FS2024 Systems for Data Science (UZH) – Summary

Chapter 1: Introduction

Big Data: is anything that doesn't fit on one machine. No clear definition, but usually it fulfills the 4 V's

-**Volume**: huge amount/scale of data (it is estimated that we will produce 181 Zettabytes per year by 2025)

-**Variety**: different forms/sources/formats of data

-**Velocity**: a lot of data is produced in very high frequency (by e.g. large network of sensors)

-**Veracity**: noise in data (have to cope with inaccuracies)

Challenges of Analyzing Big Data: in Data Science

-**Heterogeneity**: Most algorithms need uniformly structured data. But in practice, relevant data is unstructured, and has multiple sources. Context (e.g. given by metadata) has much relevance. But how do we get this metadata and how do we keep it?

-**Inconsistency and Incompleteness**: Often must handle noisy data, since it comes from multiple sources with different reliability → uncertainty errors and missing values

→redundancy helps with this problem, but managing other errors is still difficult

-**Scale**: Data volume is increasing faster than CPU and other computing speeds

→ processing needs to me massively parallelized

-**Timeliness (i.e. computation speed)**: Many applications require real-time (fast) responses (e.g. frau detection, autonomous driving). But often we cannot scan the whole dataset in time.

→ need techniques for partially computing results

-**Privacy and Data Ownership**: Sensitive data is protected by strict laws (e.g. health records). Using data appropriately can be difficult. E.g. anonymization is harder than it seems: Combining datasets can deanonymize the information.

-**Interpreting Data**: Presenting data in an understandable and neutral/fair way. Interdisciplinary cooperation is necessary for analyses, but there are still problems like securing the data origin, access control, and more.

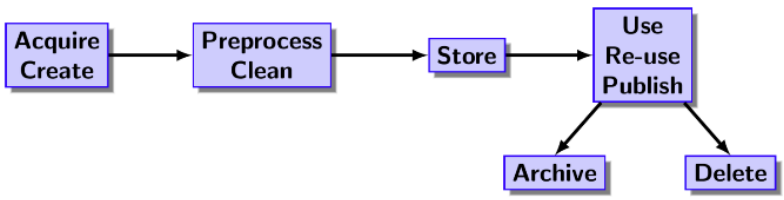
Chapter 2: Doing Data Science

Data Science: is abstractly about systematically extracting knowledge from data, based on a predefined workflow/pipeline. This is usually based on the *scientific method*:

-1. Ask a question, 2. Do background research, 3. Construct a hypothesis, 4. Test the hypothesis with an experiment, 5. Analyze data to draw a conclusion, 6. Communicate results

Data Life Cycle: doesn't have a fixed definition, and might depend on the specific domain/dataset. But usually it will look like this:

-Extension: the life cycle might go beyond the dataset, thus instead call it "data science life cycle". This additionally includes artifacts such as data, code, workflow, computational environment, and more.



Chapter 3: Brief History of Data Management and Processing

Evolution of Data Management: 20 years ago, it was enough to buy a *Relational Database Management System* (**RDMS**).

-**Data Cubes and Online Analytical Processing (OLAP)**: were an early attempt at storing business intelligence (along 3 dimensions: customers, time, products)

→ This was good for ad-hoc search, but bad for data analysis

Instead of **Scale-Up** (buy bigger computers), companies chose to **Scale-Out** (buy more smaller/conventional computers).

But conventional computers are not very reliable. Thus, it is necessary to use redundant storage/backups and specialized software (e.g. Hadoop, Spark/Map Reduce, NoSQL), so we can store large amounts of data.

PART I: Data Management

Chapter 4: Principles: Scalability

(Flat) Files: are commonly used to store data on a single machine/server.

-files are easy to read and write → but only **sequentially** (also often requires writing code for it)

-don't provide optimized access → issues with inconsistency, concurrency (several instances existing at the same time), and integrity

→ often efficient access to subsets of data is *necessary* during analysis

→ This brings the idea for **databases**, i.e. infrastructure to manage data

Relational Systems: are a type of database, which organize data in structured tables with rows and columns, using relationships between tables.

-**advantages**: mature technology, well-defined interfaces (SQL), well-organized and structured (i.e. provides metadata in form of schemas), supports multiple users (built-in user synchronization for data integrity), good at basic statistics and reporting, well-tuned query processing engines (i.e. efficient preprocessing, delivers data in desired format)

-**disadvantages**: only scales to medium sized datasets (does not scale-out well), limited functionality for analytics, problems with handling semi-structured data

Scalability: eventually there will always be too much data for files or databases on a single machine.

→switch from scaling-up to scaling-out becomes *necessary*

We have to think about how we can handle data being distributed across multiple machines.

Sharding: is a way of handling data across multiple machines. Split a large database into independent "shards", which then are distributed across multiple machines.

-each data record is assigned to exactly one node (need some form of directory)

-data is ideally close to the user accessing it

-is useful for load-balancing, i.e. distribute data according to workload

-many systems offer auto-sharding (used to be done manually)

→very efficient, but not resilient

System Failures: happen constantly. A 2% yearly failure rate for one system is optimistic.

→ *necessary* to keep copies of the data

Master-Slave Replication: Replicas of each data record are stored on multiple nodes. One master node is responsible for propagating database updates to the slave nodes. All updates are made to the master node.

-**advantages**: read can be done from any node (master/slave) → parallelizable

→read operations are efficient and resilient

-**disadvantages**: master node is bottleneck for updates (simple point of failure), keeping replicas is not feasible for large amounts of data, inconsistency is a big vulnerability ("dark" side of replication)

Peer-To-Peer Replication: multiple nodes each hold a copy of the entire database. They can independently read and write data. Changes on any node are propagated to all others, to keep consistency.

-efficient updates (since any node can do them)

-further improve performance by adding more nodes to distribute workloads (update/read operations)

-*severe* issues with inconsistency

Sharding with Replication: To improve performance and scaling, we can combine the replication architectures above with sharding.

-**master-slave with sharding**: split database into smaller manageable shards, each using a master slave approach, independently across multiple machines. One node can be both master and slave, but for different shards.

→ better load balancing, and permits scaling the data storage

-**peer-to-peer with sharding**: split database into shards, each using peer-to-peer approach, independently across multiple machines.

→ is one of the most complicated architectures to implement

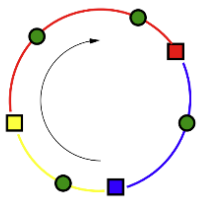
→very flexible approaches, but still problems with consistency

Hashing: can be used to implement the above database architectures (specifically: peer-to-peer sharding) Apply any hash function to the key (=unique identifier) of a data record. Then insert data record into bucket (= type of storage location, which can store several items) located at the location indicated by the hash value of the key. → we call this a **hash table**. Given a key, we can look up the data record.

-when distributing the hash table, we need to know which key is stored on which server node.

→ having one central directory would introduce a single point of failure (not peer-peer anymore)

Consistent Hashing: Describes a hashing schema to determine key locations. It is useful for efficiently handling addition and removal of nodes to a database architecture.



-keys of data records and ID's of nodes (e.g. MAC addresses) are hashed with the same hash function.

-domain of the hash function is wrapped around into a circle

-each hashing value corresponds to one data record/bucket

-introduce several nodes (i.e. servers) to the hashing circle. Each node is responsible for the range of hashing values from its own down to the value of its predecessor

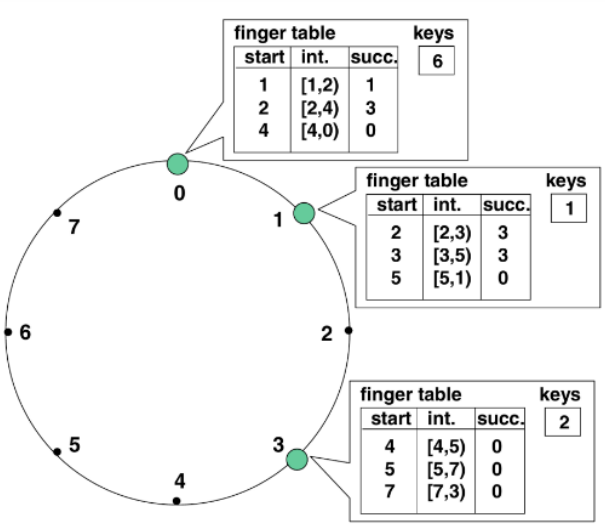
More formally (not too much in detail): Assume we have a circle of length modulo 2^m (here m=3). Then nodes can have IDs (i.e. keys) with hashing values between 0 and $2^m - 1$.

-**successors**: Inserting a new data record with key (hashing value) **k** is assigned to the first node with ID (i.e. hashing value) **≥ k**. This node, we call the *successor*.

-**joining node**: splits the range of an existing node. Values smaller than the hash value of the new node, are assigned to the new node.

-**replication**: for resilience, replicate every record n times. The additional n-1 copies are stored in the n-1 successor nodes of the responsible node.

-**Problems**: above description assumes all nodes know each other. But with constantly leaving and joining nodes (dynamic peer-to-peer), keeping track of all nodes is unrealistic.



Chord: is a protocol, that uses consistent hashing in a dynamic peer-to-peer network. Scales to joining nodes. The method won many awards and is used in many well-known systems.

-uses SHA-1 as has hashing function (160-bit values), with large image space (2^{160} possible values). This makes collisions very unlikely.

-distributes its values quite evenly (good load balancing)

-assumes inter-node communication is bidirectional and reliable

-in principle, every node only needs to know its successor, but this is inefficient, and a leaving node could break the chain. Also, for every node to know every other node is infeasible (constantly join/leave)

for node 0 we have:

i	distance (start)	successor
1	$0+2^{1-1}=1$	1
2	$0+2^{2-1}=2$	3
3	$0+2^{3-1}=4$	0

-**Routing (i.e. finding responsible node)**: any node n (=hash value) maintains a *finger table*. It has at most m entries (assuming hash space of size 2^m), where the i-th entry (i.e. "finger") points to the first node that succeeds n by at least 2^{i-1} in the hash value space. We write: n.finger[i] = successor(n+2^{i-1}), with all computations in modulo 2^m. The first finger is the immediate successor of n.

→ each node only stores information about a few close nodes. This might not be enough information to find the successor of any arbitrary key (ex: node 3 doesn't know successor of k=1).

--**but** each node *needs* to be able to contact any other node with a request.

--**finding successor node of key k**: n tries to find a node whose id (i.e. hash value) is closer to the key k. Node n looks in its finger table, and chooses the node j which most immediately precedes k. Then n asks j about the responsible node for k. Repeat the process for node j, until we reach a node that knows the most immediate successor of k (i.e. if k is in between a start and corresponding successor value in the finger table of node j).

-finger tables divide the address space: interval[i] = [distance[i], distance[i+1]) (excluding last val)

-to find successor, we send the request about k to the successor node of the corresp. interval

→ with high probability, the search can be done in **O(log(n))**

--maintaining finger tables in dynamic network is difficult. Needs to use stabilization strategy

→ each node n also stores its predecessor n' (to simplify updates)

--**Joining (of new node)**: new node n needs to know at least one existing node n'. n takes immediate successor of n' as its own successor. Run: **n.join(n')**

n.join(n')
  n.predecessor = **nil**
  n.successor = n'.find_successor(n)

--**Stabilizing**: every node periodically runs the **n.stabilize()** operation.

n.stabilize()
  $sp = n$.successor.predecessor
  if sp *between* n *and* n.*successor* **then**
    n.successor = sp
  n.successor.notify(n)

Where:

n.notify(n')
  if n.*predecessor* == ***nil or*** n' *btwn* n.*predecessor and* n **then**
    n.predecessor = n'

-also, each node periodically refreshes its finger tables with the operation **n.fix_fingers()**

n.fix_fingers()
  i = random index > 1 in finger[]
  finger[i].successor = find_successor(finger[i].start)

--**Failure to find successor**: each node n maintains a **successor list**, which contains the r nearest successors of node n. If a successor lookup fails, then replace the successor of n with the first *live* successor in the list.

→ needs a modified stabilize function, which now also maintains the list of r successors

-chord assumes each node has at least one live successor in its successor list

→ to ensure that, choose appropriate length of successor list, and call stabilize often enough

--**Replication of data records**: the additional copies of the data is stored exactly at the nodes in the successor list. Use the list to store the copies. → Chord does not manage replication itself (but provides all necessary information for it to be done by a layer above)

-original Chord falsely claims **eventual reachability** (= every node is reachable from any other node, i.e. given enough time and no further disruptions, Chord can repair its defects)

-BUT: Some corner cases lead to timing issues when determining liveness of nodes. Example: if a node joins, and its successor fails, before the new node was stabilized, then new node cannot be reached.

→ **need improved Chord protocol** (modify join, stabilize, and notify (= now called rectify) operations)

-further assume node has failed if it takes too long to respond

-failed nodes can rejoin the network later, after recovering.

Improved Chord: has modified join and stabilize protocols.

-**improved join** (n joins through n'): n asks n' to search for a predecessor newPrd, such that n is between newPrd and newPrd.succList[1]. n makes sure newPrd is live, then sets its predecessor and successor list in one go.

-**improved stabilize**: does two separate stabilizations, from the view of the successor, and then the predecessor sp

```
n.join(n')
    newPrd = n'.find_predecessor(n)
    l = newPrd.query(succList)
    if query returns before timeout then
        n.succList = l
        n.predecessor = newPrd
    else
        abort
```

```
n.stabilizeFromSucc()
    s = n.succList[1]
    sp = s.query(predecessor)
    sl = s.query(succList)
    if queries return before timeout then
        n.succList = s ⊕ sl[1..r − 1]
        if sp between n and s then
            n.stabilizeFromPrd(sp)
    else
        n.succList = n.succList[2..r] ⊕ n.succList[r] + 1
        n.stabilizeFromSucc()
    n.succList[1].rectify(n)
```

```
n.stabilizeFromPrd(sp)
    sl = sp.query(succList)
    // if new successor is dead, don't change
    if query returns before timeout then
        n.succList = sp ⊕ sl[1..r − 1]

n.rectify(n')
    if n' between n.predecessor and n then
        n.predecessor = n'
    else
        n.predecessor.query(isLive)
        if query does not return before timeout then
            n.predecessor = n'
```

→ **improved Chord was proven to be correct**. Checking liveness of nodes solves subtle timing issues. Also, during join, more information than just the successor is exchanged (better resilience). Stabilize is run from predecessor and successor nodes. No changes are made to the fix_fingers function.

Checksum: is a technique to verify correctness of data (to e.g. see if a node is malfunctioning, but still sending signals). Run data through a previously specified function, to create a checksum. Receiver runs data through the same function and compares it to the received checksum.

-**on disk storage**: each sector on a disk has some additional bits with its own checksum (depends on stored data). If read data does not agree with checksum, then we get an error.

-example: parity of all bits in a sector (set 1 or 0 as checksum to make number of 1-bits in sector even).

-but: only able to detect a single corrupted bit. Several flips might even it out again.

-using n independent bits, the chance to miss an error is $\frac{1}{2^n}$

data block 1:	11110000
data block 2:	10101010
data block 3:	00111000
parity block:	01100010

Parity Blocks: are a way of using some additional storage, to detect and correct errors in an efficient way.

-split the data into blocks of the same length. Introduce a parity block of the same length. Can recreate corrupted data from <u>one</u> block by noting that each column should have an even number of 1's.

→ can only fix errors in **one known block** (does not work for errors in multiple/unknown blocks).

→ parity block becomes a bottleneck (every update affects the parity block)

BCH Codes: are codes, which can detect and correct multiple errors in messages.

-for k blocks of content, add $2t$ parity blocks, resulting in a total of $k + 2t = n$ blocks.

-with any $n - t$ blocks we can reconstruct the whole content. If we know which blocks are corrupt, we only need $n - 2t$ correct blocks for full reconstruction.

Well known examples of BCH codes are Reed-Solomon (RS) codes.

RS (Reed-Solomon) Codes: use polynomials over finite fields, to encode/decode data for error correction.

-denoted by (n, k), where each codeword consists of n symbols (i.e. blocks), each of length m bits. The first k symbols are the original data (i.e. $2t = n - k$ parity blocks).

-remember that a polynomial of degree d is uniquely defined by $d + 1$ points

-**define polynomial**: partition the file into $d + 1$ parts, which each consist of bit sequences. Interpret each sequence as a large number. Then take the polynomial that is defined by the points with x coordinates $0, \dots, d$ and y coordinates of the sequence number values.

-**oversample**: points from the above polynomial (i.e. get values for x-coord. $d + 1, d + 2, \dots$)

→ with any $d + 1$ points left, we can still reconstruct the whole file

-**problems**: in practice we have polynomials of very high degree (e.g. d=29), for which we need to use Lagrange interpolation to find the polynomial. Also, for now, we can only handle erasures (missing values or known corruption). Cannot correct if we don't know which values are incorrect.

--**Error Correction (but do not know location of error)**: Assume we have only access to points of the form (x_i, r_i), where r_i is *not necessarily* equal to y_i. Now assume we have an **error polynomial $\boldsymbol{E(x)}$**, such that $E(x_i) = 0$ if $r_i \neq y_i$ (i.e. it knows which r_i are wrong).

-note that is holds: $P(x_i) * E(x_i) = r\_i * E(x_i)$. This is true, since whenever r_i is wrong, $E(x_i)$ evaluates to zero (and 0=0). Otherwise, if r_i is correct, then by definition $r_i = y_i = P(x_i)$, and equality holds.

-**Define polynomial $\boldsymbol{Q(x) = P(x) * E(x)}$.** Then $Q(x_i) = r_i * E(x_i) \Rightarrow \boldsymbol{Q(x_i) - r_i * E(x_i) = 0}$ gives us a set of equations, to get the coefficients of $Q(x)$ and $E(x)$.

-To retrieve P(x) (which is the primary goal!), simply perform **polynomial division $\boldsymbol{Q(x)/E(x)}$**.

