Jonas B.

Zusammenfassung Big Data

- Data Independence: User sees only table and not underlaying data structure
  ↳ Logical data model >< physical storage
- Column: Attribute, Field, Property
- Primary Key: Row ID, name
- Row: Item, Entity, Record
- Relation: Set of columns ⟼ Values

Classic relational model:

1. Broken Tabular Integrity: All entries need to have same columns
2. Broken atomicic integrity: Subtables in field
3. Domain integrity: Same type for column

SQL

projection ↙

```
Select name AS who,
       birth-date AS when
From persons
where last_name = 'Crushes'
Order By when DESC
```

```
Select century, Count(*) asc
FROM persons
Group By century
Having C > 2
       ↑ selection
```

```
Select *
FROM (
      Select persons.name AS pname
      FROM persons JOIN ages
        ON persons.age = ages.age
     )
```

- Product expensive

1. Atomic values
2. Key identifies all columns
3. No subkeys

BCNF: nonkey ↛ key

- FROM => WHERE => GROUP BY => HAVING => SELECT
- Declarative / functional language

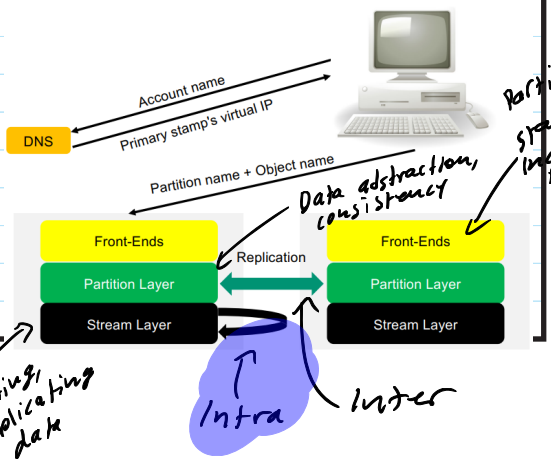
- Update / Insert / Delete Anomaly caused by not being in 2NF
- DDL: Data definition language / DML: "manipulation"

- ACID: · Atomicity: Whole transaction is done at once
  · Consistency: After transaction, back consistent
  · Isolation: Transaction feels like nobody else is writing to Database
  · Duradility: Updates made, don't disappear

Object / Block Storage

- No data hierarchy
- Objects treated as black box
- Scale out (more servers) than scale up (better)
- AWS: BucketID => Object ID => Object (Max 5 TB)
- CAP: · Consistency: All nodes see same data
  · Availadility: Query database at all times
  · Partition Tolerance: Resettle after connection issue
  ↳ Can't have all three
- Fault Tolerance: · Node failure
  ↳ Make some replicas intra-stamp synchronous
  · Natural catastrophe
  ↳ After returning asynchronous
- Differences: Azure Blob vs. Amazon AWS:
  Account->Container->Blob Bucket->Object
  Blackbox

	block blobs	append blobs	page blobs
identifiable by block ID	yes	no	(no)
each block can have different size	yes	yes	no
max size per block (a)	4,000 MB	4 MB	512 Bytes
max # block per blob (b)	50,000	50,000	~15.6 billion (pages)
max size per blob (a*b)	190.73 TB	195.31 GB	8 TB
main function	writing big objects; do not use random read and write	append to the end of blob; update/delete existing blocks -> not supported	random read and write; write to up to 4M concurrently
example usage	Pictures and videos	logs	disks for VM



REST API (Representational State Transfer)

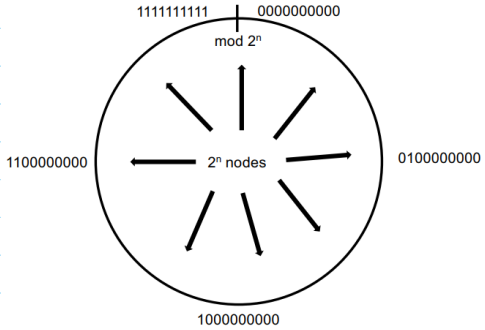
- Part of HTTP
- http://bucket.s3.amazonaws.com/object-name/api?id=foobar#thread
  (id=foobar: query; #thread: fragment)
- HTTP methods:
  - Get: side effect free
  - Put: Idempotent, if you put it once, same effect as 10 times
  - Delete
  - Post: side effects

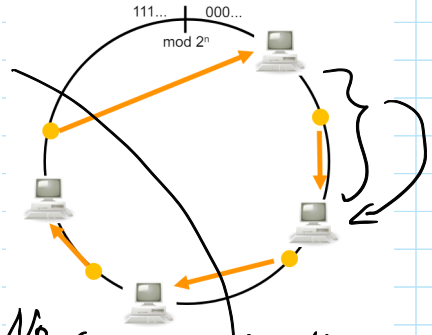
Key-Value store

- Is a datamodel, since it maps ID → object
- No metadata only ID to object
- get(key)
- put(key, value)
- delete ≙ put(key, nothing)
- Eventual consistency: eventually everyone gets same
  - ↳ Better performance
  - ↳ Better scalability
- Incremental stability: Add/remove node easily
- Symmetry: all nodes are the same
- Decentralization: no master node
- Heterogeneity: Don't need exact model configurations

Key-Value store organisation (Dynamo)

1. IDs are organized in a ring
2. Each node picks randomly a hash
3. Nodes are placed on ring based on hash
4. If a new ID+value is stored, we hash the key and it is stored next node clockwise

