per product per moth , for months of 2010 and 2011. It can be useful for other



analyses , materialized view. Second stage ; join on month and product.

11.6 Exercise 1 : Top-k

Given a set of records [Name : n , Age : g , Salary , s]. Find the top 10 employees younger than 30 with the highest salaries. Which shuffle and sort key would you use here? Null (or any other value , what's important is that it is the same for all top-10 lists)



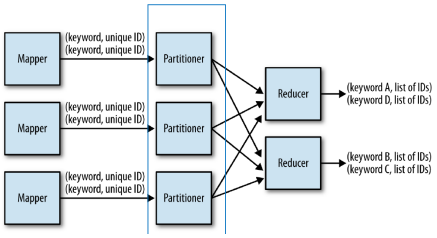
Map (key,record): insert record into a top ten sorted list if length of list is greater-than 10 then truncate list to a length of 10 emit(key,record with values of the list)

Reduce(key,record): sort record Truncate record to top 10 emit record

it works with a single reducer because it only gets 10 values per map node

11.7 Exercise 2 : Inverted Index

Given a set of Web pages [URL:ID,Content:page] build an inverted index [(k1,[u1,...,uk]),...,(kn,[u1,...,u])] where k is a keyword and u1,..,uk are URLs of the Web pages containing the keyword k



12 MapReduce Runtime Environment

12.1 MapReduce Program

A MapReduce program , referred to as a job consists of :

- A code for function Map and a code for function Reduce packaged together
- Configuration parameters
- the input , stored on the underlying distributed file system

Each MapReduce job is divided by the system into smaller units called tasks. **Map tasks (mappers)**: calling the map function over the pairs contained in one input chunk. **Reduce tasks (reducers)** : calling the reduce function over a subset of pairs generated by the intermediate level

12.2 MapReduce Input

Usually stored on a distributed file system. Other options are however possible. The file is partitioned. Each Map task processes exactly one chunk.

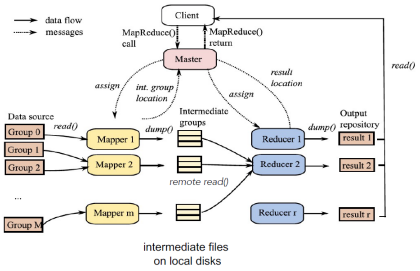
12.3 MapReduce Architecture

Master-slave architecture

The user program generates : One **Master** controller process (and node), which coordinates all the jobs run on the system by scheduling tasks and Mapreduce jobs are submitted to it. Then it generates some number of **Worker** processes at different nodes and they run either map or reduce tasks and send progress reports to the Master process and a given Worker process may run several tasks in parallel. In the end, it generates a **Client** node where the application is launched.

12.4 Processing a MapReduce job

The client starts the MapReduce process , it connects to a Master and transmits the map() and reduce(). The execution flow of the Client is then frozen. The Master creates a number of Map tasks and a number of Reduce tasks , then



assigns tasks to Workers and keeps track of the status of each Map and Reduce task.

Usually a Worker handles wither Map tasks or Reduce tasks

Map Worker (Mapper)

Usually one for each node where a group is stored. Assignment is based on the data locality principle. More than one group can be assigned to a single Mapper.

Trough an iterator like mechanism, it extracts pieces of data and applies the map() function. Intermediate pairs generated as output of the map function are locally stored (not by the DFS and often stored in partitions , one for each reducer. The Mapper then informs the Master of the location and sizes of each of these portions. The computation remains purely local , and no data has been exhanged between the nodes.

Reduce Worker (Reducer)

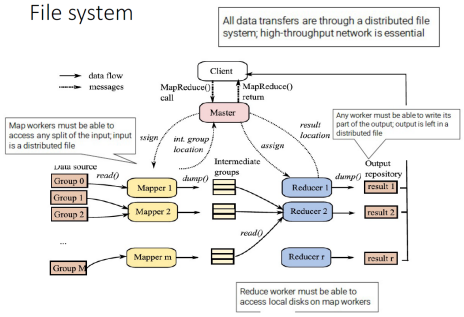
The number of Reduce tasks is supplied by the programmer , as a parameter R , along with a hash() partitioning function that can be used to hash the intermediate pairs in R groups. Grouped instances are assigned to Reducers by the Master thanks to the hash function. A Reduce task corresponds to one of the R partitions.