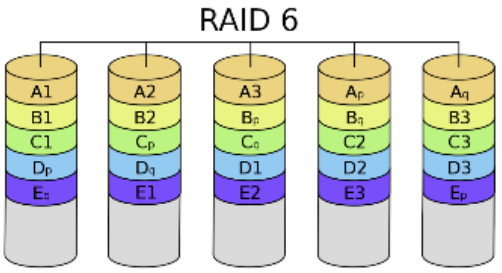
→this method can **correct up to t errors** in blocks of length $n = k + 2t$ (i.e. when $P(x)$ has degree $k - 1$, and $E(x)$ has degree t)

→ take polynomials in discrete domains (e.g. on finite fields), since floats are expensive to handle accurately

-RS codes are very well established (exist since 1950s), and are commonly used in e.g. CDs, QR codes, ...

Redundant Data Storage: is necessary for the case that individual whole data centers lose all their data e.g. burn down. Having 3 copies of every data record is quite common for large companies such as Facebook.

--**HDFS (Hadoop File System) RAID**: uses RS(14,10), developed by Facebook.



-Files are split into "stripes", each consisting of 10 blocks of data (256mb each) and 4 additional parity blocks. One stripe is distributed among different nodes (see figure: B is one stripe)

-node failures can be *expensive*. If one node fails, we loose one block from each stripe stored on that node. For reconstruction, RS-decoding needs at least 10 blocks of each affected stripe. But commonly more then 20 nodes fail per day, each storing 15 Terabytes of data.

-**Locally Repairable Codes (LRC)**: solve the problem of expensive node failures.

-categorize the 14 blocks of a stripe into 3 groups: two groups of 5 data blocks each, and one group of parity blocks. For each group, create local parity blocks: S_1, S_2, S_3

-each parity block is computed by applying bitwise XOR operation on all corresponding data blocks, e.g. $S_1 = X_1 \oplus X_2 \oplus X_3 \oplus X_4 \oplus X_5$ (can directly compute S_3 from S_1 and S_2)

-if a block is missing, reconstruction only needs 5 other blocks of that stripe. We can easily compute the missing block, e.g. $\boldsymbol{X_1} = \boldsymbol{S_1} \oplus X_2 \oplus X_3 \oplus X_4 \oplus X_5$.

-S_1 and S_2 also have to be distributed, which increases storage overhead (14 to 16 blocks)

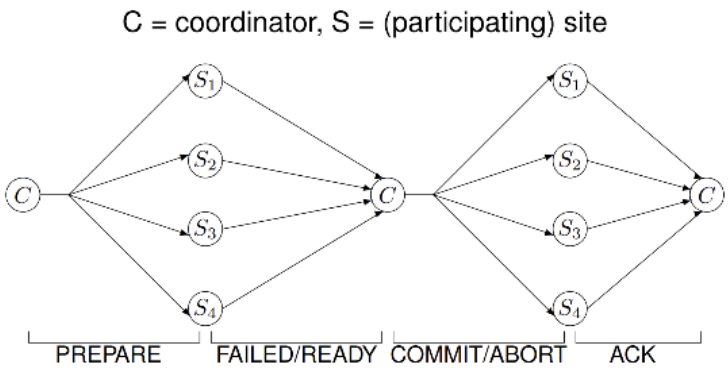
-repair time goes down (only need to ship half the data)

-if we lose more than 1 block in the same S_i – group, still need to ship 10 blocks. But this is generally not too common (usually single block/node fails).

Chapter 5: Principles: Consistency

ACID-Style Consistency: in relational database systems ensures strong consistency, by guaranteeing that we never read inconsistent data, and operations leading to inconsistencies will be aborted.

Two-Phase Commit (2PC): distribution protocol that ensures no inconsistent data updates are committed.



-have coordinator C, and data sites S_i. C sends a *commit* (finalize update change) message, *only* if *all* S_i respond with *ready*.

-**site failure**: if C detects failure of S, before S responds with *ready*, then C sends *abort* to all sites S_i. If S fails after responding *ready*, C ignores S, and S sorts out itself when back online.

-**coordinator failure**: major vulnerability of 2PC. If S has not sent a *ready* message yet, then it can safely abort the protocol, i.e. the recovered C automatically aborts, since it gets either *failed* or no message from S. If S has already sent *ready*, then S must wait for C to recover (could end in *commit* or *abort*).

-**failure of communication link**: For loss/corruption of messages, use robust communication protocols (e.g. TCP/IP). For loss of link between sites, reroute message through other sites. If the network suddenly partitions and not all actors are in the same partition, then the other nodes are treated as failed, which could lead to blocking!

→ if one participant in 2PC fails, the transaction aborts, and could even fail. Thus, it is usually not used for keeping replicas consistent. Coordinator is single point of failure (vulnerability)

Quorum Consensus: is a protocol in distributed databases for updating replicated data records.

-assign non-negative weight to every replica of a data record (e.g. depending on its nodes' reliability)
-**Consensus decision**: When performing a read and write operation on a data record A, we need to access a group of nodes/replicas, whose sum of weights is given by W(A), and which fulfill the **quorum consensus conditions**: $\mathbf{2} * \boldsymbol{Q_w}(\boldsymbol{A}) > \boldsymbol{W}(\boldsymbol{A})$ **and** $\boldsymbol{Q_r}(\boldsymbol{A}) + \boldsymbol{Q_w}(\boldsymbol{A}) > \boldsymbol{W}(\boldsymbol{A})$. Read and write operations are assigned integer values for Q_r and Q_w.
→ this guarantees that a write blocks any other write, and a read has at least one new copy.

Paxos: is a **consensus-finding protocol**. Processes, called *acceptors*, running on different nodes, want to agree on a value for a data record. Essentially based on spreading around information for proposer (of update).
-**Guarantees**: acceptors will agree on a *single* value, and if majority of nodes is live,Paxos will *not block*.
-election of new leader might be imperfect (several think they are the leader), but it will still work
-**Assumptions**: differentiate proposals with **unique timestamps** (with hostID). A higher timestamp means the message is newer → we can order the proposals (lower timestamps loose to higher)
-**Phases**: **(a) for proposer messaging acceptors, and (b) for acceptor messaging proposer**. Then do
-**Phase 1(a): Prepare**. Proposer sends timestamp t for new proposal to all acceptors.
-**Phase 1(b): Promise**. Acceptor receives *prepare* with timestamp t. Case1: t is highest value so far, then return promise to not accept timestamps $< t$. Case2: t is highest value so far, but another proposal (t', v') was already accepted (see 2(b)), additionally return (t', v'). Case3: else, ignore prepare message.
-**Phase 2(a): Recommend**. Proposer continues, if it receives 1(b) messages from majority of acceptors. Case1: *free*. If none of the messages contain 2(b) part, send out (t, v) from 1(a) with arbitrary v (since no conflicting proposals have been accepted by acceptors yet). Case2: *forced*. If some messages contain 2(b) part, then take t_{max} the highest found timestamp, with value v_{max}, and send out (t, v_{max}) for acceptance. (proposer has reached phase 2(b), and is trying to finalize acceptance or rejection)
-**Phase 2(b): Accept**. Acceptor decides whether to accept recommendation (t, v) from 2(a) message. Case1: If acceptor has promised before **not** to accept proposals with timestamp $< t_p$, but $t < t_p$, then ignore the recommendation. Case2: If no such promise was made, send *accept* (t, v) to proposer.
-**Phase 3: Confirm/Learn**. After receiving 2(b) messages from majority of acceptors, proposer informs everyone about value v. [This phase can also be left out, with **modification**: acceptors send 2(b) message to everyone, instead of proposer. As soon as acceptor receives enough 2(b) messages, they assume/know that v is accepted.]
-for practical implementation, consider more details
→ **Resilient**: if acceptors fail or network partitions, we can still continue, as long as we have a majority
-**Leader Election**: is an important application of the Paxos protocol. When a leader fails, nodes will wait some time until they send a proposal for a new leader. Find single new leader even if many proposals.
-**Network Partitioning**: if no partition has a majority, Paxos blocks. Though improbable, not foolproof.

CAP Theorem: In distributed systems you can never have all three of **C**onsistency (same strong consistency as on a centralized system), **A**vailability (system never blocks), and **P**artition tolerance (system can handle network partitioning). A system can **only have 2 at most**.
-usually, distributed systems will try to have **C and A**
-in large scale systems partitioning of the network cannot be prevented. Could lead to blocking. Thus either consistency or availability need to be dropped.
-could also resolve the issue by **dropping ACID-level consistency**. Some large companies (Amazon, Google, Facebook) do this, and choose **weaker levels of consistency** (like BASE)
-this holds for the general case, but **special cases can still be solved** (see CALM)

BASE: **B**asically **A**vailable, **S**oft-State, and **E**ventual Consistency. It always works, is not necessarily consistent all the time, but will eventually be consistent.

-**Eventual Consistency**: in some steady state, all updates will propagate through the system and reach consistency. But we might see inconsistencies if we are still issuing updates.
-writes should not be blocked in case of a network partition
-update is done locally and is propagated to all replicas asynchronously (messages might arrive out of order). Timestamps do not work, since clocks will not always be perfectly synchronized.
-usually new versions arrive last, so the system figures out itself how to reconcile messages (**syntactic reconciliation**). Application deals with mistakes (**semantic reconciliation**)

Vector Clocks: reconciling a system, by tagging values with *vector clocks*. That is, a vector $< t_1, \dots, t_n >$ with clock values (counter) for each node (n nodes total). Every time a node makes a change, it updates its own counter on the vector clock of that value.

day = Tuesday
$VC_2 = \langle A:1, B:1, C:0, D:0 \rangle$

-a vector clock is $\boldsymbol{VC_x(s_1, \dots, s_n) < VC_y(t_1, \dots, t_n)}$ strictly larger, if all clock values larger of equal to the smaller vector clock values (i.e. $s_i \leq t_i$ for all i), e.g. $(5,6,7) > (0,6,1)$
-**Semantic Reconciliation**: if e.g. $VC_1 = (1,2,3)$ and $VC_2 = (3,2,1)$, then the vector clocks have a conflict. Then at some point a node must reconcile VC_1 and VC_2. It chooses a value and updates the vector clock to the max of each entry in VC_1 and VC_2 and increases its own counter by 1, e.g. $VC_3 = (3,2,4)$. Then VC_3 supersedes the vector clocks VC_1 and VC_2.
-**implementation**: only non-zero components of vector clocks. Still, VC tend to grow, if many updates are made. Thus, systems usually prune VCs by removing oldest components→possibly cause conflicts

Partial Quorum Consensus: is an adapted version of Quorum Consensus for eventual consistency, instead of strong consistency. For simplicity assume N nodes, each with weight 1. Then take the requirements for consensus to be $\boldsymbol{2 * Q_w \leq N}$ **and** $\boldsymbol{Q_r + Q_w \leq N}$.

-need to contact fewer nodes, less latency, but we risk reading outdated data
-difficult to find the right trade-off/values for this consensus. Usually use expected latency or consistency (probabilistic models). E.g. **Probabilistic Bounded Staleness (PBS)** uses probability that two quorums overlap: $1 - \binom{N-Q_w}{Q_r} / \binom{N}{Q_r}$ (but this requires more steps)

CALM (Consistency As Logical Monotonicity): principle suggests that with logical monotonicity a distributed system can achieve eventual consistency without additional coordination. Idea monotonicity: increasing input only increases output. Identifies subclasses which fulfill all 3 properties from CAP theorem.

-**formally**: Problem P, Input S, output $P(S)$. A problem has a consistent and coordination-free implementation, **iff** it is monotonic, i.e. for any inputs $S1 \subseteq S2$: $P(S1) \subseteq (P(S2)$.
-powerful, since many systems involve much non-determinism. CALM only considers total outcome
-**tombstoning** makes process monotonous, by marking items to be deleted (much later)→monotonous
-**Distributed Deadlock**: is an example of a **monotonic process**. Want to identify cycle in a waits-for graph, but the graph is scattered across several machines. Machines exchange information to detect cycles. Each machine sends copies of graph edges (e.g. T1→T2) to others (or alternatively to coordinator). Each machine then constructs larger graph, which eventually is the global graph.
  -if any machine detects a cycle in partial graph, does not have to communicate to confirm.
  -each machines output grows monotonically with input (i.e. more edges)
  -machines agree when all information was shared.

-**Distributed Garbage Collection**: is an example of a **non-monotonic process**. Have graph distributed over several machines, and we want to identify disconnected (from root) objects (vertices of the graph). Machines exchange information about edges (e.g. T1→T2). Final decision cannot be made locally.

-each machine only holds set of currently disconnected objects (output), which gets smaller, as the input edges grow

Confluence: A program on a single machine is confluent, if it produces same output (intermediate results can be different e.g. depending on batching order) for any non-deterministic ordering and batching of a input set. Can be interpreted as deterministic function from sets to sets.

-if output always grows monotonically, **ordering is irrelevant** (simply merge all encountered outputs)

-**deletions**: to keep monotonicity, handle deletion separately from additions. For final state (*checkout operation*), simply subtract deletion set from addition set.

-checkout operation is last and must be controlled globally → coordination is necessary here.

Chapter 6: Concrete Systems: File Systems

Introduce file systems, which implement the previously introduced concepts: GFS/HDFS, GlusterFS

GFS (Google File System): was implemented due to the need of distributing huge files on machines which fail often. But most operations are append-only writes, and sequential reads→only need weak consistency.

HDFS (Hadoop Distributed File System): open-source and based of GFS. Uses **master-slave architecture**.

-File: is split into blocks if size 128mb. Large blocks keep metadata small.

-**Master Node**: manages metadata (i.e. location of data blocks and active DataNodes), maps blocks of files to DataNodes, responsible for failure detection. → low load for master.

-**Slaves**: slave DataNode(s) store data blocks and process read and write operations.

-**Metadata**: (is also cached by clients). **Heartbeat**: informs *NameNode* (i.e. any node) that a DataNode is working properly. **Blockreport**: informs a NameNode of block IDs stored on a DataNode.

-**Write File**: 1. Client requests write block of file from master. 2. Master sends location of DataNodes for block. 3. Client accesses and rewrites all relevant copies of write block. 4. Repeat the process for all relevant blocks. 5. Writing of file complete.

-**Sequential Read File**: 1. Client requests blocks from master, 2. Master sends relevant names of nodes and locations of blocks to client. 3. Client reads blocks from any relevant node.

→ single master will eventually become a bottleneck.

Gluster (GNU+cluster) FS: is a peer-to-peer approach for distributed file systems. Client contacts servers which store data and metadata.

-**brick**: basic unit of storage. Exports disk storage located on a server.

-**volume**: is a collection of bricks. Servers in a volume are part of a *Trusted Storage Pool* (TSP), i.e. not any node can join. Volumes are distributed. A server can host parts of multiple volumes.

-**types of volumes**: distributed, replicated, distributed replicated, dispersed, distributed dispersed

-**metadata**: is distributed among all nodes in form of distributed *hash tables* (DHT). Each node has a DHT directory, which assigns hash ranges to all storage nodes.

-**translator**: building block for communication between client and storage. Requests of clients first run through stack of translators. They do caching, distributing of request, modification (e.g. encryption), access control (can block requests)

-**distributed volumes**: distributes files across bricks of a volume. Does not replicate data, i.e. no protection against data loss → highly available, but not reliable.
-**replicated volumes**: files are identically replicated across bricks in a volume. Number of replicas can change for different volumes. File updates are synchronized across replicas. Should use bricks from different servers for the replicas. → high availability and reliable, but no scaling.
-**distributed replicated volumes**: combines distribution and replication. Distributes files across servers and replicates them within a replicated volume. → high availability, reliable, and scales.
-**dispersed volumes**: use erasure codes (i.e. add redundancies for reconstruction) to replicate files. Each brick stores parts of a file (i.e. data and parity blocks), bricks store same files. Similar to a replicated volume, but with lower storage costs.
-**distributed dispersed volumes**: additionally to erasure codes, distribute files. But each brick in this volume contains parts of different files.

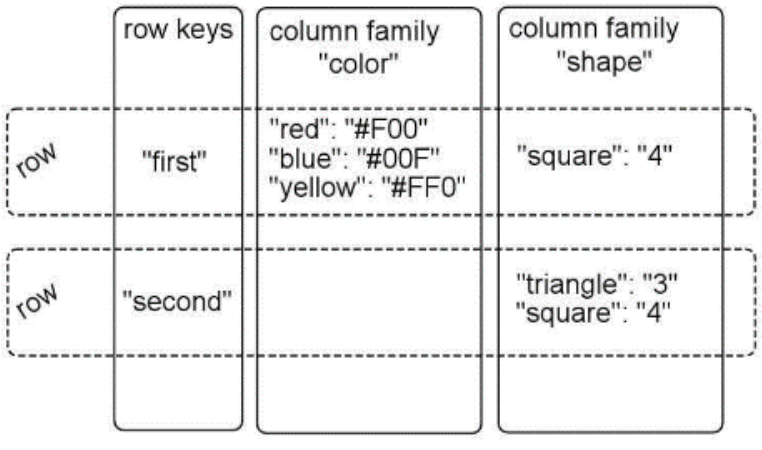
Chapter 7: Concrete Systems: NoSQL Databases

Relational systems (RDBMS) are not well-suited for scale-out and semi-structured data (JSON, XML).

NoSQL (Not only SQL): systems add some functionality to flexible file systems. Good at scale-out and usually handles semi-structured data easily. Better name would be **non-relational SQL**.
-roughly 4 different types of NoSQL systems: *key-value, columnar, document, graph*.
-**CRUD functionality**: offers operations for **C**reate, **R**ead, **U**pdate, **D**elete.