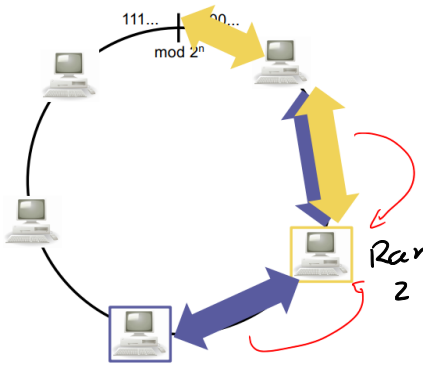




Domain of responsibility (range 1)

No communication needed → blockchain

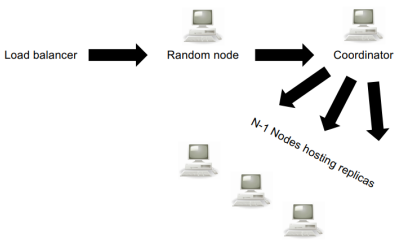
- Replication, if one node fails: ranges



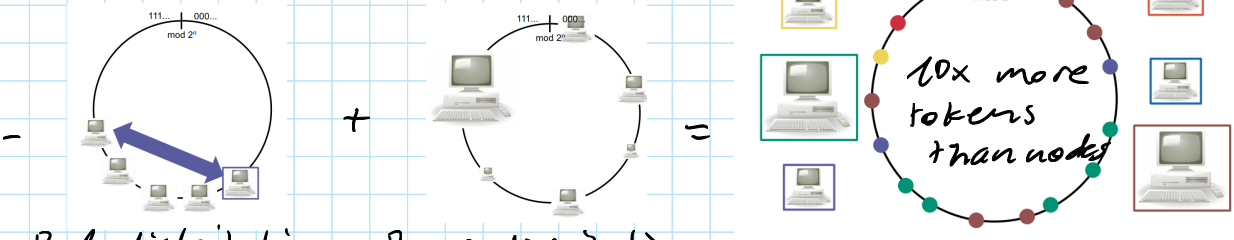
Key	Nodes
a	n1, n2
b	n1, n2
c	n2, n3
d	n2, n3
e	n2, n3
f	n2, n3
g	n2, n3
h	n2, n3
i	n3, n4
j	n3, n4
k	n3, n4
l	n3, n4