The Master sends to each Reducer the hash key , the addresses of intermediate buckets , and the Reduce function. Each reducer works as follows :

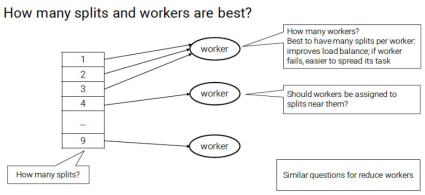
1. It reads the buckets from all the Mappers and sorts their union by the intermediate key (this now involves data exchanges between nodes , and the sort can be done in main memory or with the external sort/merge algorithm

2. it sequentially scans the intermediate the result and , for each key k2 , it applies the reduce() function on the bag of values ¡v1,v2..¿ associated with k2.

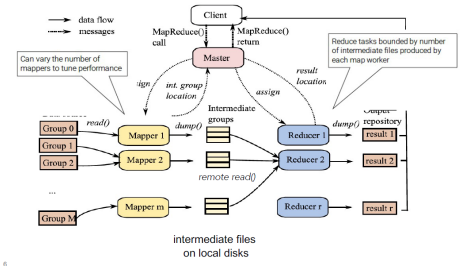
3. It sends the location of the result , in the DFS , to the Master



12.5 Partitioning and task assignment



Each map task processes a fixed amount of data (split),usually set to the distributed file system block size. Each map worker can process several mapper tasks; assignment to mapper nodes tries to optimise data locality. The number of reducer tasks is set by the user : assignment to reducers is done through a hashing of the key , usually uniformly at random.



12.6 Partitioners

As said , the user tells the MapReduce system how many Reduce tasks there will be , say R. The master controller picks a hash function that applies to keys and produces a bucket number from 0 to R-1. Each key that is output by a Map task is hashed and its key-value pairs are put in one of R local files. Local

files are organized as a sequence of pairs. Each file is destined for one of the Reduce tasks

12.6.1 Issues with default partitioners

The default partitioner assigns approximately the same number of keys to each reducer. But partitioner only considers the key and ignores the value. An imbalance in the amount of data associated with each key is relatively common in many text processing applications. In texts the frequency of any word is inversely proportional to its rank in the frequency table. The most frequent word will occur approximately twice as often as the second most frequent word , three times as often as the third most frequent word.

12.6.2 Customization of partitioners

We can specify a partitioner that : divides up the intermediate key space , assigns intermediate key-value pairs to reducers and n partitions $\rightarrow$ n reducers. Whatever algorithm is used each key is assigned to one and only one Reduce task.

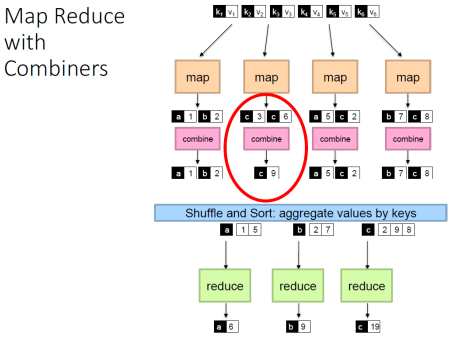
12.6.3 Sorting(reprise)

Partitioner sends to reduces according to sorting key order. Two steps : analyze (determines the ranges of values) and order (with partitioner relying on ranges to ensure data is totally ordered.

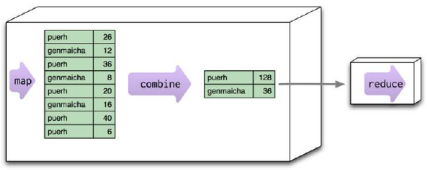
12.7 Combiners

12.7.1 Combine function

High communication cost between mapper and reducers. Some pre-aggregation could be performed to limit the amount of network data. The advantages of the combine function are : the reduction of the amount of intermediate data and the reduction of network traffic.