--**Key-Value Stores**: stores huge amounts of data as key-value pairs (similarly to a hash table). Dynamo was popular implementation used by Amazon. Some KV stores allow complex value types, and means to iterate through the keys. Usually offer CRUD functionality. Great for performance, but bad for complex queries.
-**Redis (Remote Dictionary Server)**: use *SET* for assign values to keys (e.g. *set uzh www.uzh.ch*). Use *GET* to read a value (e.g. *get uzh*). Can do this with multiple values, delete kv-pairs, and update values.
-also supports more **complex datatypes**: Hashes (= nested redis objects), organize data in Lists/Sets, add expiration times to keys.
-keeps data in main memory (RAM), but can save data to disk (if asked to)
-can take snapshots after n changed keys, or after s seconds passed.
-default runs on one machine but is compatible with master-slave replication: assumes by default it is a master. Can configure to connect to another Redis server as a slave. But KV-stores can also be distributed in different ways (e.g. Dynamo uses Chord-like architecture)
→ KV stores are very fast and scalable, but only simply structured data, no indexes and data scanning, and no consistency.

--**Columnar Store**: also called wide-column or column-family store. First well-known system was "Big Table".
-looks like relational database (e.g. HBase stores data in tables, which contain cells (i.e. intersections of rows and columns), but no table schema is enforced. It is actually a map of maps. In a table, *keys* are arbitrary strings, that map to a row of data, each *row* is also a map, whose keys are called columns that are mapped to values. A *value* is an uninterpreted array of bytes. Columns are grouped into column families.



-**column families**: store values of similar type. A row key can be associated with arbitrary number of column families (not necessarily all columns, see example). Allows independent performance tuning for each column family. Only one column family = KV store.

-**rows**: all operations are done atomically at the row level, i.e. does not matter how many rows are affected and consistent view of particular row. → not full ACID consistency.

-**CRUD functionality**: is offered. But first create a table by defining names of table and column families. Then add KV-pairs to a column family (e.g. HBase does not check what values you add as KV-pair). Read: there are commands to scan whole table, retrieve a row/individual cell, filter by content. Delete: individual cells with *delete*, complete rows with *deleteall*. Additional commands (table needs to be taken offline before) for dropping a table and altering the schema.

-**HBase**: supports Interactive shell, Java API, Thrift (binary protocol), REST and MapReduce.

→ scalability (with horizontal partitioning), can handle very large clusters of nodes, compression and versioning capabilities, row-level consistency, but not as flexible as other NoSQL systems since it needs some kind of schema.

--**Document Store**: like a KV store, but we can look into the **documents** (i.e. values) and interpret their (complex) structure. MongoDB (from huMONGOus) is well known document storage system.

-**MongoDB**: uses JSON (JavaScript Object Notation) and BSON (Binary JSON) to represent documents. Commonly used for data exchange (like e.g. XML Extensible Markup Language). It is **schema less**.

-**JSON document**: arbitrarily deeply nested fields with values (see figure)

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": 10021
    }
}
```

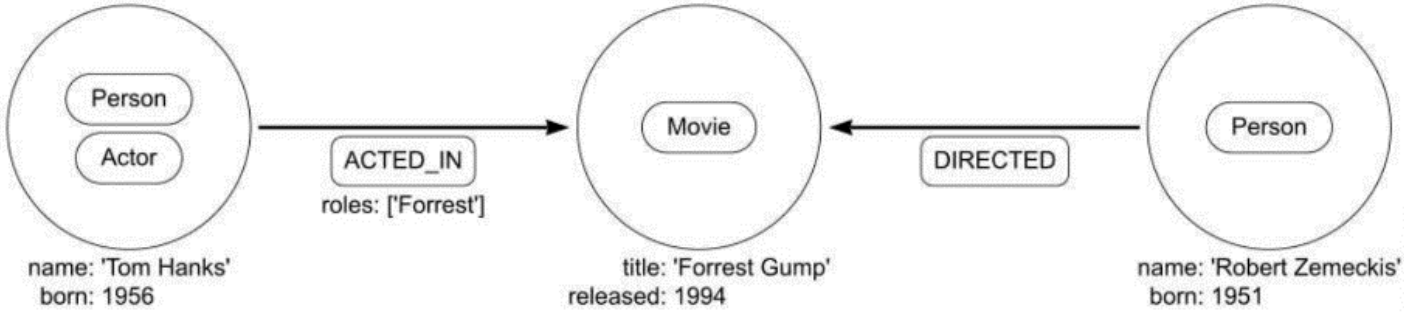
-every JSON document in MongoDB needs a special **ID field** *_id* with arbitrary, but unique (!) value. System can generate IDs automatically: 12-byte numbers by combining timestamp, machine and process ID, and 3-byte counter (e.g. 4d0ad975 bb3077 3266 f39fe3)

-**CRUD functionality of MongoDB**: provides interactive shell with JavaScript command line interface. Create database/collection happens implicitly by inserting a value. Has commands for inserting, selecting, updating and deleting (parts of) JSON documents. Retrieve documents via *find()*, provides functionality for matching partial values, lack of matching values, RegEx matching, Boolean operators. When updating documents, we can replace the whole document, or just parts of it. Deleting method uses similar syntax as *find* (i.e. documents matching the condition will be deleted).

-MongoDB also supports indexes, grouping/aggregation, MapReduce. Does not support joins, but documents can reference each other (references can be followed in queries).

→ high flexibility (any number of arbitrarily complex documents), easy to distribute and replicate, can be run on different consistency modes (e.g. master/slave replication), but may be difficult to find things.

--**Graph-based Systems**: popular system is **Neo4J** ("whiteboard-friendly"), which comes with query language Cypher. Emphasizes relationships between data items, rather than their values.



-no schema has to be defined before adding data. Represent all data as graphs (i.e. nodes, edges). A **node** can have arbitrary number of **labels** (= classify a node), and a set of ***properties*** (= key-value pair describing a node). **Edges** describe relationships between nodes, they also have **types** (=labels) and **properties**. Naming conventions: Node labels start with uppercase (e.g. Actor), relationship types use all uppercase with _ (e.g. ACTED_IN). Various properties, each with a value (see example).

-**Cypher**: standard query in Cyper contains: START … MATCH … WHERE … RETURN …

-START: selects starting node, MATCH: describes searched pattern, WHERE: adds specific query predicates, RETURN: specifies what should be returned in answer.

-example: want all actors of the movie "Forrest Gump". First line matches all edges that point to a node with label "Movie". We can access both edge endpoints.

```
MATCH (a)-[:ACTED_IN]->(m:Movie)
WHERE m.title = "Forrest Gump"
RETURN a;
```

-**CRUD functionality**: RETURN clause retrieves data. Replace RETURN with DELETE or SET, to delete or update data. Use CREATE clause for creating nodes and relationships.

-Neo4J offers other APIs: RESTful, Gremlin,… . For languages Java, Ruby, PHP,…

→ well-suited for applications involving networks, <u>but</u> difficult to distribute/replicate parts of graphs, uses completely different querying model

-**Disadvantages of NoSQL**: not as mature as RDBMSs, no standard interfaces (causes development overhead), schemaless can lead to technical debt (i.e. shortcuts/compromises in development), sacrifices data integrity for performance, lack of analytics functionality (but can be plugged into scalable architectures).

PART II: Data Processing

<u>Chapter 8: Principles: Map Reduce</u>

Processing Data: There is no standard declarative query language (like e.g. SQL). Often we have interfaces like APIs or RESTful Interfaces (Representational State Transfer via HTTP). But doing everything manually is tedious.

-**example**: build web index (like Google). First, fetch many webpage links by crawling the web. Then map it to a forward index (i.e. list of elements that it contains). For query, loop through all data (highly inefficient). Alternatively use **inverted index** (i.e. each word is associated with keys to documents which contain that word). This provides very **efficient lookups**, but index **building is expensive**.

-in practice, web indexing is a little more complicated, but builds on the idea of inverted index approach

→ *Map reduce* helps building this inverted index

Map Reduce: provides abstract framework for parallelization. Should rather be called map/shuffle/reduce.

-**Map Step**: iterate over large number of records, extract something of interest from each shuffle, and sort intermediate results. Map each input key-value pair to the set of intermediate key-value pairs.

-**Reduce Step**: Aggregate intermediate results. Group (into list) intermediate key-value pairs by duplicates of key. Then reduce each group separately into a output key-value pair.

-**parallelize**: mapping and reducing operations, but aggregating values by keys must be done all together (need to process all key-value pairs that were extracted from every map operation).

-**implementation**: programmer must only specify *map* and *reduce* functions. *map(k1,v1) -> list(k2,v2)* processes the key-value pairs, and produces a set of intermediate pairs. *reduce(k2,list(v2)) -> list(v2)* combines all intermediate values for a particular key and produces set of merged output values.

-**building inverted index**: each document is input to a map process, in which an occurring word (e.g. "statistics") is a key, and the document id (e.g. doc1) is the value. In aggregation, we take groups of

words (e.g. A-M as first letter is one group). Then reduce each group to get a key with corresponding value being a list of all documents in which the word occurs.
-**Master node for coordination**: Mappers write intermediate results to local disk and inform master when they are finished. Master tells reducers where to find output of mappers. Reducer fetches and processes its partition and stores its own output on global disk.
-**mapper fails**: Master pings mappers regularly. If they do not reply for some time, they are considered to have failed. Any in-progress/completed task is rescheduled, and reducers are notified about re-execution.
-**reducer fails**: only in-progress tasks are rescheduled, since completed tasks were stored globally.
-**master fails**: Either restart (whole) map-reduce job, or let master write regular checkpoints (in case of failure, another node can take over from last checkpoint). Happens quite rarely.
-**in case of global variables**, parallelization becomes difficult. Must synchronize access temporarily for others (example: ICA with SGD weight matrix W has to be blocked during parallelization).
→ not efficient for iterative algorithms (mappers read and reducers write at every step), disk access ist major bottleneck, for each iteration new mappers and reducers have to be initialized (much overhead for short-lived tasks)
→ difficult to program Map Reduce directly, not all problems fit this scheme, performance and scalability issues for some problems, but Map Reduce is important step in building scalable systems.

Chapter 9: Principles: Stream Processing

In many applications results need to be continuously computed and reported (e.g. monitoring production machines, real-time input from sensor networks). Continuous data stream is generated.

Stream Processing: a system needs to constantly incorporate new data and compute a result. Input data is unbounded, i.e. has no defined beginning/end. E.g. keep running count, or aggregate time windows of data.
-can be seen as **opposite to DBMS** with persistent data, transient queries. But in Streaming we have transient data, persistent queries.
-**batch processing**: involves computations over fixed input datasets, size of data is known, processing can go back to older data (since usually there is a backup). But in stream processing don't keep data.
-**Advantages of SP compared to BP**: lower latency (produces output more quickly, which is necessary in some applications), more efficient (compute results incrementally, while BP would have to go through whole dataset, even for small amount of new data).
-**example use cases**: Notifications and alerting, real-time reporting, incremental ETL, real-time decision making, online machine learning.
-**Notifications and Alerting**: constant monitoring, to be notified when certain event occurs.
-**Real-Time Reporting**: e.g. real-time dashboards to keep people informed (to examine current state)
-**Incremental ETL (Extract, Transform, Load)**: turn raw data into structured data, to make it available faster. But this has to be done more carefully (make sure each data item is processed only once)
-**Real-Time Decision Making**: analyzing new inputs and responding automatically (e.g. fraud detection)
-**Online Machine Learning**: related to real-time decision making, but instead of comparing current data to fixed patterns, also analyze streaming data.
-**Stateless Functions**: always apply the same function to input. Possibly: output size ≠ input size.
-**Stateful Functions**: keep a state from earlier events (e.g. moving average). Sometimes more complicated, like comparing different streams or only look at most recent values. Typical method for this is *windowing* of the last k elements in the stream.

-**windowing**: k most recent events are bundled into a window. All events inside window are processed. Tumbling windows do not overlap (side by side by full window length). Sliding windows slide by smaller distance, and thus overlap.

-**Chaining**: Operators/functions can be chained together. Can create arbitrarily complex workflows.

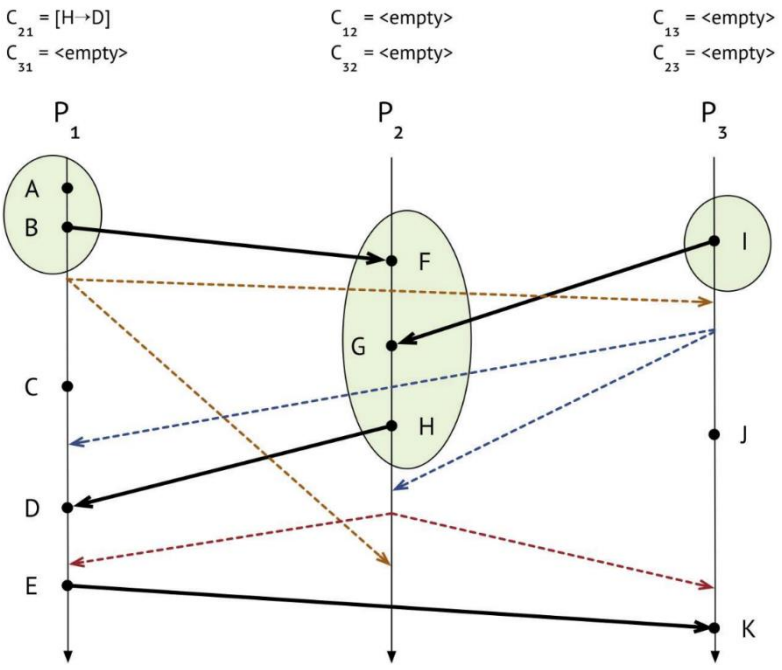
Scaling Out for Stream Processing: in case of stateless functions, simply parallelize. Stateful functions usually need some kind of synchronization

-**Fault Tolerance**: need to create snapshots/checkpoints (since we cannot restart whole stream). But creating a consistent snapshot over the whole distributed system can be very difficult.

-**Distributed Snapshots**: need to ensure "exactly-once semantics", i.e. do not lose events, or create duplicates. Simply drain pipelines and stop processing would cause considerable delay. Instead, use super-precise synchronized clocks, so all nodes can take snapshot at exactly the same time. But note that even then some events in transit might be missed.

-**Chandy-Lamport Protocol**: creates consistent distributed snapshots. Assume system is a set of processes which process events and send/receive messages, call processes P1,P2, … , send messages via channels C_ij (is channel from process Pi to Pj; and fulfills FIFO property). The snapshot is a recording of the state of each process and channel. Assembling these fragments will give a sensible result. If an event is in the snapshot, all events occurring before this event are also in the snapshot

-**EXAMPLE**: Any process can **initiate a snapshot**. Look at case for P1 initiating. P1 records its own state, sends *marker message* on all outgoing channels (C_12, C_13) (just for coordination), and keeps track of messages on incoming channels (C_21, C_31). Events A,B (marked in green), marker messages (yellow dashed lines) are sent out. 2 Cases for **arrival of message**. **Case1**: receiving process (P3) sees a marker message for the **first time**. Then P3 records its own state, label channel through which marker message came (C_13) as empty (so that any other messages from this channel are not part of snapshot), send marker messages on all its outgoing channels (C_31, C_32). Keep track of messages on incoming channels that are not empty (C_23). **Case2**: Receiving process P1 has already received a marker message (sending a marker message also counts as having seen one). Then P1 stops channel on which marker message arrived (C_31), saves recording as final state of this channel (P1 hasn't received anything on C_31, since sending having sent the marker message, so the state is empty). (**END cases**). Marker message arrives from P3 to P2. This is first marker message for P2, so P2 records its state, marks channel C_32 as empty, sends out marker message on C_21 and C_23, and starts recording on all non-empty channels i.e. C_12. Now all processes have recorded their state. Now **wrap up the channels**. Message from P1 to P2 arrives, P2 has already seen marker, stops recording C_12, and saves final state of C_12. **P2** is now completely **done** (recorded own and all incoming states). P1 receives marker message from P2, P1 has seen marker, stops recording C_21, saves final state of C_21 (needs to save received (application) message H->D). **P1** is completely **done**. P3 receives marker from P2, has already seen marker, stops recording C_23, and saves final state of C_23. **Completely done**. If an event was recorded, then also all its prior events were recorded. Have no message going backwards in time.



Watermarks for Stream Processing: are used by many systems to cope with data arriving out of order. When events arrive in perfect timestamp order, the timestep of each event can act as a watermark, i.e. no event with timestamp earlier than the watermark will arrive in the future. But in practice, events often arrive out of order.

-set a threshold (i.e. watermark) at timestamp (e.g. 11) meaning that any events with timestamp less than 11 (earlier) will be ignored at this point.

-**monotonic**: each new watermark must be greater than the previous one (consistent progression)

-less watermarks result in **larger latency, but better accuracy** (improve latency by allowing partial incremental computation)

-**Disadvantages**: difficult to implement and program (has own reasoning, and thus harder to debug)

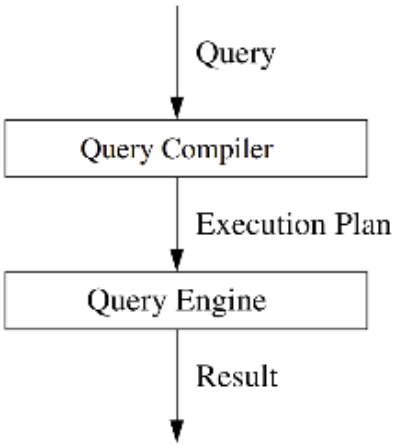
Design Decisions for Stream Processing: are not straightforward and depend on application scenario.

-**Record-at-a-time vs. High-level APIs**: Raat is simple to implement (pass each event to application), full control over processing of data, application has to maintain state. HL-API are easier to use, usually declarative (user says what to do), and failure recovery is often already built-in.

-**Event-time vs. Processing Time**: ET events are processed in order of timestamps, important in large-scale networked application (expect delay from some resources), will process late events correctly. PT events are processed in received order, easier to implement (ok for local systems), problems with events arriving out of order.

-**Continuous vs. Micro-batch Execution**: CE input streams are processed one data record at a time, state can be updated with each individual record, low latency. MBE system waits (e.g. 500ms) and accumulates data into small batches, processing of batch can be optimized (e.g. vectorized processing), results in better throughput.

Chapter 10: Principles: SQL on Big Data



SQL (Structured Query Language): is the standard for high-level declarative query languages, when interacting with structured tabular data. Good, since most datasets are structured in tabular way.

-**declarative**: even for RDBMS, query has to be translated into procedural program that can be run on the query engine. DBMS usually translate SQL into relational-algebra-like language

Canonical Translation: of SQL into relational algebra. Formally, we *select A1,...,An; from R1,...,Rn; where p*. The Ai describe some attributes, Ri describe where we take the results from (?), p is a condition that filters what we read from the Ri. Grouping/aggregation operators are quite complicated. Denote it by γ_{Ai1, Ai2, ..., Aim; θ(Aj) } (R), which means that we group tuples according to attributes Ai1, Ai2, ..., Aim, and aggregate the attribute Aj using function θ. Then γ is the translation of the following query into algebra: *select Ai1, Ai2, ..., Aim; θ(Aj) ; from R ; group by Ai1, Ai2, ..., Aim*.

-use **query optimizers** to make this process more efficient. But this is a difficult problem.

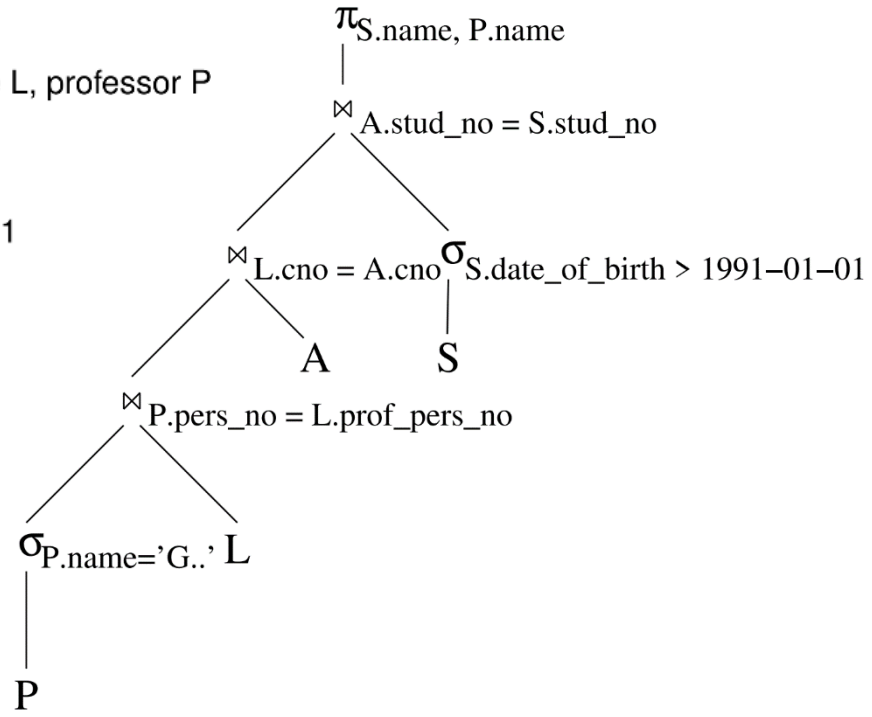
Query Optimization: DBMS can estimate execution costs, using cost models and statistics. Investigating all possible plans is too expensive. Thus use heuristics to optimize queries. We can optimize on the *logical* and on the *physical level*.

-**Logical Level**: Start with canonical relational algebra expression. Do equivalent transformations to algebraic expressions. Rule of thumb: minimize input/output of individual operators. Some basic techniques are: breaking up selections, pushing “down” selections, converting selection and cartesian

products into joins, determining join order, inserting projections, pushing "down" projections.

EXAMPLE: See figure for optimized plan for query.