- Preference list
- Every node knows about the ranges of the others

↑ coordinator



Virtual nodes as tokens

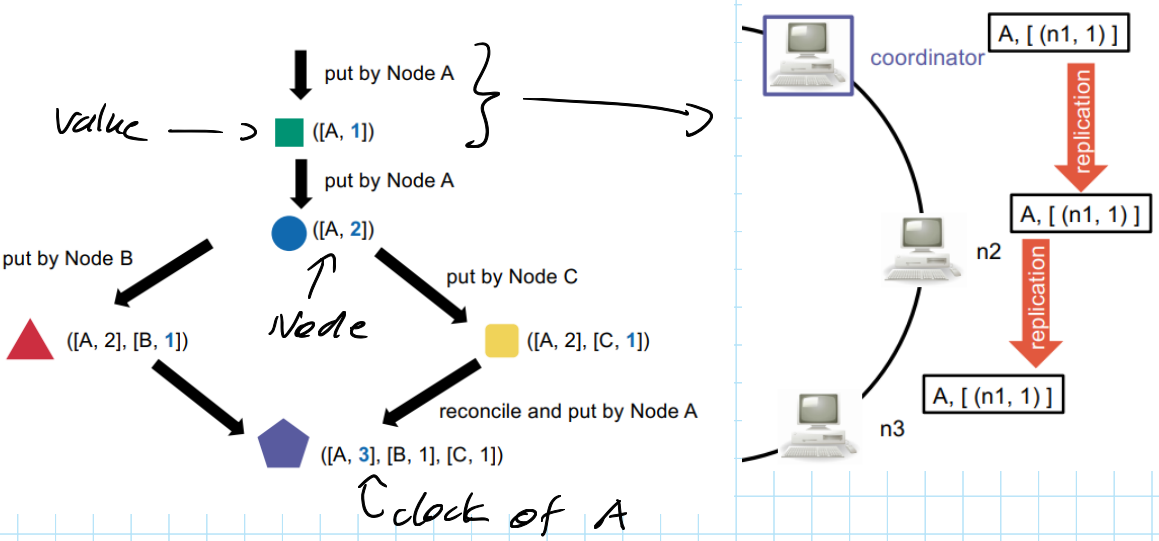


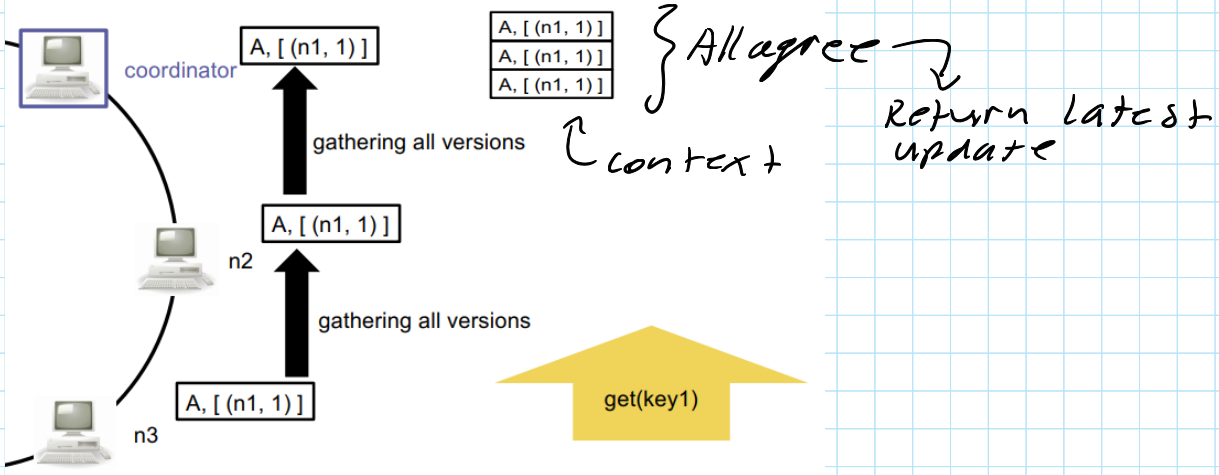
Bad distribution → Power distribution

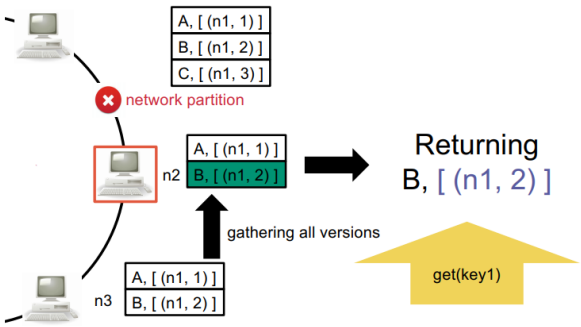
- Vector Clocks

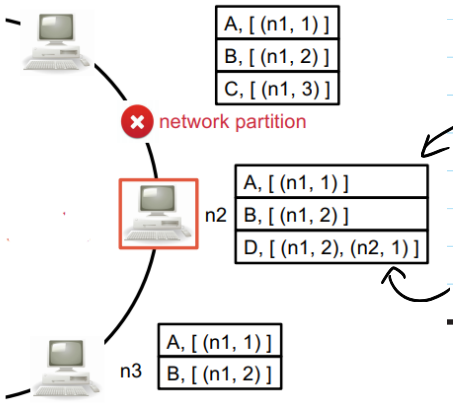
• Solves the issue of eventual consistency

• Each node has a clock



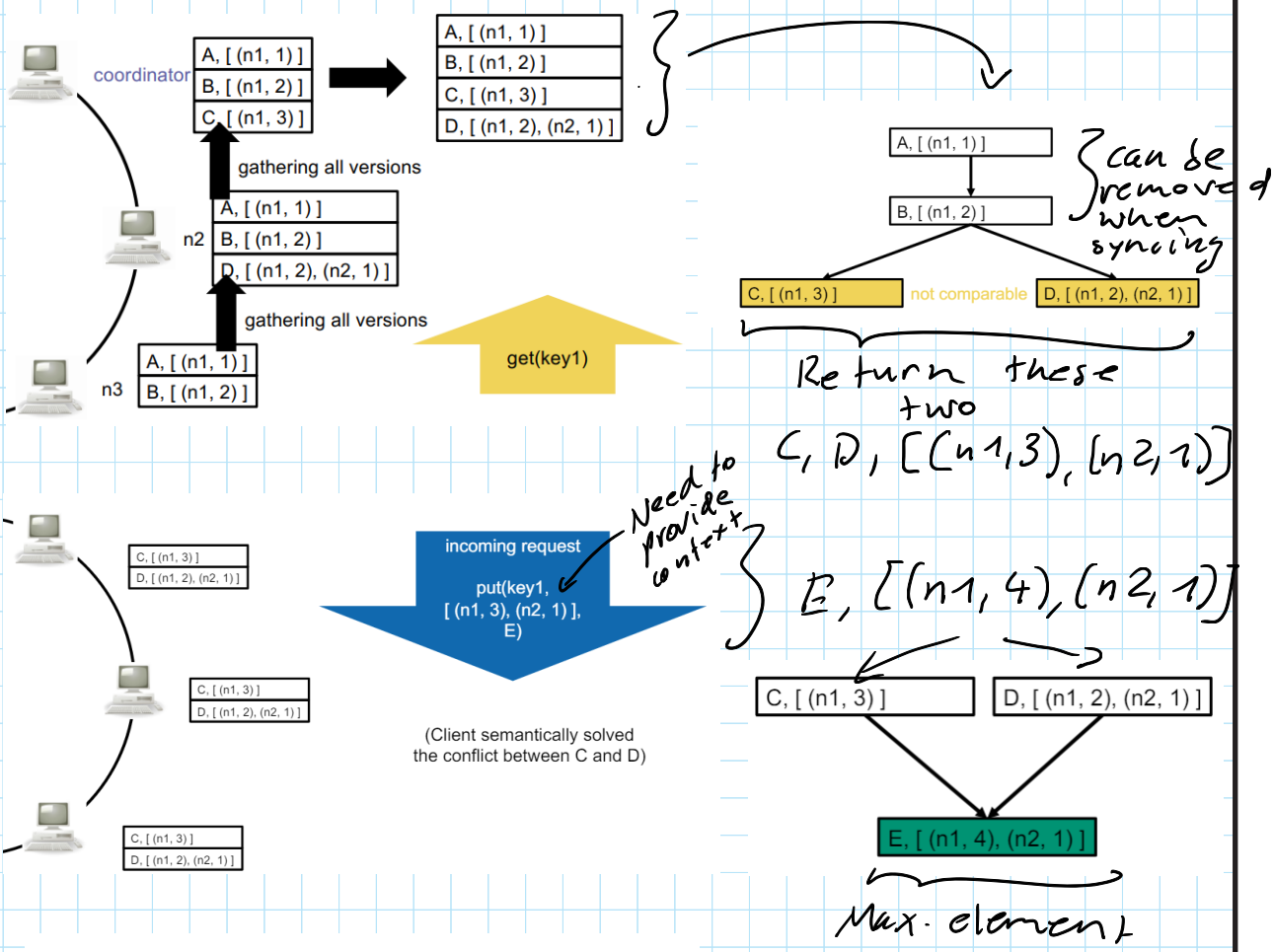






put(key1, [(n1, 2)], d)