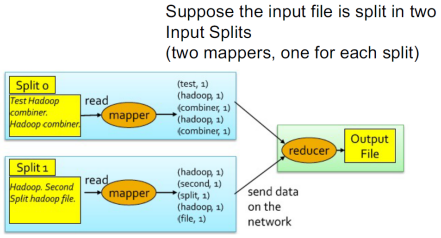


In many cases the same function can be used for combining as the final reduction. When the Reduce function is associative and commutative.



SUM

12.7.2 Word Count without combiners

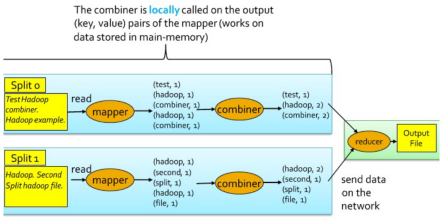


12.7.3 Word Count with combiners

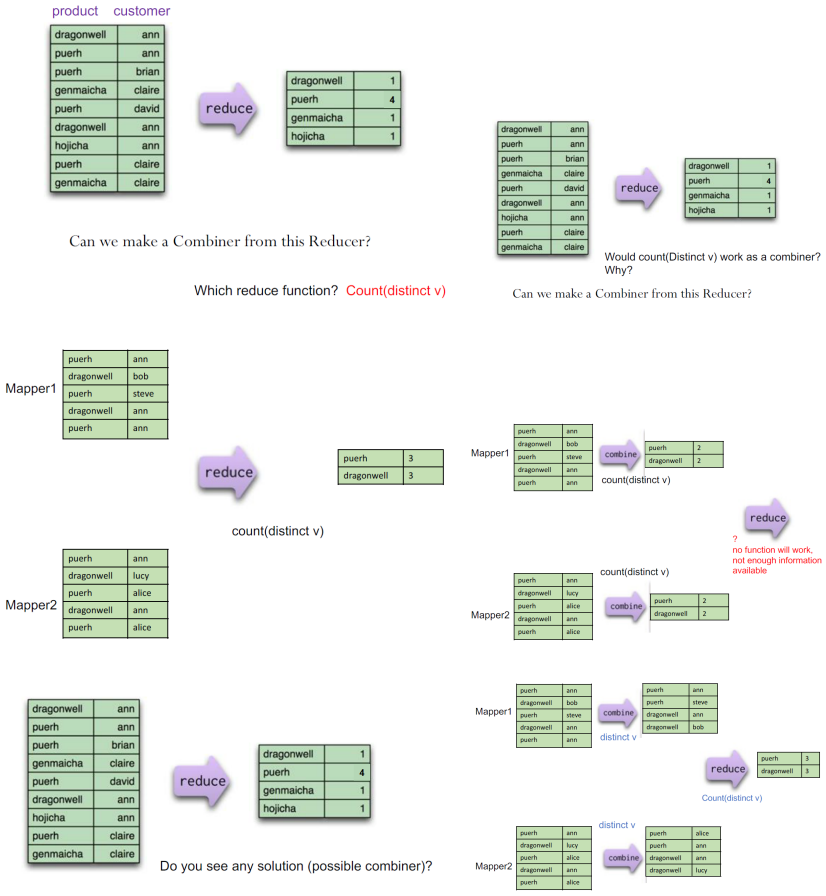
12.8 Questions and Answers

Reconsider the different variants of the word count alternatives now that the actual architecture is clear

No , since it is not associative. Do you see any solution ? Compute average by sum and count , that are associative.



Reducers and Combiners can differ!