```
select  S.name, P.name
from    student S, attends A, lecture L, professor P
where   S.stud_no = A.stud_no
and     A.course_no = L.course_no
and     L.prof_pers_no = P.pers_no
and     S.date_of_birth > 1991-01-01
and     P.name = 'Gamper';
```



-**Physical Level**: choose optimal implementation of an operator. Also, decide whether to use indexes (and which ones), whether to materialize intermediate results, when to sort and eliminate duplicates, and which models/statistics to use.

Distributed Database Systems: SQL was originally built for centralized architectures. RDBMS add functionality of table fragmentation, allocation of nodes, and techniques for distributed query optimization.

-but RDBMS were never built for massive scale out, thus overheads like 2PC or distributed lock management are necessary. Also, RDBMS assume reliable component (failure could block everything)

Large Scale SQL: We try to translate SQL into Map Reduce (MR) (which does scale out, compared to SQL). A complex SQL query will need several MR phases.

-**Mapping Operators**: only needs an appropriate map function for *table scan*, *selection* σ, *projection without duplicate elimination* π^d. The reduce is simply identity function. Operators *Join* ⋉, *Grouping/aggregation* γ, and *Duplicate Elimination* π require more complex Reduce function.

-**Tuple Representation and Table Scans**: Assume table contents are stored on distributed storage system. Scan Map operator reads a tuple t and converts it into key-value pair (t, t), i.e. both key and value are set to t. (t, t) is sent further into the pipeline. For each (t, t), that a reduce operator receives, it stores t in the output (i.e. basically identity function)

-**Selections and (Simple) Projections**: To translate a selection σ_P (with some predicate P), read input tuple by tuple. If t satisfies P, then output (t, t), otherwise discard it. To translate a projection $\pi^d_{\{A1,A2,\dots\}}$, read the input tuple by tuple, strip out all attributes not in A_1, A_2, ..., creating t'. Output (t', t').

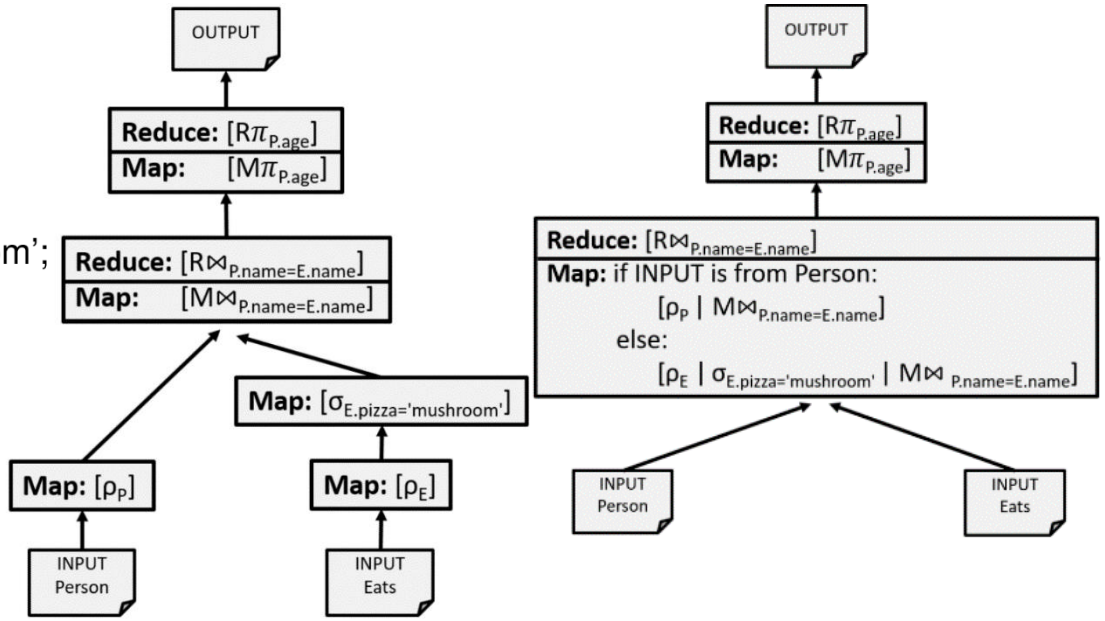
-**Joins**: To translate a (conjunctive equi-) join $R \ltimes\_\{R.Ai = S.Bj\}(S)$, the Map operator for each tuple in R, produces the key-value pair $(A_i, \langle R, A_1, A_2, \dots, A_n \rangle)$, and for each tuple in S, produce key-value pair $(B_j, \langle S, B_1, B_2, \dots, B_m \rangle)$. For the Reduce operator each key x either has a value $\langle R, \dots \rangle$, or $\langle S, \dots \rangle$, and output $A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m$ of each element in Cartesian product as a tuple.

-**Grouping and Aggregation**: To translate $\gamma_{\{A_{i_1}, A_{i_2}, \dots, A_{i_m}; \theta(A_j)\}}$, the Map operator produces a new key-value pair of the form $(\langle A_{i_1}, A_{i_2}, \dots, A_{i_m} \rangle, A_j)$. The Reduce operator: each key x represents one group. Apply function θ to the list of values $[y_1, y_2, \dots]$ associated with key x, then output $(x, \theta([y_1, y_2, \dots]))$

-**Duplicate Elimination**: To translate $\pi_{\{A1,A2,\dots\}}$, keep the Map operator same as for $\pi^d_{\{A1,A2,\dots\}}$, i.e. strip out all attributes not in A_1, A_2, ..., creating t'. Output (t', t'). The Reduce operator: for each key, the length of the list of values may vary. However, the values will all be the same t'. Output t' for every key only once.

-**Example of SQL as MP:**