Added vector clock of n2 because it was put to n2 first



$$VC(x) \leq VC(y) \iff \forall z[VC(x)_z \leq VC(y)_z]$$

Distributed File Systems

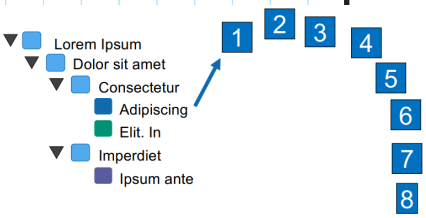
- Object Storage: · Billions of TB files
  · Many small files
  - Random access
  · Flat model: key -> object
  · Object stores as whole

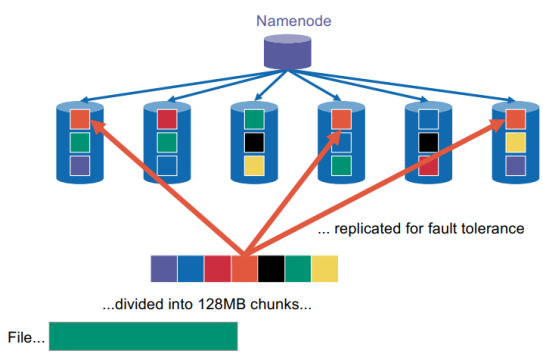
Key-Value Model
Object Storage

- File-Storage: · Millions of PB files
  · Less, big files
  · Scan the file sequentially
  · Single writer append only fashion
  · Top priority: throughput (huge files)
  · File hierarchy
  · Files split up in blocks/chunks (HDFS, Google file system)

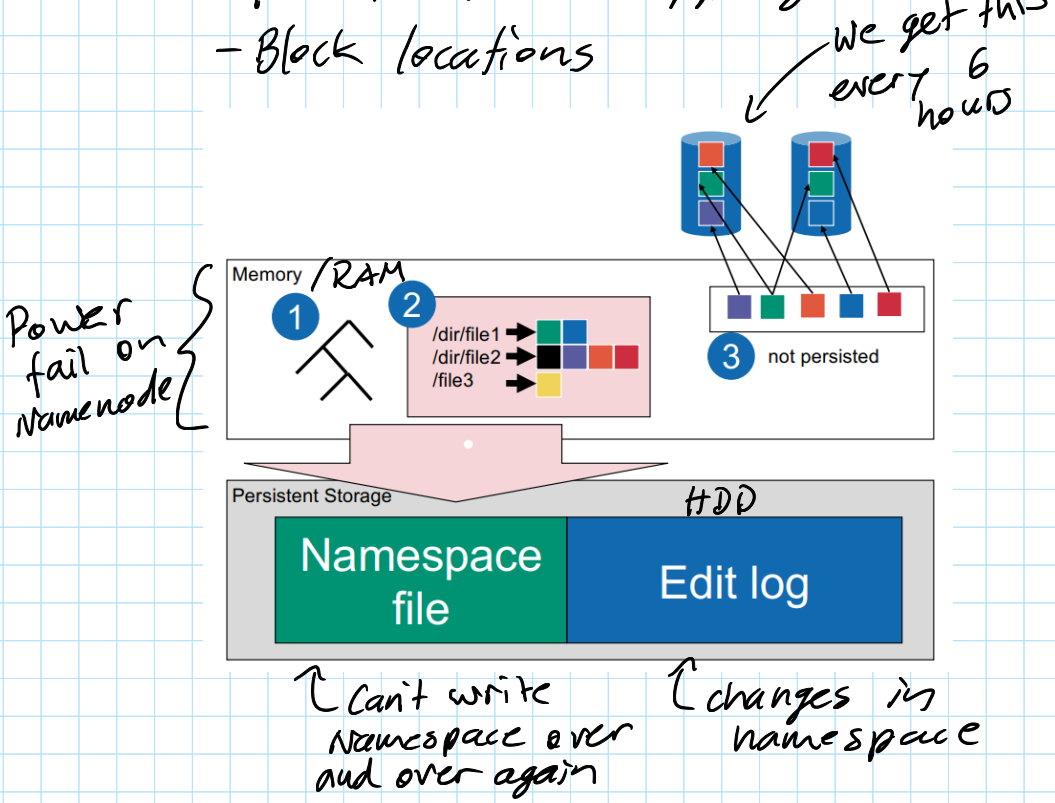
File System
Block Storage

- HDFS: · Hadoop Distributed File System
  · Every file is linked to a list of blocks
  (-> cut files in blocks because they are too big for one machine (PB-files))
  · We have high latency therefore bigger blocks than relational database or filesystem (64 MB - 128 MB)
  (-> less seeks
  less interactions with client
  - many operations on same chunk
  - less metadata that can be kept in RAM