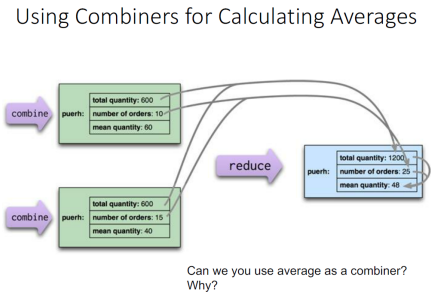


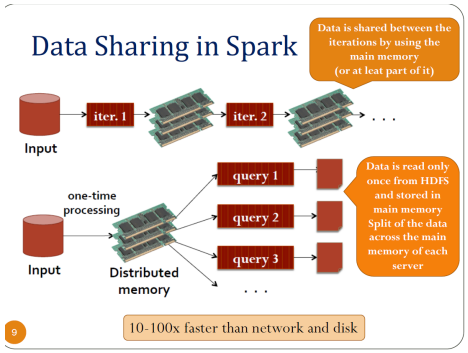
13 Large Scale Data Processing Frameworks : Spark

13.1 Introduction and Motivation

13.1.1 MapReduce limitations

Iterative jobs , with MapReduce , involve a lot of disk I/O for each iteration and stage.





Slow due to disk I/O , high communication , and serialization. Inefficient for : Iterative algorithms and Interactive Data Mining

Programming model

Hard to implement everything as a MR program. Multiple MR steps can be needed also for simple operations. Lack of control structures and data types.

Efficiency

High communication cost. Frequent writing of output to disk. Limited exploitation of main memory.

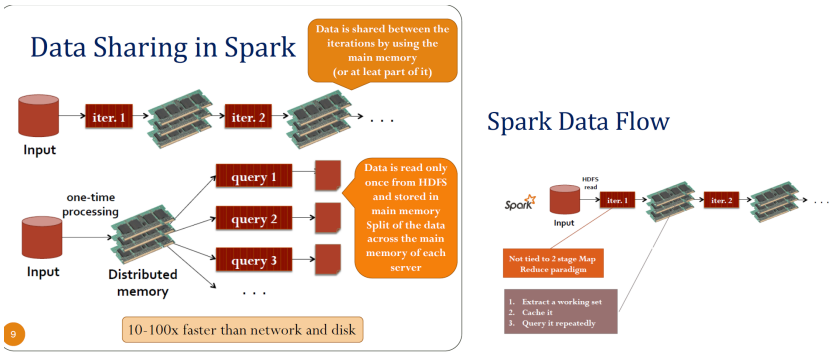
Real-time processing

A MR job requires to scan the entire input. Stream processing and random access impossible.

The solution to keep more data in main memory and enabling data sharing in main memory as a resource of the cluster is SPARK.

Everything you can do in Hadoop , you can also do it in Spark. But it relies on Hadoop (HDFS) for storage. Spark's computation paradigm is not just a single MapReduce job , but a **multi-stage , in memory dataflow graph based on** Resilient Distributed Datasets (RDDs)

Data are represented as RDD's. Partitioned/distributed collections of objects spread across the nodes of a cluster. Stored in main memory or on local disk. Spark programs are written in terms of operations on RDDs.



13.1.2 Spark Computing Framework

Providing a programming abstraction based on RDDs and transparent mechanisms to execute code in parallel on RDDs.

Manages job scheduling and synchronization. Manages the split of RDDs in partition and allocates RDDs' partitions in the nodes of the cluster. Hides complexities of fault-tolerance and slow machines.

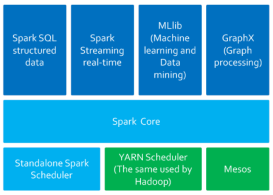
Map Reduce vs Spark

	Hadoop Map Reduce	Spark
Storage	Disk only	In-memory or on disk
Operations	Map and Reduce	Map, Reduce, Join, Sample, etc…
Execution model	Batch	Batch, interactive, streaming
Programming environments	Java	Scala, Java, R, and Python

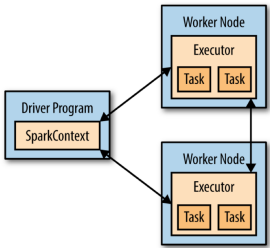
- Lower overhead for starting jobs
- Less expensive shuffles