```
select   distinct P.age
from     person P, eats E
where    P.name = E.name
and      E.pizza = 'mushroom';
```



-**Optimization**: try to optimize MR execution by reducing the number of phases. Use *Chain Folding*, *Broadcast Join* and *Multiway Join*.

-**Chain Folding**: merges multiple map steps into one. This helps to reduce the number of phases, i.e. that reduce writes its output on a global disk.

In **example**, the rename and selection can be done in one step, to receive second graph (on the right of the page).

-**Broadcast Joins**: improve performance in case of very unevenly sized relations. If smaller relation fits into main memory, send it o the other nodes, and to the join in the map step without shuffling the data.

-**Multiway Joins**: Assume we want to join the relations R, S, T. In two separate phases, intermediate result (e.g. R⋉S) is stored on global disk. Save space by trying to join all relations in one single phase.

-**Case1**: assume we use the same join attribute A. Then join $R(A,B), S(A,C)$, and $T(A,D)$ on attribute A. Instead of tagging tuples with R or S, we tag them with R, S or T. Then translating $R \ltimes_{\{R.A=S.A\}} S \ltimes_{\{S.A=T.A\}} T$ means the Map operator for each tuple in X produces the key-value $(A, \langle X, A, \dots \rangle)$ with $X = R$ or S or T. The Reduce operator: Each key x either has a value $\langle R, \dots \rangle, \langle S, \dots \rangle$ or $\langle T, \dots \rangle$. Then construct cartesian product of all $\langle R, \dots \rangle, \langle S, \dots \rangle$, and $\langle T, \dots \rangle$ values.

-**Case2**: assume we use different join attributes, i.e. join $R(A,B)$ and $S(B,C)$ on attribute B, and then join $S(B,C)$ and $T(C,D)$ on attribute C. Now assume we have k reducers, and two hash functions h and g. h maps B-values into b buckets $\beta_1, \dots, \beta_b$, and g maps C-values into c buckets $\gamma_1, \dots, \gamma_c$. We require that $b * c = k$. Each reducer is responsible for a pair (β_i, γ_j) of buckets. Each reducer has to look at tuples $(u,v) \in R$, $(v,w) \in S$, and $(w,x) \in T$ if $h(v) = i$ and $g(w) = j$. Mappers reads and prepares tuples, then sends them to a reducer. For a tuple $(v,w) \in S$, the reducer for pair $(\beta_{h(v)}, \gamma_{g(w)})$ is responsible. But for tuples from R and T, this is more complicated, since they have to be sent to multiple reducers. For $(u,v) \in R$ we only know $h(v)$. It needs to go to the reducers for the pairs $(\beta_{h(v)}, y)$ for all possible values of y. For a tuple $(w,x) \in T$, we only know $g(w)$. It needs to go to the reducers for the pairs $(z, \gamma_{g(w)})$ for all possible values of z.

Chapter 11: Concrete Systems: Spark

Apache Spark: is a cluster-based high-performance system for big data processing. Initially developed at UC Berkeley. It is used by many major companies.

-**In-memory cluster computing framework**: allows writing to disk between operations. In-memory caching for reading data multiple times

-**Advanced Execution Engine**: Allows optimization for complex sequences of execution steps. Reduces overhead by re-using application containers.

-**more features**: runs on commodity hardware, built-in fault tolerance

-**Comparison to Hadoop MR**: Apache is considered the successor of Hadoop Map Reduce (MR). It's easier to use, while also more general, spark can use more than 100 operators (Hadoop MR only 2), allows complex processing pipelines of arbitrary length and many different operators (Hadoop MR only supports sequences of Mad and Reduce), Spark supports advanced distributed data structures (e.g. DataFrames designed to support data science)

-**Spark Core**: implements execution engine and framework. It contains the basic functionality of Spark, i.e. execution plan creation, task scheduling, memory management, in-memory computing (via Resilient Distributed Datasets (RDDs)), fault-handling and scalability. There are APIs for Scala, Java, Python, R and SQL.

-**Spark SQL**: can run SQL queries on distributed data. Can also interact with relational databases (via SQL or HQL (Hive Query Language; SQL-like layer on top of HDFS) queries)

-**Spark Streaming**: responsible for executing operations on data streams. Code for batch-mode only needs minor modifications to be applied to streams.

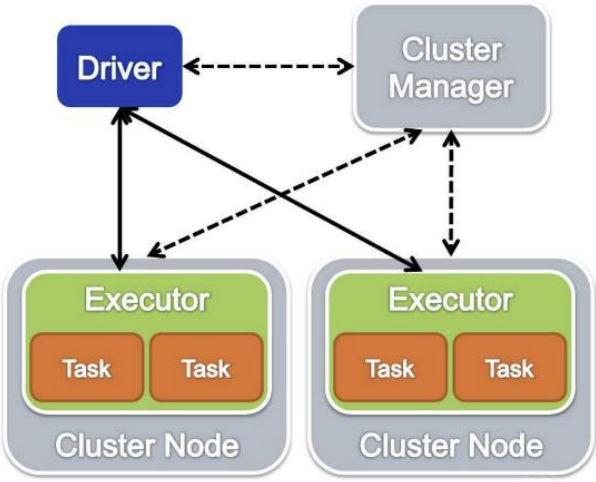
-**MLib / GraphX**: provide functionality for machine learning (i.e. classification, regression, clustering,...). All algorithms are built for parallel processing. Also provides library for graph processing.

-**Cluster Management**: use YARN (Hadoop), Apache Mesos, simple Spark cluster manager

-**How to run applications**: Spark applications consist of a **driver process**, which runs the *main()* function, and is responsible for maintaining information about application, responding to a user's program or input, and distributing and scheduling work (compared to executors). And a **set of executor processes**, that do the actual work, by being responsible for executing the code assigned to them, and reporting their status back to the driver.

-**SparkContext**: is the entry point to Spark Core. It sets up internal services and establishes a connection to a Spark execution environment. When a SparkContext is created, we can access Spark services and run jobs.

-**architecture**: *Driver* handles user code and SparkContext, and runs the node in the cluster. *Executor* is the JVM process. *Task* (execution thread) is the parallel work unit. *Cluster Manager* can be one of Spark Standalone, Mesos, or YARN. (see figure)



Data Abstractions: Spark uses different core data abstractions also called *distributed collections*, such as DataFrames/Datasets or Resilient Distributed Datasets (RDD).

-**Partitions**: Spark breaks data into partitions and executors perform work in parallel. Spark hides complexity by not allowing to manipulate partitions manually. Only specify high-level transformations.

-**DataFrame**: organizes data into (named) columns. Conceptually like a table in relational database system, but the table can span across many servers. High-level interface to interact with data, APIs for many languages are supported by Spark (Scala, Java, Python, R and SQL).

-**Dataset**: very similar to DataFrame, but its API additionally provides type safety and object oriented aspects. APIs only available for Scala and Java (since Python and R are dynamically typed)

-**Resilient Distributed Dataset (RDD)**: offers a lower level API (compared to DataFrame/Dataset) and reveals some physical execution characteristics, but still hides a lot of underlying complexity. User can store arbitrary Java or Python objects in every record of an RDD. More control over these records, but now also need to write code for data manipulation. Must do optimizations by hand.
-RDDs used to be primary API in Spark. DataFrames and Datasets are always compiled down to RDDs. Spark knows their structure, so it can **apply optimizations**.
-**Convert** data abstraction to a lower **level** is easy: Dataset → DataFrame → RDD. Level up more difficult.
-**what to use**: Advised to use DataFrames/Datasets whenever possible (easier to use, spark can apply optimizations). RDDs can be necessary, if desired functionality is not available (e.g. control of physical data placement in cluster or manipulating unstructured data). Understanding them can be helpful.
-Spark core data structure cannot be changed once it is created → data cannot be changed anymore.

Transformations: in Spark describe how we want to modify a data structure. But applying a transformation returns a new data structure and does *not* modify the existing one.

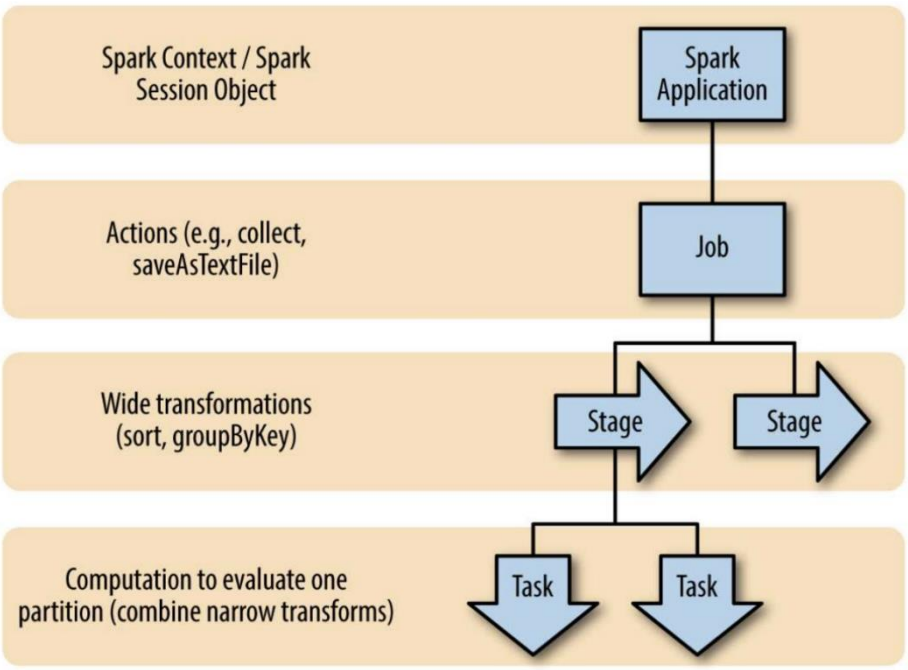
-**example**: the variable *myRange* contains a collection of numbers, then `divBy2 = myRange.where("id % 2 = 0")` will produce no output, until we call an action.
-we can apply any sequence of transformations e.g. `divBy6 = divBy2.where("id % 3 = 0")`
-**Lineage Graph**: tracks performed operations and their dependencies. Spark waits until last moment to execute a computation (i.e. lazy evaluation). This way it can build an efficient execution plan, and only computes what is strictly needed. This also simplifies the restart after losing a data partition (since original data is not changed, and every partition contains information to recalculate a partition)
-**Narrow Transformations**: each input partition contributes only to one output partition. Can be processed very efficiently (multiple filters can be performed in-memory via pipelining). The *WHERE* filter is used as a narrow transformation.
-**Wide Transformation**: each input partition contributes to many output partitions. Also called Shuffle. Partitions are exchanged across the cluster, which involves writing data to disk → less efficient

Actions: are operations in Spark which return something other than an RDD, DataFrame or Dataset.

-**trigger** actual **computation** and evaluation of transformations
-**simple example**: `divBy6.count()` determines number of data records in structure.
-**3 types of actions**: 1. Bring data to the driver (e.g. viewing data in console), 2. Collect data to naïve objects in resp. programming language, 3. To write output data sources.

Executing Operations: with every action, the Spark scheduler builds an **execution graph** and launches a **Spark job**.

-Each **Job** consists of **Stages**, and each Stage consists of **Tasks**. Spark scheduler builds a Directed Acyclic Graph (**DAG**) for the stages of each job (call it **DAG scheduler**).
-The wide transformations determine the **boundaries of each stage**, such that each stage can be computed without moving data across partitions



and no communication between executors or with the driver is necessary.

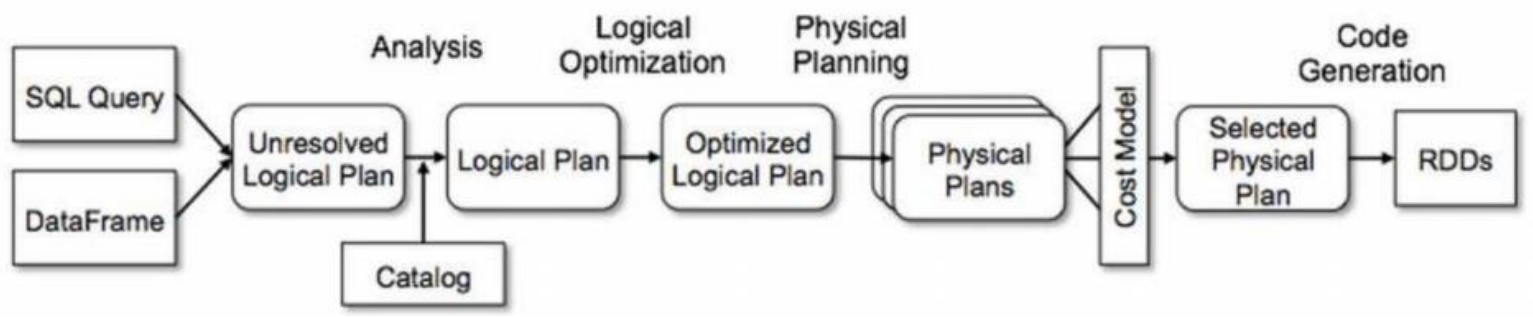
-**Execution** of a job is handled by the DAG, which determines that locations of where to run each task. --

-This information is then passed to the **Task Scheduler**, which is responsible for **running the tasks**.

-A **task** is the **smallest unit** representing one local computation, such that it cannot be run on more than one executor.

Query Optimization: DAG scheduling happens on RDD level. In the case of DataFrames/Datasets, Spark uses their metadata and context to build better execution plans.

-**Catalyst**: is Spark's optimizer. Builds a logical plan and applies several optimizations to it. Then builds a physical plan from an optimized logical plan.



Spark UI: monitors the progress of a job. It is available on port 4040 of driver node (local mode: `http://localhost:4040`). Displays current job state, environment, cluster state. Useful for debugging and tuning.

Spark SQL: There used to be as SQL API for RDDs (SchemaRDD), which became obsolete when DataFrames were introduced. Some transformations are easier to formulate in SQL, than the DataFrame API.

-use *sql()* methods to run queries on DataFrames. Generally, methods from SQL/DataFrame can be mixed, since they both operate on DataFrames.

-for using Spark SQL, we need to create an **SQL Table**. These are logically equivalent to DataFrames, but are defined in a database (while DF are defined in programming language scope)

-**SQL Table**: can be created from different sources, i.e. Hive metastores, access data stored in Hadoop environment, or create tables on the fly (e.g. loading from a file, or taking result form a query)

-**managed tables**: Spark manages their metadata and data. Dropping the table, deletes it completely (data is gone)

-**unmanaged tables**: Spark only manages the metadata, but data is stored on another system (e.g. some file system, or an external database). Dropping the table, Spark just looses access.

Streaming in Spark: used to work with DStreams (discretized streams), which does not work on DataFrames/Datasets. It does micro-batch processing, and only supports processing time.

-**Structured Streaming**: (new) high-level API, which runs on DataFrames/Datasets and supports event-time processing. It does micro-batch processing.

-uses DataFrame/Dataset and SQL APIs, which makes it easier to write applications for them.

-re-uses a lot of infrastructure, such as Catalyst (i.e. the Spark query optimizer)

-treats a stream like a table, to which data is appended. Checks for new input data, then outputs an internal state (if necessary), and writes out the result.

-supports almost all batch-mode transformations and actions.

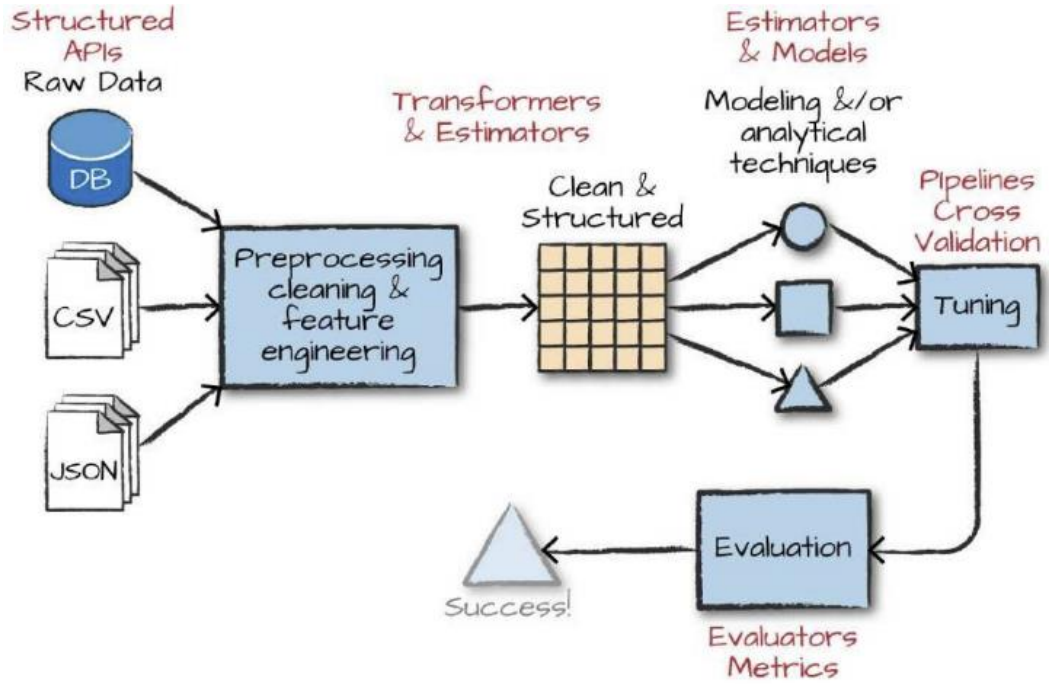
-**Input Sources**: Apache Kafka (dedicated streaming platform), Files in distributed System (like HDFS), Sockets (for testing purposes)

-**Sinks (outputs)**: Apache Kafka, almost any file format, Console/memory (for testing/debugging)

-**Output modes (how to output data)**: append (only add new record), update (change record in place), complete (rewrite full output)
-**Triggers (when to output data)**: Default is to output data is soon as mini-batch is processed, triggers based on processing time are also supported.
-**Watermarks**: define how long data should be kept (arriving after watermark will not be considered)

Machine Learning in Spark: is about how to do **scalable** Machine Learning for the case of **supervised learning** (find parameters of a function, which best fit data for which the input and desired output is provided).

-**ML (MLlib)**: is the library that supports the Machine Learning **workflow** (see figure) for many ML algorithms. This allows for scalability to huge datasets. MLlib is older version for RDDs, ML is new version for DataFrames. (similarly to scikit learn)



-**fundamental types/functions**: DataFrames, *Transformers, Estimators, Evaluators, Pipelines*.

-**Transformers**: are functions for converting data, used primarily for preprocessing and feature engineering (e.g. Normalizing/Standardizing, Casting (i.e. changing) data types, converting categorical variables to numerical ones. Take DataFrame as input, and output a new DataFrame with added column(s).

-**example**: **`Binarizer`**, which converts continuous values into binary (0 or 1), via some threshold.

For `DF` a dataframe, and a column `A`, write

```
from pyspark.ml.feature import Binarizer
binarizer = Binarizer(threshold=0.5, inputCol="A",outputCol="bin_A")
binDF = binarizer.transform(DF)
```

-**Estimators**: are algorithms, which take DataFrames as inputs, and create a transformer based on the content of the DataFrame. They can be used for complex preprocessing (e.g. standardizing, which needs two passes through the data), or building and training models (e.g. for regression).

-**example**: **`StandardScaler`**, which standardizes a set of values to have mean 0 and std 1.

```
from pyspark.ml.feature import StandardScaler
scaler = StandardScaler(withMean=True, withStd=True, inputCol="V",
                        outputCol="scaled_V")
scalerModel = scaler.fit(DF)
scaledData = scalerModel.transform(DF)
```

-**Example: Logistic Regression**. Use estimators to build a regression model. In the end, the transformer add the column `prediction` to the `testingDF`

```
from pyspark.ml.classification import LogisticRegression
trainingDF = ...
lr = LogisticRegression(maxIter=10, regParam=0.3)
```

```
lrModel = lr.fit(trainingDF)
testingDF = ...
prediction = lrModel.transform(testingDF)
```

-**Evaluators**: compares the output of a model to the ground truth, given some evaluation metric. We have evaluators for classification: precision, recall, F-measure, receiver operating characteristic (ROC) and for regression: mean absolute error, and root mean squared error

-**example**: use `BinaryClassificationEvaluator`

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator
evaluator = BinaryClassificationEvaluator(metricName="areaUnderPR",
rawPredictionCol="prediction", labelCol="label")

evaluator.evaluate(lrModel.transform(testingDF))
```

-**Vector**: is a low-level datatype in Spark. It is usually used for passing features into a ML algorithm. **Dense** vectors are an array of all values, while **sparse** vectors only specify the non-zero elements.

-**example**: define same vector as dense and as sparse vector:

```
from pyspark.ml.linalg import Vectors
denseVec = Vectors.dense(1.0, 0.0, 3.0, 0.0, 0.0, 2.0, 0.0, 0.0)
sparseVec = Vectors.sparse(8, [0, 2, 5], [1.0, 3.0, 2.0])
```

sparse input: `(vector_size, non-zero_indices_list, non-zero_values_list)`

-**High-Level Transformer**: operate a higher level of abstraction, i.e. more complex transformations in one step, which is usually implemented in a declarative way (hiding complexity). Some examples are: VectorAssembler, RFormula (to use R), SQL Transformers (to use SQL), StringIndexer (turn categorical data into numerical values), Text Data Transformers (tokenizers,stop word removal,computing TF_IDF,..)

-**Pipelining**: is useful when tuning many parameters, since applying all workflow steps (see figure earlier) manually is tedious. Spark offers pipelines to facilitate this in a scalable and reliable way. Each workflow step becomes a *stage* in a pipeline: e.g. specify stages for transformation and tuning

```
from pyspark.ml import Pipeline
rForm = RFormula()
lr = LogisticRegression(labelCol="label", featuresCol="features")
pipeline = Pipeline(stages=[rForm, lr])
```

-**Grid/Exhaustive Search**: is also supported by Spark, e.g. code

```
from pyspark.ml.tuning import ParamGridBuilder
params = ParamGridBuilder()\
    .addGrid(rForm.formula, [“y~x1+x2” , “y~x1+x3”])\
    .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0])\
    .addGrid(lr.regParam, [0.1, 2.0])\
    .build()
```

-validation nets are necessary to prevent overfitting when doing hyperparameter tuning. Typically do:

-**Cross-Validation**: uses validation sets during training / hyperparameter tuning. Partition data into k equally sized blocks, then use them to train k distinct models. Specifically, model I is trained on k-1 blocks (all except block i), and then validated on block i. Particularly useful for small datasets.

-**in Spark**: here, the `evaluator` could be e.g. the one defined above

```
from pyspark.ml.tuning import TrainValidationSplit
tvs = TrainValidationSplit()\
      .setTrainRatio(0.75)\
      .setEstimatorParamMaps(params)\
      .setEstimator(pipeline)\
      .setEvaluator(evaluator)
```

-**Find best model**: all parts have previously been defined. Now start the search for the best model. From `tvsFitted` we can extract the parameters for the best model

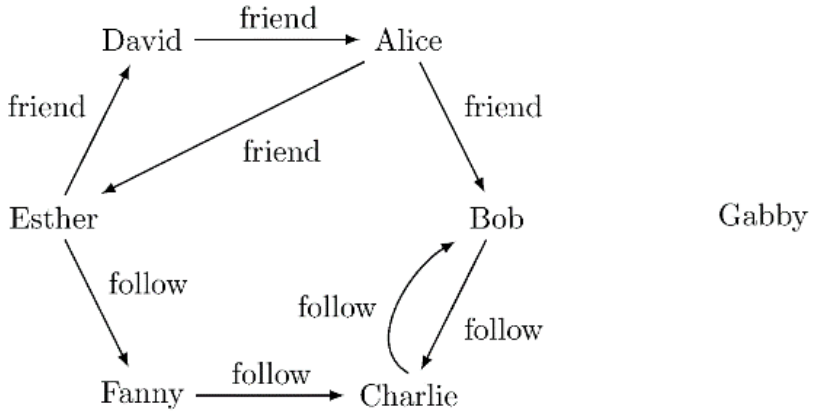
```
train, test = DF.randomSplit([0.7, 0.3])
tvsFitted = tvs.fit(train)
evaluator.evaluate(tvsFitted.transform(test))
```

-**Deploying a Model**: store model on the disk: `tvsFitted.save("/tmp/modelLocation")`
→ we can train large scale models, but Spark is **not ideal for deploying** the fitted models. This is because Spark is not optimized for latency, and starting jobs is not cheap.

GraphX / GraphFrames: are libraries for computing graphs. GX is an older version, based on RDDs. It provides a low level interface, that is not easy to use and optimize. GF is based on DataFrames and provides a more comfortable higher-level API, but is not yet fully integrated into Spark. GF is built for scalable graph processing (which Neo4J struggles with), not focused on graph storage or query languages.

-**GraphFrames (GF)**: are DataFrames for storing graphs, given by a set of vertices and edges. GF API supports several graph queries: get degree of nodes, extract subgraph, standard graph algorithms (BFS, connected components, shortest paths, etc.)

--**Example**: (see figure). Load **vertices** and **edges** into dataframe




```
vertices =
sqlContext.createDataFrame([
("a", "Alice", 34), ("b",
"Bob", 36), ("c", "Charlie",
30), ("d", "David", 29), ("e",
"Esther", 32), ("f", "Fanny",
36), ("g", "Gabby", 60)],
["id", "name", "age"])
edges =
sqlContext.createDataFrame([
("a", "b", "friend"), ("b",
"c", "follow"), ("c", "b", "follow"), ("f", "c", "follow"), ("e", "f",
"follow"), ("e", "d", "friend"), ("d", "a", "friend"), ("a", "e",
"friend")], ["src", "dst", "relationship"])
```

Load then into a **GraphFrame**: `g = GraphFrame(vertices, edges)`

-**node degrees**: we can get degrees of nodes with `g.degrees` or `g.inDegrees` or `g.outDegrees`.

-**filter methods**: `g.edges.filter("relationship = 'follow'").count()`
OR `g.vertices.filter("age = 29")`

-**subgraphs**: by combining filters we can get a subgraph (e.g. keep friend edges, and nodes A, B, E)

```
g2 = g.filterEdges("relationship = 'friend'").\
     filterVertices("age > 30").\
     dropIsolatedVertices()
```

-**Motifs**: can be used to describe complex structures in a similar way to the MATCH-clause in Cypher: e.g. describe nodes connected by edges in both directions:
`g.find("(a)-[e]->(b); (b)-[e2]->(a)")` (outputs a DataFrame for further processing)

-**BFS (Breath-first search)**: finds shortest path from one node to another. E.g. start from node *Esther* and stop as soon as you find a node with age<32: `g.bfs("name = 'Esther'", "age < 32")`

-or specify edge filters and restrict the path length:

```
g.bfs("name = 'Esther'", "age < 32",\
      edgeFilter="relationship != 'friend'",\ maxPathLength=3)
```

-**compute connected components**: is where Spark reaches its limits, since this is one of the most expensive GF algorithms. Use a checkpoint directory to save the job after every iteration (to continue from there in case of a crash)

```
sc.setCheckpointDir("/tmp/checkpoint")
scc = g.stronglyConnectedComponents(maxIter=10)
```

Spark in a nutshell: is a distributed platform for efficient processing of large datasets, whose functionality goes well beyond MapReduce. It's also very API user friendly. Generally much involvement in improving it.