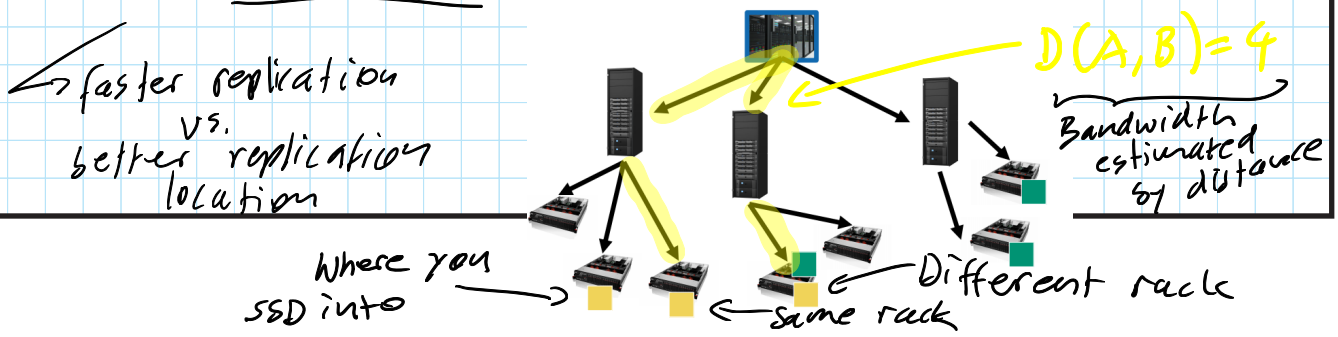


- **Namenode:** - File hierarchy
  - Access control
  - File to block mapping
  - Block locations

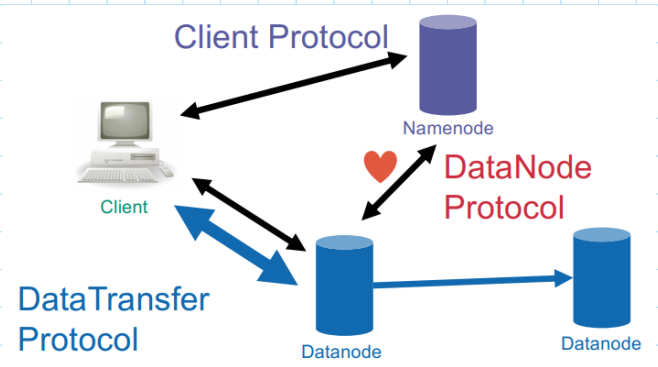


1. Restarting and going throug edit log takes 30 minutes
2. Secondary namenode once-in-a-while backup namenode and play edit-log
3. Standby namenode maintains same information as namenode
4. Specialized namenodes for subdirectories (federated DFS)

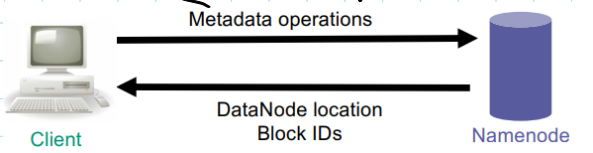
- **Replication:** - 3 replicas per file

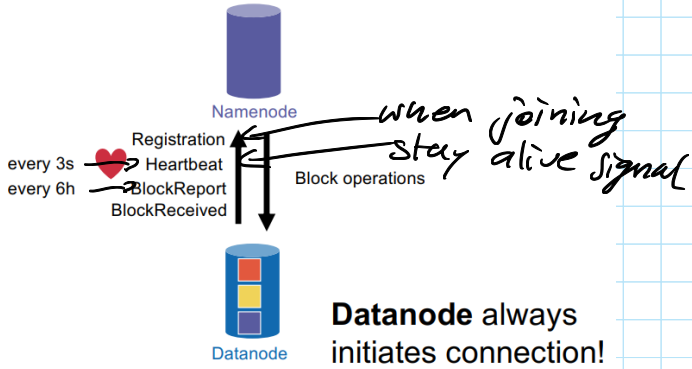


· Communications : -

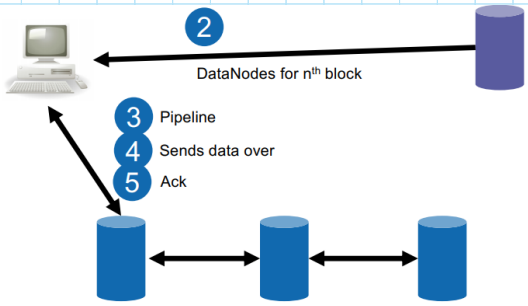


- Create / delete directory
- Read / write file
- Append to file
- Delete file





- Client writes a file



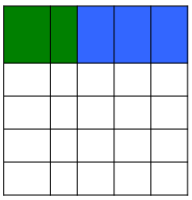
1. Create Metadata to Namenode
2. Datanode IDs for first block
3. Client creates pipeline
4. Client sends data over
5. Ack from all DataNodes
6. Client closes/releases lock to Namenode
7. Namenode checks if min 3 replications are present
8. Namenode Acks to Client
9. Further unsynchronized replicatio

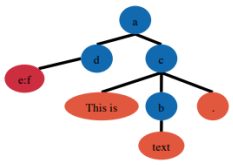
- client reads a file
  1. Client ask for file
  2. Get block locations, multiple dataNodes per block, sorted by distance
  3. Read data block by block

	object storage	block storage
logical model	Key-Value Model	File System
storage capacity	Object Storage Potentially if you have enough commodity machines, object storage can store more data than block storage! **Billions** of <TB files vs. Millions of **<PB** files	Block Storage
limitation on object size	yes, e.g., S3, 5T/object	no, since each object can be split into blocks
partition of files possible	no, only to a single node	yes
implementation (API)	key-value interface with restAPI, get/put/delete	file storage API in java, a file system (hadoop fs –ls /)
use case	1. large objects which do not require file hierarchy, e.g., netflix movies 2. backups which are rarely read	huge files of experimental data (read/write required), e.g., CERN data (LHC data>30PB/year)

Normalization

- Write intensive: normalized due to update anomalies
- Read intensive: denormalized to avoid joins





Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Structured | Semi-structured | Unstructured

SQL | NoSql

- Well-formedness: Is document d part of the language L?