13.2 Spark Main Components

An API with core features and extended modules for data manipulation in different languages. Spark supports any Hadoop-ready storage source (local file system , HDFS...) Spark provides a stand-alone cluster manager



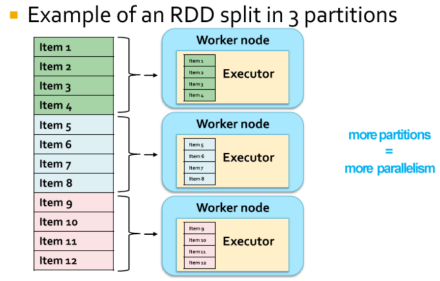
Spark Core contains the basic functionalities of Spark exploited by all components. Provides the API that are used to create RDDs and apply transformations and actions on them.



13.3 RDDs

RDDs are the primary abstraction in Spark. RDDs are distributed collections of objects spread across the nodes of a cluster. They are split in partitions and each node of the cluster is used to run an application contains at least one partition of the RDD that is defined in the application.

RDDS are stored in the main memory of the executor running in the node of the clusters on on the local disk of the nodes if there is not enough main memory. RDDs allow executing in parallel the code invoked on them. Each executor of a worker node runs the specified code on its partition of the RDD.



RDDs are immutable once constructed , the content of an RDD cannot be modified. Track lineage information to efficiently recompute lost data; for each RDD, Spark knows how it has been constructed and it can rebuild it if a failure occurs. This information is represented by means of a DAG connecting input data and RDDs.

13.4 Operators over RDDs

13.4.1 Creation

RDDs can be created :

- By paralellizing existing collections of the hosting programming language (The number of partitions is specified by the user)
- From(large) files stored in HDFS or any other file system(There is one partition per HDFS block)
- By transforming an existing RDD(The number of partitions depends on the type of transformation)

Programmer can perform 3 kinds of operations.

13.4.2 Transformations

Transformations are lazy operations on a RDD that return RDD objects or collections of RDD (map , filter , join) Transformations are lazy and are not exectued immediately , but only after an action has been executed. There are two kinds of transformations :

Narrow Transformations
They are the result of map,filter, and such that the result is from the data from a single partition only. An output RDD has partitions with records that originate from a single partition in the parent RDD. Spark groups narrow transformations as a **stage**

Wide Transformations
They are the result of groupByKey and reduceByKey The data required to compute records in a single partition may reside in many partitions of the parent RDD. All of the tuples with the same key must end up in the same partition , processed by the same task. Spark must execute RDD shuffle , which transfers data across clusters and results in a new stage with a new set of partitions.

Main transformations are : filter , map , sample , union , intersection , distinct , groupByKey , reduceByKey , sortByKey , join , cogroup , cartesian

13.4.3 Actions

Actions are synchronous operations that retun values. Any RDD operation that returns a value of any type but an RDD is an action. Actions trigger execution of RDD transformations to return values. Until no action is fired, the data to processed is not even accessed. Only action can materialize the entire process with real data , actions cause data retrieval and execution of all transformations on RDDs.

Main actions are : reduce , collect , count , first , take ...

13.5 Persistence

By default , each transformed RDD is recomputed each time you run an action on it , unless you specify the RDD to be cached in memory. RDD can persisted on disks as well. Caching is the key tool for iterative algorithms. Using persist() , one can specify the Storage Level for persisting an RDD.

Caching allows us to avoid iterative recomputation of data.

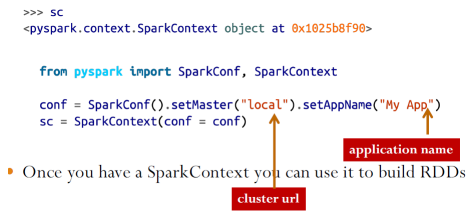
RDD persistence is especially useful for iterative algorithms and fast interactive use. Fault-tolerant : if any partition of an RDD is lost , it is recomputed by applying the same transformations that created it.