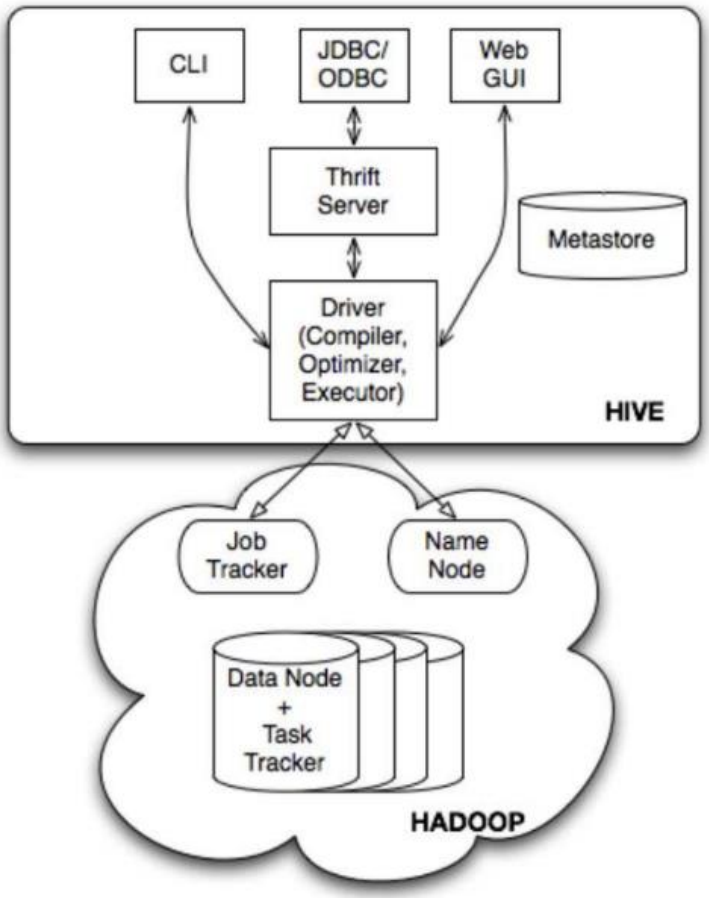
Chapter 12: Concrete Systems: Hive

Hive: is an open-source append-only system built on top of Hadoop. Supports Queries written in *HiveQL* (SQL-like declarative language). Queries are then translated into map-reduce jobs. Map-reduce scripts can be put into HiveQL queries. First step in building scalable (op-src) data warehouse.

-**Data Model**: data is organized in tables which play same role as in RDBMS. Each table has a HDFS directory, where content is serialized and stored. A table is divided into partitions, which determine mapping to subdirectories in the table directory. Each partition can be divided into *buckets*. Each bucket is then stored in one file.

-**Data Types**: usual primitive data types (integers, floats, strings, dates,...) and collection types (arrays, maps,...). Can also have user-defined types.

-**Metastore**: stores all metadata (system catalog). For a database, this is the namespace for the tables belonging together. For a table, this is a list of columns which their datatypes (also includes location of table/its partition/its buckets in file system, and specifies used serialization). Interacts with all other components. It uses an RDBMS or regular file system (since HDFS is optimized for sequential scans).



-**Interface Level**: top level provides external interfaces: command line (CLI), web user interface, API for JDBC/ODBC programming languages. JDBC/ODBC does not directly connect to driver, but uses the thrift server for crass-language services (transforms requests written in Java JDBC, C/C++ ODBC, ...)

-**Driver**: manages life cycle of query statement (i.e. compilation, optimization and execution).

-**Compiler**: translates query statement into DAG map-reduce jobs, which is then sent to execution engine (Hadoop). Specifically, the *parser* translates query string into a parse tree. The *semantic analyzer* turns this into an internal query representation. This is then converted into a *logical plan*, which is rewritten an optimized. Finally, a *physical plan* is generated and executed.

-**HiveQL**: is the query language of Hive and supports standard SPJ-queries (i.e. select, project, join). Data Definition Language (DDL) is used to create tables (specifying serialization format, partitions, and buckets). Data Manipulation Language (DML) only allows load and insert statements (can use multi-table inserts, i.e. with same input load different tables in one statement)

-**HiveQL vs SQL**: major differences are 1.**Updates**: SQL supports insert/delete/update while HiveQL only supports insert (multi table version). 2.**Subqueries**: Can be used in any SQL clause,but in HiveQL only in FROM,WHERE and HAVING.3.**Transactions**:Fully supported in SQL, only limited support in HQL.

Part III: Resource Management

Chapter 13: Principles: Scheduling and Resource Allocation

In dynamic (i.e. jobs starting, while others are finishing) and large cluster systems it is important to utilize resources in an efficient way. But this can be complicated to solve. (large companies: 20-30% CPU utilization)

Resources: can be CPUs, memory, disk (bandwidth), network (bandwidth), specialized resources (e.g. GPUs).
Challenges: usually different frameworks run on the same cluster (e.g. Hadoop + Spark), so clusters have to deal with **diverse workloads** (like Interactive OLAP applications with fast responses, or Large Batch jobs with high throughput).

-even in same class, **usage patterns** can **change** while a job is running. E.g. OLAP: analyst sends query, looks at results, thinks, and then sends next query,BatchJob: can change from CPU-bound to I/O-bound
-constraints with **dependencies** (Reduce tasks depend on Map tasks; different stages in Spark DAG)
-different tasks require different resources, e.g. Map Reduce: Map bound by CPU (data available locally), Shuffle is bound by network (much data transfer), Reduce is more balanced

First In First Out (FIFO) Policy: is very simple. Jobs are scheduled in the order they arrive. Waiting jobs are held in a queue. As soon as a resource becomes available, the oldest job in the queue is executed.

-every job **eventually completes** (no "starvation")
-**throughput** can **suffer** i.e. small jobs in between large jobs wait
-low response time. No prioritization means even smaller jobs that could be squeezed in have to wait.

```
while True do
    if resource demands of next job can be met then
        allocate resources to job
        start job
    else
        wait until more resources are available
```

Max-Min Fairness Policy: Assume we have fixed amount of resource C, which is shared among n jobs. Each job i has demand d_i. Allocate share a_i of C to each job i. Then clearly $\sum_i a_i \leq C$ has to be satisfied. Consider the case when $\sum_i d_i > C$ (otherwise trivial).

-assume all jobs have **equal right** to C. Allocate a_i in order of increasing d_i (never assign share to a job that its demand, i.e. $a_i \leq d_i$). Unsatisfied demands get an equal share of what is left.
-**Water Filling**: Starting from 0, assign capacity to every job at same pace until one demand is fully satisfied or we run out of resource. If resource is still available, continue with remaining demands.
-**Procedure**: to get allocated weights. Divide resource into equal shares. Overfulfilled demands allocate their demand, and hand back the rest. Divide this rest among the remaining unfulfilled demands. Stop when we have no more overfulfilled demands.
→ has been shown to be **optimal**, under the assumption that all participants have same access rights. Otherwise, optimality does **not** hold.
-**Disadvantages**: only works for single resource type. But in practice we need to allocate several types, for which Max-Min can lead to very suboptimal solutions.

Dominant Resource Fairness (DRF): attempts to equalize the shares of different users of several different resources.

-**Fairness**: is defined via **4 properties**. 1. **Envy-freeness**: user does not prefer share of another user.
2. **Sharing incentive**: user is better of by sharing than getting a fixed share (i.e. C/n for n users).
3. **Pareto-efficiency**: not possible to increase share of one user without decreasing share of another,
4. **Strategy-proofness**: lying about demands does not pay off.
-for a single resource type, max-min fairness fulfills the above description. But for several resource types, this can get quite complicated.
-**dominant share**: is the maximum allocated share of a user, among all its other shares (for different resources) (i.e. user is allocated 3/8 in CPU and 1/4 GB, then 3/8 is its max share).
-**dominant resource**: is the resource corresponding to the dominant share (above: CPU).
-**Idea**: apply max-min fairness across users' dominant shares. Roughly, DRF maximizes the smallest dominant share, then the second smallest, and so on.
-**Formal Description**: For each resource r we get a separate constraint of the form $\sum_i a_i * d_{i,r} \leq C_r$ (with a_i the amount tasks that one user i can run, and d_i the demand of one task for that resource), that should be fulfilled. Every feasible allocation (i.e. implicitly given by a combination of a_i) thus can be visualized by a region bounded by linear hyperplanes. To then choose the best allocation within that region, choose the point where the feasibility border is intersected by the line formed equalizing the dominant shares (i.e. which allocates same share of most important resource across all users).

-**But** note that this assumes tasks run by one user all look the same (i.e. can use one demand vector), and a user can execute fractions of a task.

-**Discrete DRF Algorithm**: allows tasks with different demand vectors, but only satisfies pareto efficiency. It tracks resource capacity $C = \langle c_1, \ldots, c_m \rangle$, total allocated resources to each user $U_i = \langle u_{i,1}, \ldots, u_{i,m} \rangle$ (initially 0), total sum of allocated resources $A = \langle a_1, \ldots, a_m \rangle = \sum_i U_i$, and the dominant shares of each user s_i (initially 0).

pick user i with lowest dominant share s_i
$D_i \leftarrow$ demand of user i's next task
if $A + D_i \leq C$ **then**
  $A = A + D_i$ // update total allocation vector
  $U_i = U_i + D_i$ // update user's allocation vector
  $s_i = \max_{j=1}^{m} \{\frac{u_{i,j}}{c_j}\}$ // update user's dominant share
else
  return // cluster is full

Fairness: Fairness is often defined in term of **utility**. In case of 1 resource, let $U_i(a_i)$ be the utility of allocation a_i to user i. Then U_i should be concave and non-decreasing. This is e.g. fulfilled by the log-function (captures the law of diminishing returns). Essentially, fairness becomes an optimization problem.

-in **max-min fairness**, we first maximized the utility of least demanding user, then of the second, ...

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} U_i(a_i) \\ \text{subject to} \quad & \sum_{i=1}^{n} a_i \leq C \end{aligned}$$

α-Fair Functions: describes a family of functions given by $U_\alpha(a)$.
Where for $\alpha = 0$: **maximum efficiency**, $\alpha = 1$: **proportional fairness**
If $\alpha \to \infty$: max-min fairness.

$$U_\alpha(a) = \begin{cases} \frac{a^{1-\alpha}}{1-\alpha} & \text{for } \alpha \geq 0, \alpha \neq 1 \\ \log a & \text{for } \alpha = 1 \end{cases}$$

-can be **generalized to multiple resources**.
-allows to "dial" **level of fairness** between maximum efficiency, and max-min fairness
-feasible regions are convex sets, since they are defined by hyperplanes. This means we can use well-known **convex optimization** algorithms (e.g. gradient ascent, interior point method, ADMM,...)

Chapter 14: Principles: Virtualization

Describes the use of virtual machines, which is a fairly old idea.

Goals: Use Virtualization to achieve several goals, i.e. 1. **Mimic behavior of a system**: provides a platform for running legacy code. 2. **Better resource utilization**: allow users to share resources. 3. **Flexibility**: makes it easier to deploy/migrate software. 4. **Security**: isolates applications and their environment form the rest of the system.

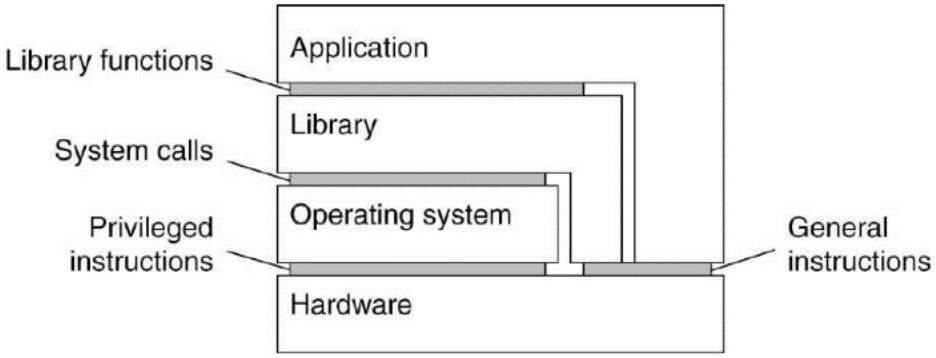
-**Typical Uses**: Data Center consolidation (improve resource efficiency and availability), Running multiple operating systems (OS) on one host, Prototyping new features (e.g. for OS: kernel debugging/testing, for hardware: new features of architectures).

Interfaces and Levels of Systems: Computer systems provide different interfaces on different levels.

-Hardware: is the instruction set architecture (ISA) is a set of machine instructions, which is divided into privileged instructions, which can only be executed by OS, and general instructions, which can be executed by any program.

-**Operating System (OS)**: system calls offered by an OS.

-**Library functions**: an API hiding lower levels.



Virtual Machines: the virtualizations of a computer system (i.e. CPU) can be achieved in two ways:

-**Emulation**: Software that completely simulates hardware (e.g. CPU registers, memory mgmt,...). The hardware is represented in data structures, where machine language instructions update the state.

-**runs anywhere** (no host OS support needed), but **very slow** (except for old hardware)

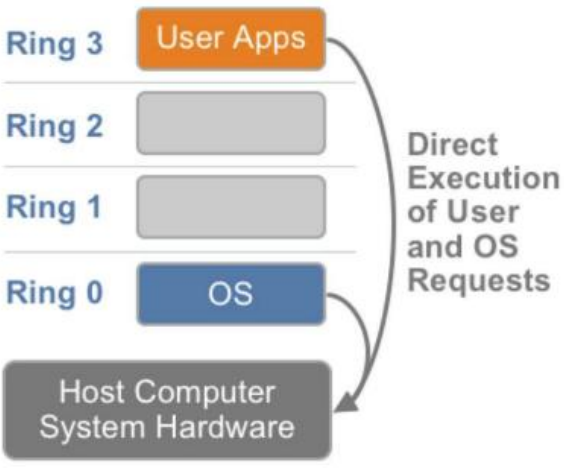
-**Native Virtualization**: implement a layer shielding the hardware. Then provide an instruction set of the hardware in form of an interface (alternatively also provide instruction set of other hardware). We call the layer a **native virtual machine monitor (VMM)** (also called *hypervisors*; native, since it sits directly on top of the hardware). Interface of a VMM can be used by multiple guest OS at the same time.

-**Hosted Virtualization**: compared to a native VMM, which needs to implement device drivers for all hardware resources, a **hosted VMM** runs on top of a host OS (using existing functionality of host OS). Hosted VMM needs special privileges to work (cannot be run as simple user application), since general instructions are usually executed directly. Privileged instructions are more complicated

→ VMMs are much faster than emulators, <u>but</u> they have some implementation issues (e.g. handling privileged instructions)

Privileges: Many architectures of physical machines operate with privileges. A x86 architecture has 4 rings of privilege (see figure; ring 0 is most privileged).

-**Rings:** usually only ring 0 and 3 are used. **Ring 0** (system/kernel mode): code can access hardware directly. Only low-level trusted parts of OS run in this mode. **Ring 3** (user mode): no direct access to hardware. Accessing hardware is delegated to OS. **Ring 1** and **Ring 2** are rarely used.

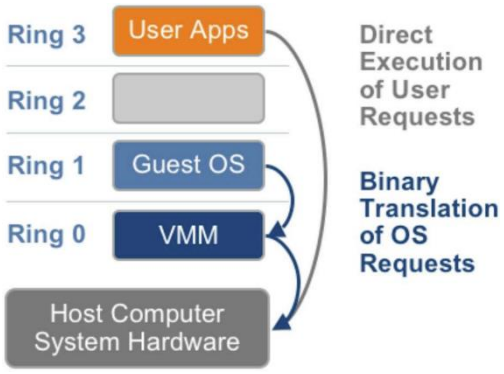


-**transitioning** between modes is expensive but makes the system much more **secure**. If a **user mode** code oversteps its bounds, the system throws an exception, and the application crashes instead of the whole system. Crash in **system mode** brings the whole system down.

-**non-privileged instructions**: can be run on a virtualized architecture in a non-privileged more, using a VMM. This can improve performance.

-**Full Virtualization**: (using binary translation) is used when we handle **privileged instructions**. Such instructions are trapped by the VMM, which then emulates these instructions.

→guest OS is decoupled from hardware, no modifications necessary, since it is not aware of being virtualized, <u>but</u> not all instructions trap properly



Sensitive Instructions: Are instructions that do not trap properly by the VMM in Full Virtualization (see above). A **control sensitive instruction** affects the configuration of a machine, e.g. by changing the memory layer output or interrupt table. The effect of a **behavior-sensitive instruction** depends in the context, e.g. affects certain registers depending on whether it is run in system or user mode. If all sensitive instructions were privileged, all of them would be trapped properly.

-but not all architectures have privileged-only sensitive instructions. E.g. x86 has **17 sensitive non-privileged instructions**, each of which can be run in user mode without trapping.

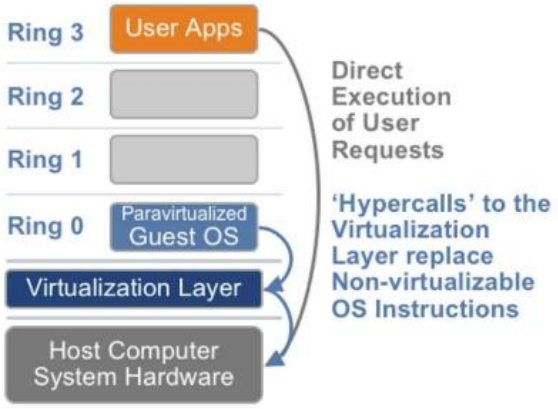
-we do not want to emulate *all* instructions. So, we could scan the code for problematic instructions, and then insert code to **divert control** to the VMM for them.