JSON

- Array: [
  3, 1415,
  true,
  "Test",
  {"foo": "bar"}
  ]

- Object: {
  "foo": "bar",
  "bar": false
  }

Keys always need to be strings

↑ No duplicate keys

- { "foo" : "bar", "foo" : "bar2" }

{ [1] : "bar", 2 : "bar2" }

{ "1" : "bar", "2" : "foo" }

↑ always in "..." ↑ when number, no need for "..."

{ foo: "bar", bar: "bar2" }

{ "foo" : "bar", "bar" : "foo" }

sales	
product	orders
varchar(30)	text
Phone	customer (text): John, Peter, Mary; quantity (integer): 1, 2, 1

```
{
  "product" : "Phone",
  "orders" : [
    { "customer" : "John", "quantity" : 1 },
    { "customer" : "Peter", "quantity" : 2 },
    { "customer" : "Mary", "quantity" : 2 }
  ]
}
```

- Object, array, number, "string, true, false, null

↳ ", /, \ needs to be escaped

XML

- Element: <foo> [more XML] </foo>
- Empty tag: <foo/> = <foo></foo>
- Attribute:
- Text: <a> Text

default
- ASCII

```
<?xml version="1.0" encoding="UTF-8"?>
<foo/>
<bar/>
<a foo="bar" foo="bar2"/>

<a foo="bar" bar="foo"/>
```

```
<a>1 < 2</a>

<a>1 &lt; 2</a>
```

```
<person>
  <name first="Alan" last="Turing"/>
  <profession value="computer scientist"/>
  <profession value="mathematician"/>
  <profession value="cryptographer"/>
</person>
```

- ' -> '
- &quote; -> "
- & -> &

Always

- <123/>
- <a<b/>
- <xml/>

- <foo123/>
- <_bar/>

Control characters															
Control characters															
SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Allowed anywhere in name
Allowed but not at start
not allowed

1. Element names are case-sensitive.
2. Element names must start with a letter or underscore.
3. Element names cannot start with the letters xml (or XML, or Xml, etc).
4. Element names can contain letters, digits, hyphens, underscores, and periods.
5. Element names cannot contain spaces.

- XML vs. JSON

XML	JSON
• Doesn't support Arrays	• Doesn't support Namespaces
• Start/End tag	
• Case sensitive	

- CDATA escapes XML syntax
- <!-- test -->

XML Namespaces

```
<m:math xmlns:m="http://www.w3.org/1998/Math/MathML"/>
```

namespace binding

This attribute starts with xml
It is no an attribute like any other.
It obeys a different specification.
It has different semantics.

- Prefix: m (used as proxy to namespace)
- Local name: math
- Namespace: http://... } only this matters

```
<foo:math xmlns:foo="http://www.w3.org/1998/Math/MathML"/>

<m:math xmlns:m="http://www.w3.org/1998/Math/MathML"/>

<ma:math xmlns:ma="http://www.w3.org/1998/Math/MathML"/>

<mathematics:math
     xmlns:mathematics="http://www.w3.org/1998/Math/MathML"/>
<math xmlns="http://www.w3.org/1998/Math/MathML"/>
```

} default schema

```
<foo:bar xmlns:foo="http://example.com/foo">
```

↑ namespace

Scope of the foo namespace binding

prefix
local name

```
</foo:bar>
```

```
<rdf:RDF xmlns:rdf="http://www.w3.org/TR/REC-rdf-syntax#">
  <rdf:Description xmlns:dc="http://purl.org/dc/"
```

```
<?xml version "1.0"?>
<foo:bar
    xmlns:foo="http://example.com/foo"
    xmlns:bar="http://example.com/bar">
  <foo:foo/>
  <bar:foobar>
    <bar:foo/>
    <bar:foo/>
    <bar:foo/>
    <bar:foo/>
  </bar:foobar>
  <foo:foo/>
  <foo:bar/>
  <foo:bar/>
  <foo:foo/>
</foo:bar>
```

Put all namespace bindings in root element

```
<?xml version "1.0"?>
<bar>
  <foo/>
  <foobar>
    <foo/>
  </foobar>
</bar>
```

1. No namespaces at all.

```
<?xml version "1.0"?>
<bar xmlns="http://example.com/a">
  <foo/>
  <foobar>
    <foo/>
  </foobar>
</bar>
```

2. One default namespace.

```
<?xml version "1.0"?>
<a:bar xmlns:a="http://example.com/a">
  <a:foo/>
  <a:foobar>
    <a:foo/>
  </a:foobar>
</a:bar>
```

3. One prefixed namespace.

```
<?xml version "1.0"?>
<a:bar
  xmlns:a="http://example.com/a"
  xmlns:b="http://example.com/b"
  xmlns:c="http://example.com/c"
  xmlns:d="http://example.com/d">
  <b:foo/>
  <c:bar>
    <d:foo/>
    <a:foobar/>
  </c:bar>
</a:bar>
```