The method unpersist() allows to manually remove objects from cache , being handled by the Java garbage collector.

13.6 Spark Programs

13.6.1 Spark Context

SparkContext is the basic entry point to the Spark runtime environment an underlying file system. SparkContext represents a connection to a computing cluster.



13.6.2 RDD Transformations

Transformations take one RDD as input and return another RDD as output.

input = sc.textFile(”log.txt)

errorsRDD = inputRDD.filter(lambda x : ”error” in x) warningsRDD = inputRDD.filter(lambda x: ”warning” in x) badLinesRDD = errorsRDD.union(warningsRDD)

They define the logical structure of the dataflow , but they do not actually trigger any computation.

13.6.3 Persitency and Caching of RDDs

The previous dataflow program access the badlinesRDD twice to perform two actions : a count() and a take() To avoid repeated computation of the badlinesRDD from the input RDD , we can explicitly persist the former within the cluster’s main memory. Just call persist() on the RDD before the actions are invoked.

Passing functions and lambda's

- Java (either anonymous inner classes or named classes)

```
JavaRDD<String> inputRDD = sc.textFile("log.txt");
JavaRDD<String> errorsRDD = inputRDD.filter(
  new Function<String, Boolean>() {
    public Boolean call(String x) { return x.contains("error"); }
  }
});
```

Table 3-1. Standard Java function interfaces

Function name	Method to implement	Usage
Function<T, R>	R call(T)	Take in one input and return one output, for use with operations like map() and filter().

- Scala

```
val inputRDD = sc.textFile("log.txt")
val errorsRDD = inputRDD.filter(line => line.contains("error"))
```

- Python

```
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
```

- Java 8 (lambda expressions)

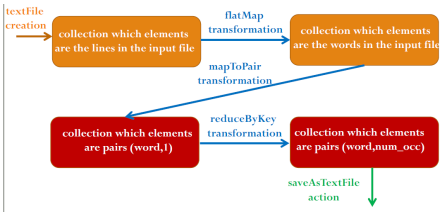
```
RDD<String> errors = lines.filter(s -> s.contains("error"));
```

- Actions take an RDD object as input and perform a final operation to obtain a non-distributed, either mutable (e.g., a list) or immutable (e.g., a file) object as output

```
print "Input had " + badLinesRDD.count() + " concerning lines"
print "Here are 10 examples:"
for line in badLinesRDD.take(10):
    print line
```

- Actions trigger the actual computation of a dataflow

13.7 Spark program : Word Count




```
import sys from pyspark
import SparkContext, SparkConf

# create Spark context with necessary configuration
sc = SparkContext("local","PySpark Word Count Exmaple")

# read data from text file and split each line into words
words = sc.textFile("input.txt").flatMap(lambda line: line.split(" "))

# count the occurrence of each word
wordCounts = words.map(lambda word: (word, 1)).reduceByKey(lambda
a,b:a +b)

# save the counts to output wordCounts.saveAsTextFile("output/")
```

13.8 Spark Transformations and Actions (Low-Level API)

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
mapPartitions(*func*)	Similar to map, but runs separately on each partition of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numTasks*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance. Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numTasks argument to set a different number of tasks.
reduceByKey(*func*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configuarble through an optional second argument.

Transformation	Meaning
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numTasks*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean `ascending` argument.
join(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`.
cogroup(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called `groupWith`.
cartesian(*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(*command*, [*envVars*])	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
coalesce(*numPartitions*)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
repartition(*numPartitions*)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
repartitionAndSortWithinPartitions(*partitioner*)	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling `repartition` and then sorting within each partition because it can push the sorting down into the shuffle machinery.

Action	Meaning
reduce(func)	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(n)	Return an array with the first n elements of the dataset.
takeSample(withReplacement, num, [seed])	Return an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(n, [ordering])	Return the first n elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(path)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(path) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(path) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded usingSparkContext.objectFile().
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(func)	Run a function func on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. Note: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. See Understanding closures for more details.