-but this makes virtualization much more **difficult to implement**.

Paravirtualization: is another solution to the problem of *sensitive instructions*. The guest OS is modified, replacing all problematic instructions with call to the VMM. But this affects **compatibility** and **portability**.

-compared to Full Virtualization, the guest **OS** needs to be **aware of the virtualization and cooperate** with the VMM (i.e. instead of trapping instructions, call the VMM directly).

-**Modifying** the guest **OS** is usually **easier** than implementing full virtualization for problematic instructions.



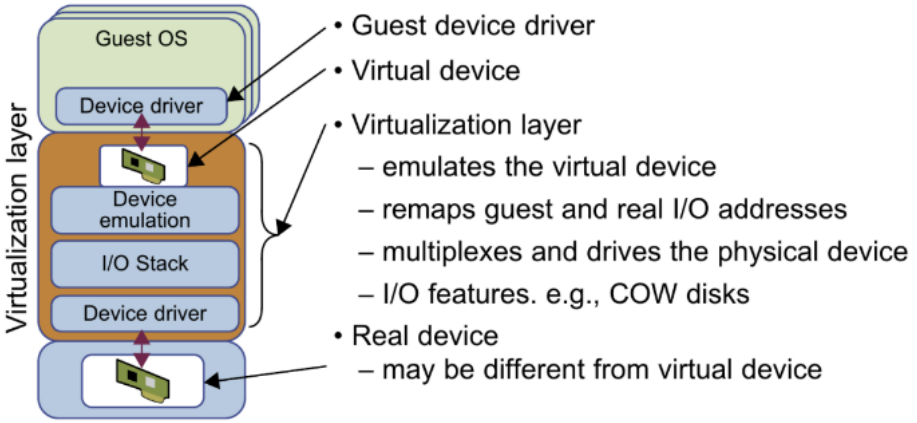
Hardware-Assisted Virtualization: Some hardware supports Virtualization, by introducing a new execution mode below Ring 0. VMM runs in this new **root mode**. Then privileged instructions are automatically trapped to the VMM.

Memory Virtualization: is similar to virtual memory in OSs. Applications have their own address space, which is not directly connected to physical memory. An OS uses page tables to map virtual pages to physical pages.

-is **supported by hardware**: *Memory Management Unit* (**MMU**) that handles all memory accesses, and *Translation Lookaside Buffer* (**TLB**) that supports MMUs by caching virtual memory to physical memory translations.

-running multiple Virtual Memories (VM) in a system needs virtual MMUs, which adds another layer to a memory lookup

I/O (input/output) and Device Virtualization: virtual devices require mapping them to physical devices. Emulating a device is a common way of doing this. (see figure)



Chapter 15: Concrete Systems: Cluster Managers

Look at two types of cluster managers: YARN and Mesos.

Hadoop Version 1: is an early Hadoop version and tightly couples HDFS(Hadoop File System)with map reduce

-**Workflow**: User **specifies job** with location of input/output files, implementation of map and reduce functions (in jar file), and various parameters describing the job. Hadoop sends this information to the **JobTracker**, which configures the worker nodes and plans/supervises the individual tasks. The **TaskTrackers** execute the tasks on the individual nodes and write the result of the reduce step into the output files.

-**JobTracker**: takes care of Job Scheduling (matching tasks with TaskTrackers), and Task Progress Monitoring (i.e. keeping track of tasks, restarting failed or slow tasks, doing task bookkeeping).

→ will be **overwhelmed** at some point. Since it manages both jobs, tasks and resources, monitors current state, and is responsible for fault tolerance (i.e. replacing current state, which is too much work for rapidly changing/updating states)

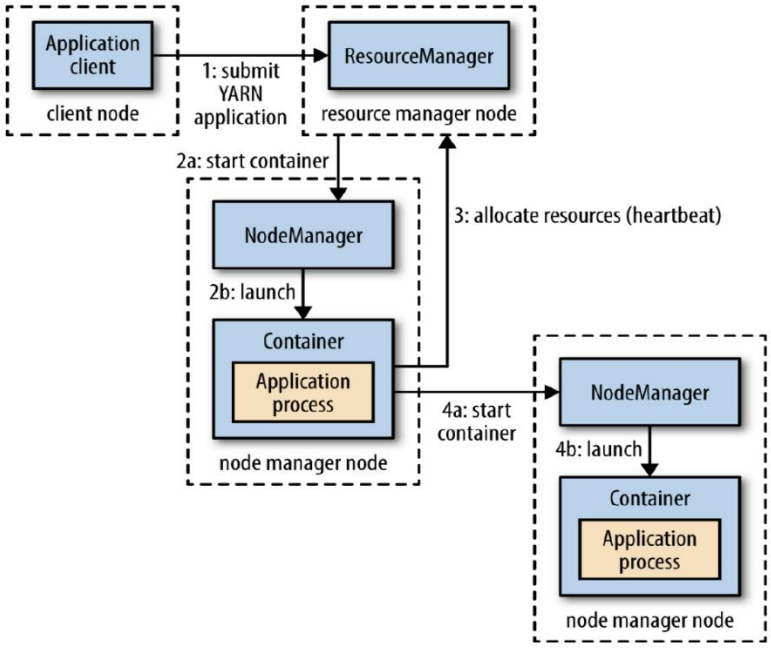
-only **scaled** up to 4.000 nodes and 40.000 tasks.

-**Unflexible Utilization**: Each task tracker has an allocation for fixed-size slots, where each slot was configured as either a map or a reduce slot, which could not be switched (caused unnecessary waiting)

YARN (Yet Another Resource Negotiator): was introduced in Hadoop Version 2 as a resource manager between map reduce and HDFS. Allowed Hadoop to scale to 10.000 nodes and 100.000 tasks, and could run other applications besides map reduce.

-**Master-Slave Architecture**: Master is the *ResourceManager* (**RM**), responsible for managing/allocating resources and scheduling applications. Slaves are *ApplicationMasters* (**AMs**), handling application lifecycles, including faults. One RM per cluster, and one AM per application.

-**ResourceManager (RM)**: runs as a master. Additionally, each node runs a NodeManager (NM), which handles local resources on a singe node. RM and NMs are run in master-slave setup. Resources on local nodes are



bundled up in containers (logical representation of resources, like memory/cores).
-**ApplicationMaster (AM)**: is started as the first container of an application. Registers with the RM, and can then update about its status (heartbeat), request resources, and run the application. An AM does not run as a privileged service, since it runs the user code, which could be malicious.
-**Resource Request**: when requesting a set of containers, AM can specify memory, CPUs and **locality**. This means a container can be requested on a specific node, rack, and location in the cluster. This allows a task to run near the node that stores the data, keeping network traffic low.
-**Scheduling**: YARN runs **3 different kinds of schedulers**. 1. **FIFO**: as previously described. 2. **Capacity**: runs separate queue for small jobs. 3. **Fair**: divides resources equally among jobs.

-**Capacity Schedulers**: allow sharing of Hadoop cluster among user groups. Each group is allocated a queue with certain capacity. A group can further subdivide a queue hierarchically. Within a queue, jobs are treated like in FIFO. For utilization, queues can be configured to be *elastic*. Spare resources might be allocated exceeding a queues capacity temporarily. Upper bound for max capacity can be defined. Queue below capacity has to wait for jobs in another queue to finish to get back the full capacity.

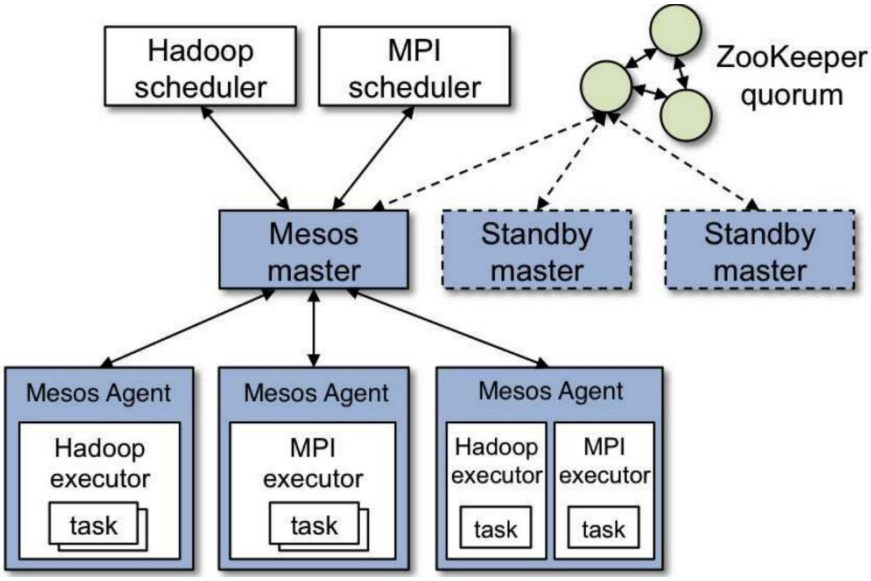
-**Fair Schedulers**: By default, all jobs get equal share of resources. Dominant resource fairness is also supported. But fair sharing can also be applied between and within queues.

Apache Mesos: is built to run **multiple** distributed **frameworks** and applications efficiently on the **same cluster**, allowing for **dynamic resource sharing** and management across diverse workloads such as Hadoop, Hive, and Flink, as well as **different configurations** of the same framework.

-**Motivation**: partitioning the cluster would assign a partition to each framework. Disadvantages of this are **utilization** (each framework can only use its partition, even if other parts are idle), and **data-sharing** (often same data is used and users do not want to copy it between partitions)

→ want to share resources dynamically.

-**Framework Requirements**: **depends** on the specific framework, e.g. large long-running batch jobs, short interactive sessions, fast real-time processing. Sometimes even new frameworks are launched. Thus, building and maintaining a such flexible cluster manager is very **difficult**.



-**Architecture**: like master-slave architecture (master manages agents running on each node). But Mesos delegates control to achieve flexibility. Scheduling and task execution is delegated to *Mesos Frameworks*.
(See Figure: Mesos Components (blue), Delegated Part (white))

-**Mesos Frameworks**: A framework on Mesos has 2 components: 1. **Scheduler**, that registers with master, 2. **Executor Processes** that are launched on agents. The master controls cluster resources, i.e. determines how many resources to **offer** to a framework (framework scheduler decides how many to use). When scheduler accepts offer, it passes task description to Mesos, which launches the tasks on the corresponding agents.

-**Resource Offers**: The master *offering* resources to a framework, makes Mesos very **flexible**. To know what resources are desired (so offers are not constantly rejected), frameworks can specify filters, so only offers, which pass the filter are forwarded. A framework might still reject a passed offer.
-**Allocated Module**: is a "pluggable" module, i.e. user can choose allocation policy for cluster. Used can use *Strict Priority Allocation*, Dominant Resource Fairness (DRF), or implement own policy.
-**Fault Tolerance**: what happens if

Master Fails: Zookeeper (distributed coordination service) holds leader election among a group of additional standby master nodes. The elected node takes over. To reconstruct the state, it reads all important information from the schedulers and executor processes.
Scheduler Fails: Recovery is delegated to the schedulers. Two or more schedulers are always registered with the master, so the master just switches to another. Consider this when implementing a scheduler (include mechanism for sharing its state).
Agent Fails: Master constantly monitors agents. If agent does not respond for a while, master assumes it failed. But agent regularly takes snapshots of state.

Table of Contents

Classwork Tutorials

Raft: is a distributed consensus algorithm designed to manage a replicated log (e.g. data update, system change) in a distributed system. Its fault tolerance and performance are equivalent to Paxos, but Raft is written to be more understandable. Possible node states are: *Leader, Follower* and *Candidate*.

-**Leader**: is responsible for managing the log (receives update form the user, and propagates them)
-**Follower**: replicates the log entries, received from the leader
-**Candidate**: is a node attempting to become a leader (by majority election).
-**Terms**: is a counter, which each node increments by one when it was involved in electing a new leader. If after a network partition, two leaders reconcile, the leader with less terms steps down.
-**Leader Election**: *Trigger*: When a Follower does not receive heartbeats (regular messages from leader) within a timeout period, it becomes a candidate. *Vote Request*: Candidate increments its terms and sends vote request to the other nodes. *Voting*: Nodes vote for candidate, if they havenßt yet voted in the current term, and if candidate‘s log is at least as up to date as theirs. *Majority*: Candidate becomes leader, if it receives votes from majority of nodes.
-**Log Replication**: Leader receives client request to change system state. Leader appends request as new entry in its log and sends AppendEntries message to followers, to replicate the entry. When majority sends acknowledgements to leader, it commits the entry. Leader sends commit notification to followers, who then also commit the entry.
-**Log Matching**: If two logs contain same entry at given index, then logs are identical up to that index.
-**Commitment Guarantee**: An entry is committed when it is stored on a majority of nodes, ensuring that it will remain committed in the future.
-**similar to 2PC**: wrt. replication process, but instead of requiring acknowledgment from all nodes, the leader only needs confirmation from a majority.
-**Network Partition**: minority partition cannot obtain majority to elect leader. Thus it ends in continuous election attempts without success. This prevents them from progressing in terms.
-to be elected, a node must be the **most up-to-date**, to receive votes from other nodes. Otherwise nodes will not vote for it.
→Raft ensures **robustness** in handling leader failures and network partitions. This also guarantees **consistency**.

Redis (p.11): is a NoSQL Key-Value Store, which offers much more in terms of functionality and performance. It uses specialized data structures to store and manipulate data. These data structures are: *Strings, Lists, Sets, Sorted Sets, Hashes*. Each structure has unique methods and is optimized for different types of operations.

-**Redis Databases**: are identified by numbers (default is 0), between which you can switch using the command: e.g. `select 1`
-**Keys and Values**: keys are strings used to identify data. Values can be of various types (e.g. strings, integers, or serialized objects (JSON, XML)). Redis treats values as byte arrays (does not check type), allowing flexibility in data storage. A key might look like this: `users:leto`
-**Store Data**: using the command set key value. Example:
```
set users:leto '{"name": "leto", "planet": "dune", "likes": ["spice"]}'
```
-**Retrieve Data**: using the command get key, e.g. `get users:leto`
-Redis does not support querying values directly, but **only queries keys**.
-**in-memory store** with persistence options: **Snapshotting** saves the database to the disk based on number of changes over time. **Append-Only File** updates a log file on disk every time a key changes.
→ exceptionally fast (thanks to in-memory store), but higher RAM usage.

-**Strings**: are simplest data structure in Redis, and often used for key-value pairs. They are useful for storing objects, counters, and caching data Valid operations are **`get`** and **`set`** (as in examples above), and typical string operations like **string length, range, append**

`strlen users:leto` OR `getrange users:leto 31 48`

OR `append users:leto " OVER 9000!!"`

-and also **increment** and **decrement** (specific to numeric values):

`incr stats:page:about` AND `incrby ratings:video:12333 5`

-**Hashes**: are more structured than strings because they allow more granular updates and retrievals without needing to manipulate the entire value. This makes them ideal for representing objects with multiple attributes. Some basic Hash operations are:

`hset users:goku powerlevel 9000` OR `hget users:goku powerlevel`

OR `hmset users:goku race saiyan age 737`

OR `hmget users:goku race powerlevel` OR `hgetall users:goku`

-**Lists**: are ordered collections of values, allowing efficient index-based operations, like `lpush newusers goku` OR `ltrim newusers 0 49`. Lists maintain the order of elements and are useful for tasks like tracking recent users or logging actions. They support operations to push, pop, and trim elements, making them versatile for various ordered data needs.

-**Sets**: store unique values and support set-based operations (e.g. intersections, unions). Useful for scenarios like friends lists or tagging systems. Some example operations are:

`sadd friends:leto ghanima paul chani jessica`

OR `sadd friends:duncan paul jessica alia`

OR `sismember friends:leto jessica`

OR `sinter friends:leto friends:Duncan`

OR `sinterstore friends:leto_duncan friends:leto friends:duncan`

-**Sorted Sets**: are like sets with an additional score that determines the order. Thus enabling ranking and sorting elements by the score. Perfect for e.g. leaderboard systems. Example operations:

`zadd friends:duncan 70 ghanima 95 paul 95 chani 75 jessica 1 vladimir`

OR `zcount friends:duncan 90 100`

OR `zrevrank friends:duncan chani`

-**Runtime Performance**: of some algorithms in Redis is O(1) (`sismenber`), O(log(N)) (`zadd`), O(N) (`ltrim`), O(long(N)+M) (`zremrangebyscore` i.e. removing elements in score list, N total elements, M affected elements), O(N+M*log(M)) (`sort`)

-**Pseudo Multi Key Queries**: We can assign two different keys to the same value for more flexibility.

Example: `set users:9001 '{"id": 9001, "email": "leto@dune.gov", ...}'`

AND `hset users:lookup:email leto@dune.gov 9001`

 # To then get a user by ID: `get users:9001`

 # Or to get a user by email: `id = redis.hget('users:lookup:email', 'leto@dune.gov')` **AND** `user = redis.get("users:#{id}")`

-**References and Indexes**: we can manually manage indexes and references between values. Example:

`sadd friends:leto ghanima paul chani jessica`

#If Chani changes name or deletes her account: `sadd friends_of:chani leto paul`

-**Round Trips and Pipelining**: Redis supports batch operations and pipelining, to minimize network overhead. **Examples**: **Using mget:** `ids = redis.lrange('newusers', 0, 9)`

AND `redis.mget(*ids.map {|u| "users:#{u}"})`

Using sadd: `sadd friends:vladimir piter`

AND `sadd friends:paul jessica leto "leto II" chani`

-**Transactions**: ensure **atomic execution** of multiple commands. Example: `multi` AND `hincrby groups:1percent balance -9000000000` AND `hincrby groups:99percent balance 9000000000` AND `exec`

-**watch key** to conditionally apply transaction: `redis.watch('powerlevel')` AND `current = redis.get('powerlevel')` AND `redis.multi()` AND `redis.set('powerlevel', current + 1)` AND `redis.exec()`

→ ensure this fails when the client changes the value of `'powerlevel'`

-**Keys Anti-Pattern**: Instead of keys command (high complexity O(N)), use hashes to organize data:

Instead of using keys: `keys bug:1233:*`

Use a hash: `hset bugs:1233 1 '{"id":1, "account": 1233, "subject": "..."}'`

AND `hset bugs:1233 2 '{"id":2, "account": 1233, "subject": "..."}'`

To get all bug IDs for an account: `hkeys bugs:1233`

To delete a specific bug: `hdel bugs:1233 2`

To delete an account: `del bugs:1233`

HBase (p.11/12): is a NoSQL columnar store (which stores data in cells, which are part of columns.