4. Many prefixed namespaces.

```xml
<?xml version "1.0"?>
<bar xmlns="http://example.com/foo">
  <foo/>
  <foobar attr="value">
    <foo/>
    <foo/>
    <foo/>
    <foo/>
  </foobar>
  <foo/>
  <bar/>
  <bar/>
  <foo/>
</bar>
```

Unprefixed attributes are in no namespace
even if there is a default namespace in scope

```xml
<?xml version="1.0" encoding="utf-16"?>
<!DOCTYPE movies [
<!ENTITY copy "&#169;">
]>
<movies>
    <Movie id="56225">
        <title>Love Story</title>
        <title></title>
        <year>1980</year>
        <_director name='Coppola'></_director>
        <comment text="Five start"/>
        <comment text="Average"/>
        <newcomment text="An &lt;important&gt;
                          text">Oscar</newcomment>
        <comment lang="de">&copy; 1980 Warner Bros.</comment>
        <!-- Famous movie of the 80s -->
    </Movie>
</movies>
```

Well formed

```xml
<svg xmlns="http://www.w3.org/2000/svg"
      width="12cm" height="10cm">
  <ellipse rx="110" ry="130" />
  <rect x="4cm" y="1cm" width="3cm" height="6cm" />
</svg>
```

Also belong to svg

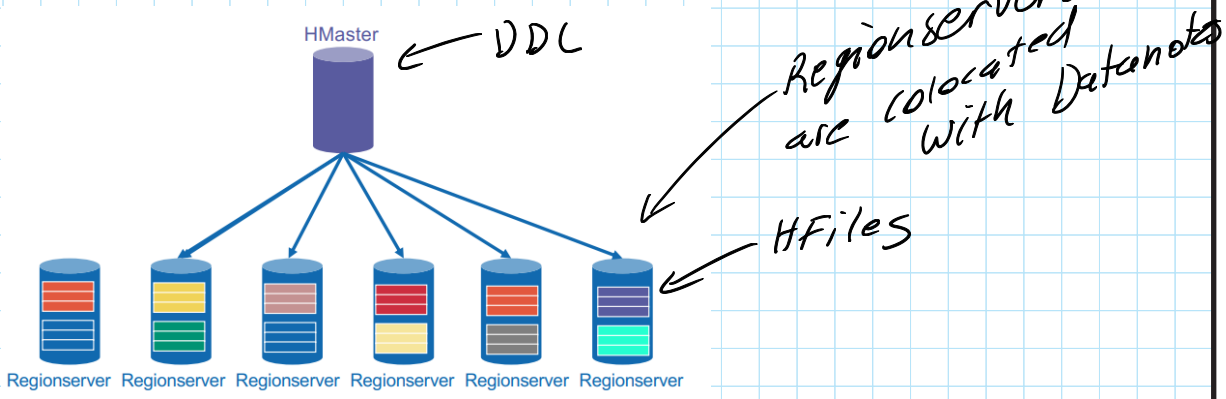
Wide Column Stores

- Similar to key-value model, but we still have the columns and not a black box
- Instead of storing rows together, store columns

Column family

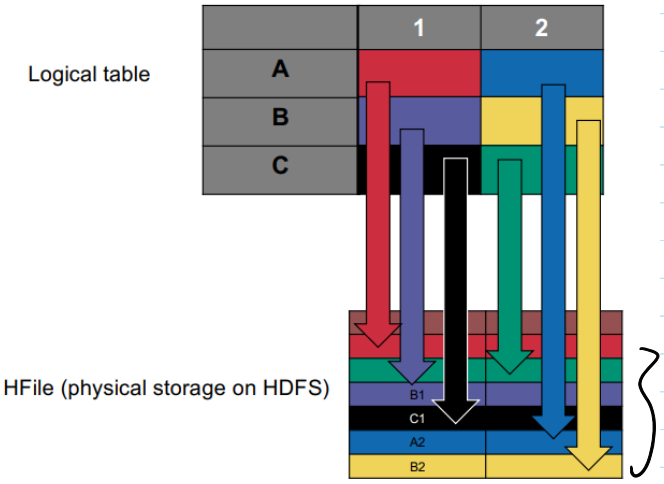
Row ID					
000					
002					
0A1					
1E0					
22A					
4A2					

- Column families are stored on same PC
- Families must be predefined
- But columns can be added later on
- HBase
  - Use HDFS for storage

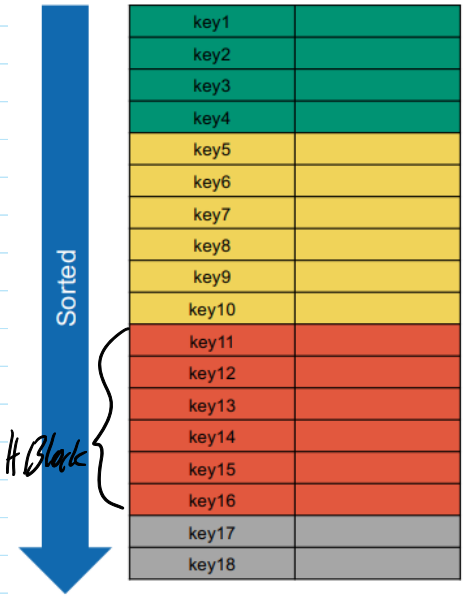


  - HMaster assigns regions to Regionservers with no replication, because this is done by HDFS
  - Column families stored on HFiles, sorted lists of key-value pairs