-**create table**: by specifying the table name and the column family name: `create 'test', 'cf'`

Multiple column families: `create 'wiki_small', 'page', 'author'`

-**confirm table exists**: `list 'test'`

-**see table details and configuration defaults**: `describe 'test'`

-**put data into table**: `put 'test', 'row1', 'cf:a', 'value1'`

OR `put 'test', 'row2', 'cf:b', 'value2'`

OR `put 'test', 'row3', 'cf:c', 'value3'`

-**fetch all data from table**: `scan 'test'`

-**get one row**: `get 'test', 'row1'`

-**disable a table**: before deleting or changing settings `disable 'test'`

-**re-enable the table**: `enable 'test'`

-**delete table**: `drop 'test'`

-**get column (family)**: `get 'test', 'row3', {COLUMN => 'cf'}`

OR `get 'test', 'row3', {COLUMN => 'cf:a'}`

-get data with specific **timestamp**:

`get 'test', 'row3', {COLUMN => 'cf:c', TIMESTAMP => xxx}`

-force **flush** table data to disk: `flush 'test'`

-**clean up deleted data** by forcing major compaction: `major_compact 'test'`

-**alter table schema**: `alter 'test', NAME => 'cf', VERSIONS => 3`

-**delete** data **by timestamp** (xxx): `delete 'test', 'row3', 'cf:c', xxx`

-**return multiple versions** of data: `scan 'test', {VERSIONS => 3}`

-**count rows**: `hbase org.apache.hadoop.hbase.mapreduce.RowCounter 'wiki_small'`

-**display available filters**: `show_filters`

-**Filters**: can be used to query data. **Examples**:

```
scan 'wiki_small', {STARTROW=>'100009', ENDROW=>'100011'}
scan 'wiki_small', {COLUMNS => ['page:page_title',
                'author:contributor_name'], ROWPREFIXFILTER => '1977'}
scan 'wiki_small', {COLUMNS => ['page:page_title',
'author:contributor_name'], FILTER => "SingleColumnValueFilter('author',
'contributor_name', =, 'substring:tom')"}
```

```
scan 'wiki_small', {COLUMNS => 'author:timestamp', FILTER =>
"ValueFilter(=, 'substring:2017')"} scan 'wiki_small', {COLUMNS =>
'page:page_title', FILTER => "ValueFilter(=, 'substring:Sydney')"}
```

MongoDB (p.12): is a NoSQL Document Store, which uses JSON and BSON to represent documents. Start with some crud operations (create, read, update)

-**Create**: adds new documents to a collection. Collection is automatically created, if it does not exist. All write operations are atomic on the level of single documents. Une functions `db.collection.insertOne()` or `db.collection.insertMany()`

-**example**: `db.users.insertOne({name:"sue", age: 26, status: "pending"})`

-**Read**: retrieve documents from collection, based on query filters. Use: `db.collection.find()`

-**example**: `db.users.find({age:{$gt:18}},{name: 1, address: 1}).limit(5)`

-**Update**: modify existing documents in collection. Operations are atomic on level of single document. Use: `db.collection.updateOne()` OR `db.collection.updateMany()`

OR `db.collection.replaceOne()`

-**example**: `db.users.updateMany({age:{$lt:18}},{$set:{status:"reject"}})`

-**Delete**: documents from a collection. Deletions are atomic on the level of operations. Use:

`db.collection.deleteOne()` OR `db.collection.deleteMany()`

-**example**: `db.users.deleteMany({status:"reject"})`

--**Query in MongoDB using** `db.collection('inventory').find();` **function.** All following examples describe combinations of symbols that can be written inside the `find()` argument. (leave out the part before for simplicity). Insert any types of documents: `db.collection('inventory').insertMany([...]);`

-**Query Documents**

-select **all** documents: `{}`

-`status` **equals** value `D`: `{status:"D"}`

-`status` **equals** `A` **or** `D`: `{status:{$in:['A', 'D']}}`

-***AND* condition**: `{status:'A',qty:{$lt:30} }`

-***OR* condition ($lt means 'less than')**: `{$or:[{status:'A'},{qty:{$lt:30}}]}`

-**AND + OR conditions**:
```
{ status: 'A',
        $or: [{ qty: { $lt: 30 } }, { item: { $regex: '^p' } }] }
```

-**Query Embedded/Nested Documents**: can be done using dot-notation, e.g. `"field.nestedField"`. Note that field and nested field must always be inside the quotation marks.

-all documents where nested field `uom` in the filed `size` **equals** `'in'`: `{'size.uom': 'in'}`

-use **`$lt`** in nested field h: `{ 'size.h': { $lt: 15 } }`

-**AND condition** in nested field: `{'size.h':{$lt:15},'size.uom':'in',status:'D'}`

-specify **several** nested fields: `{ size: { h: 14, w: 21, uom: 'cm' } }`

-**Query Array Fields**

-**Equality condition**: e.g. query documents where the field `tags` is an array of the form `['red', 'blank']` example: `{tags: ['red', 'blank']}`

-array **includes** element: check for documents where `tags` is an array that contains the string `'red'`, example query: `{tags: 'red'}`

-**Conditions**: on array elements, e.g. documents where `dim_cm` **contains at least one** element with value **greater than** 25: `{dim_cm: { $gt: 25 }}`

-**Multiple Conditions**: can specify whether a single array or any combination of array elements meets that condition: `{dim_cm: { $gt: 15, $lt: 20 }}`i.e. dim_cm contains such array

-**multiple criteria** for one array **element**: `dim_cm:{$elemMatch:{$gt: 22, $lt: 30}}}`

-query **element** by array **index position**: `dim_cm.1` denotes the second element of the array `dim_cm`. **example**: `{ 'dim_cm.1': { $gt: 25 } }`

-**array length**: documents where array `tags` has 3 elements: `{tags: { $size: 3 }}`

-**Query Arrays of Documents**

-**condition** for elements of array `instock` example: `{instock:{warehouse:'A',qty:5}}`

-**equality match**: `{instock: {qty: 5, warehouse: 'A'}}`

-check documents where `instock` array has at least one **embedded document** that **contains** the **field** `qty` with specified condition: `{'instock.qty': { $lte: 20}}`

-check for first element of array `instock`: `{'instock.0.qty': { $lte: 20 }}`

-find embedded document in array fulfilling: { instock: { $elemMatch: { qty: 5, warehouse: 'A' } } }

-document in array `instock` has its `qty` field with conditions: {'instock.qty': { $gt: 10, $lte: 20 } }

-**Query Operations with Projection**: restrict which fields are included/excluded in the documents returned by the query. Use 0 and 1 like with Booleans. By default, the **`_id`** field is **always returned**.

-**example**: `.find({ status: 'A' }).project({ item: 1, status: 1 });`
returns only the fields `_id, item` and `status` from each document.

-**exclude _id field**: `.project({ item: 1, status: 1, _id: 0 });`

-**exclude any field**: `.project({ status: 0, instock: 0 });`

-**embedded documents**: `.project({ item: 1, status: 1, 'size.uom': 1 });`

-project **specific elements**: to be included/excluded in the final document using the operators `$elemMatch`, `$slice`, and `$`. **Example**: .project({ item: 1, status: 1, instock: { $slice: -1 } });

-**Query Operations for Null Values**: check for equality is analogous**:** `{item: null}`

-Check for **non-null value**: `{item: { $ne: null }}`

-**type check** contain item of value `'null'` (which is BSON type 10): `{item:{ $type: 10}}`

-**Existence Check**: `{item:{ $exists: false } }`

--**Further Information about MongoDB**:

-**Sharding**: is done using 3 main components: *Shard, Config Server, and Query Router (mongos)*. These components can be mapped to HDFS components: Shard (DataNode), Config Server (NameNode), Query Router (HDFS Client).

-**Replication**: is based on replica set (for redundancy and high availability). It consists of primary node (handles write operations) and multiple secondary nodes (replicate data; can become primary).

-**Replication + Sharding**: Every shard is implemented as replica set. Secondary nodes do re-election if primary node fails. Recommended to have a least one primary node, and two secondary nodes.

-**CAP Theorem**: MongoDB focuses on availability and Partition Tolerance. Eventual Consistency rather than strong consistency. → AP System.

Cypher (Neo4J) (p.12): Neo4J is a NoSQL Graph-Based system and uses the query-based language **Cypher**.

-**MATCH**: Retrieve data based on specific patterns:

```
MATCH (p:Person)-[:FRIEND_OF]->(f:Person)
RETURN p.name, f.name;
```

-**CREATE**: Create nodes and relationships: `CREATE (p:Person {name: 'Alice'})`

```
CREATE (f:Person {name: 'Bob'}) CREATE (p)-[:FRIEND_OF]->(f);
```

-**RETURN**: Specify what data to return from a Cypher query: `MATCH (p:Person) RETURN p.name;`

-WHERE: Filter data based on conditions.

```
MATCH (p:Person) WHERE p.age > 30 RETURN p.name;
```

-**MERGE**: Ensure a pattern exists in the graph, creating it if it doesn't already exist.

```
MERGE (p:Person {name: 'Alice'}) ON CREATE SET p.created = timestamp();
```

-**SET**: Set properties on nodes and relationships.

```
MATCH (p:Person {name: 'Alice'}) SET p.age = 35;
```

-**DELETE**: Remove nodes and relationships from the graph.

```
MATCH (p:Person {name: 'Alice'})-[r:FRIEND_OF]->() DELETE p, r;
```

-**WITH**: Chain together multiple clauses in a Cypher query.

```
MATCH (p:Person) WITH p WHERE p.age > 30 RETURN p.name;
```

-**UNWIND**: Transform a list into rows.

```
UNWIND [1, 2, 3] AS number RETURN number;
```

-**ORDER BY**: Sort the results of a query.

```
MATCH (p:Person) RETURN p.name ORDER BY p.age DESC;
```

-**LIMIT**: Limit the number of results returned by a query.

```
MATCH (p:Person) RETURN p.name LIMIT 10;
```

Kafka (p.21; only mentioned): is a distributed streaming platform, which allows real-time data capture, storage, processing and routing, to enable continuous data flow and integration across various systems.

-**3 key capabilities**: 1. Publish and subscribe to event streams, 2. Store event streams durably and reliably, 3. Process event streams in real-time retrospectively.

-distributed, highly scalable, flexible, fault-tolerant, secure.

-**Servers**: run as cluster (brokers) for storage, and Kafka connects them for data import/export.

-**Clients**: applications and microservices reading, writing and processing streams of events.

-**Event**: consists of key, value, timestamp and metadata.

-**Producers** write events to Kafka, **Consumers** read and process events. **Topics** organize and store events, like a folder. **Partitions** are distributed into buckets for topic data (enable parallelism and scalability; events with same key are written to same partition). **Replication**: ensures fault-tolerance and high availability.

-**APIs**: **Admin API** manages and inspects topics, brokers and Kafka objects. **Producer API**, publish a stream of events to Kafka topics. **Consumer API** subscribe to and process streams of events. **Kafka Streams API** implement stream processing applications and microservices. **Kafka Connect API** build and run connectors from data import/export to integrate with external systems.

-**Create Topics**: `bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic test`

-**List existing Topics**: bin/kafka-topics.sh --bootstrap-server localhost:9092 –list

-**Produce message to a Topic**: bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test

-**consume message from a topic**: `bin/kafka-console-consumer.sh --topic test --from-beginning --bootstrap-server localhost:9092`

-**delete Topic**: bin/kafka-topics.sh --bootstrap-server localhost:9092 --delete --topic test

-**Kafka with Python**: (do `pip install kafka-python`).

-**example Producer**:

```
from kafka import KafkaProducer
producer = KafkaProducer(bootstrap_servers='localhost:9092')
producer.send('temperature', b'{"value": 23.5}')
```

-**example Consumer**:

```
from kafka import KafkaConsumer
consumer = KafkaConsumer('temperature',bootstrap_servers='localhost:9092')
for message in consumer: print(message.value)
```

Apache Spark (p.19): is a unified analytics engine for large-scale data processing, supporting high-level APIs in Java, Scala, Python, and R, as well as an optimized engine that supports general execution graphs.

-**Creating DataFrame**: `myRange = spark.range(1000).toDF("number")`

-**Filtering Data**: `divBy2 = myRange.where("number % 2 = 0")`

-**count rows**: `divBy2.count()`

-uploading and reading **csv files**: `from google.colab import files` **AND** `uploaded = files.upload()` **AND** `flightData2015 = spark.read.option("inferSchema", "true").option("header", "true").csv("2015-summary.csv")` **AND** `flightData2015.show(10)`

-**Sort data**: `flightData2015.sort("count").show(3)`

-**Explain execution plan**: `flightData2015.sort("count").explain()`

-**SQL Queries**:
```
flightData2015.createOrReplaceTempView("flight_data_2015")
    sqlWay = spark.sql(""" select dest_country_name, count(1) from
        flight_data_2015 group by dest_country_name """)
    sqlWay.show(3)
```

-**Word Count using Map Reduce**:
```
text = sc.textFile("file.txt")
textsplit = text.flatMap(lambda line: line.split(" "))
tuples = textsplit.map(lambda word: (word,1))
counts = tuples.reduceByKey(lambda a, b: a + b)
counts.collect()
```

-**Inverted Index using Map Reduce**:
```
documents = sc.wholeTextFiles("files")
docsplit = documents.flatMap(lambda tuple: [(tuple[0],word)
        for word in tuple[1].split(" ")])
docsplit.collect()
```

-**Read Streaming Data**:
```
streaming = spark.readStream.schema(dataSchema)\
    .option("maxFilesPerTrigger", 1).json("data/activity-data")
```

-**Aggregation Query on Streaming Data**:
```
activityCounts = streaming.groupBy("gt")\
    .count()
activityQuery = activityCounts.writeStream \
.queryName("activity_counts").format("memory").outputMode("complete").start()
spark.sql("select * from activity_counts").show()
activityQuery.stop()
```

-**Transformation Query on Streaming Data**:
```
simpleTransform = streaming.where("gt = 'bike'").where("Device ='nexus4_2'")
transformQuery = simpleTransform.writeStream.queryName("simple_transform")\
.format("memory").outputMode("append").start()
spark.sql("select * from simple_transform").show()
transformQuery.stop()
```

-**Tumbling Windows**:
```
from pyspark.sql.functions import window, col
winCounts = withEventTime.groupBy(window(col("event_time"), "10 minutes"))\
    .count()
winQuery = winCounts.writeStream.queryName("events_per_win")\
    .format("memory").outputMode("complete").start()
spark.sql("select * from events_per_win order by window").show(20,False)
winQuery.stop()
```

-**Sliding Windows**:
```
winCounts2 = withEventTime\
.groupBy(window(col("event_time"), "10 minutes", "5 minutes")).count()
winQuery2 = winCounts2.writeStream.queryName("events_per_win")\
.format("memory").outputMode("complete").start()
spark.sql("select * from events_per_win order by window").show(20,False)
winQuery2.stop()
```

-**Windowing with Watermark**:
```
winCounts3 = withEventTime.withWatermark("event_time", "30 minutes")\
```

```
    .groupBy(window(col("event_time"), "10 minutes", "5 minutes")).count()
winQuery3 = winCounts3.writeStream.queryName("events_per_win")\
    .format("memory").outputMode("complete").start()
spark.sql("select * from events_per_win order by window").show(20,False)
    winQuery3.stop()
```

-**Removing Duplicates in Stream**:

```
winCounts4 = \
          withEventTime.withWatermark("event_time", "30 minutes")\
          .dropDuplicates(["User", "event_time"]).groupBy("User").count()
winQuery4 = winCounts4.writeStream.queryName("events_per_win")\
        .format("memory").outputMode("complete").start()
spark.sql("select * from events_per_win order by User").show()
winQuery4.stop()
```

Spark ML Example Workflow: 1. Define RFormula (for R code later) and Logistic Regression Model, 2. Define pipeline (with stages RFormula, regression model), 3. Initialize the grid searcher, 4. Initialize the binary classification evaluator, 5.do train/test split, training, and evaluation, 6. Get best model from pipeline.

HiveQL: is part of Hive, a data warehouse infrastructure built on top of Hadoop (p.25), providing tools for data analysis and queries using SQL-like syntax.

-**Select one column**: `select orderid from orders;`

-**Join Tables**: Take columns of both tables, and join them on `orderid` and `prodid` to combine related rows. Example: `select * from orders o, order_item i, products p where o.orderid = i.orderid and i.prodid = p.prodid;`

-**Explain Execution Plan**: `explain select orderid from orders;`

-**Aggregation Query**: Group rows in table by `orderid`, and count the number of items for each `orderid`. Example: `select orderid, count(*) from order_item group by orderid;`

-**Aggregation and Sorting Query**: group rows in table `order_item` by `orderid` and count the number of items for each `orderid`, and then sort the results ascending by item count. Example: `select orderid, count(*) as cnt from order_item group by orderid order by cnt;`

-**Query Execution Plans**: **Projection Query** (single phase), **Join Query** (multiple phases, 3 stages, not reduce phase for small relations), **Aggregation Query** (includes reduce phase), **Sorted Aggregation Query** (additional stage for sorting).