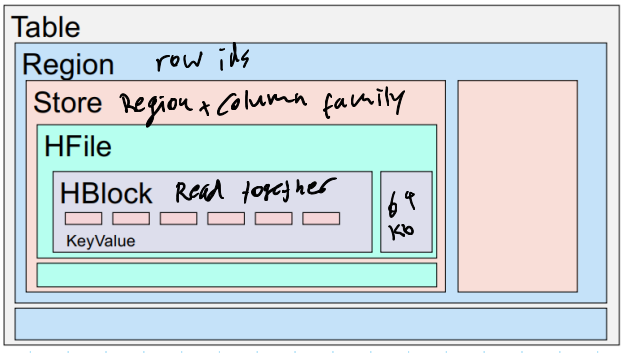
"store"

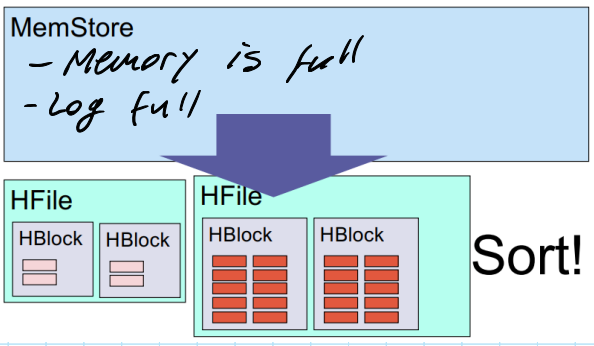


- HBase guarantees ACID on row level
  ↳ Lock whole row for writes/reades



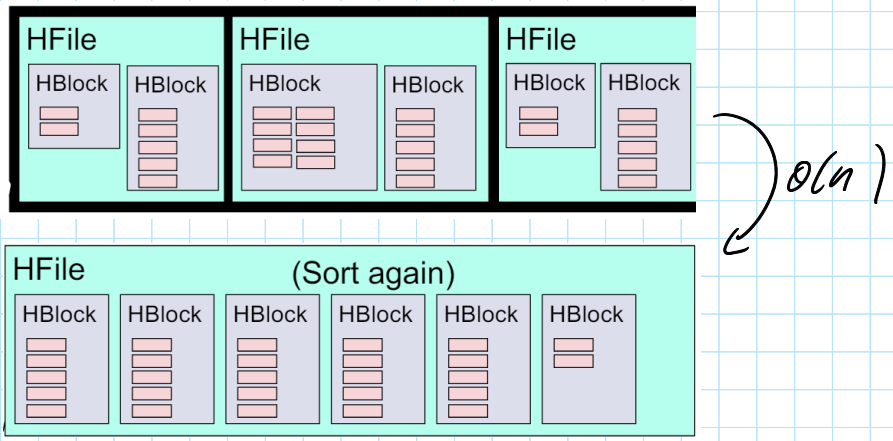
HFile is kept in memory to maintain order when adding new row, when memory is full, flush to HFile



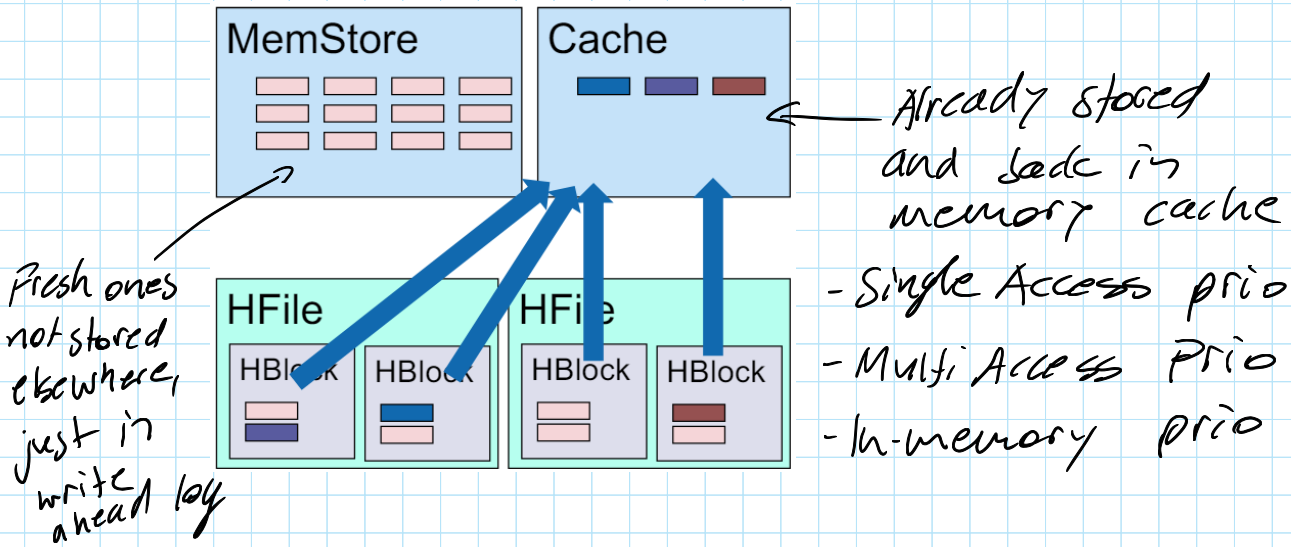


- To prevent corruption from power-loss, there is a HLog

- Reading/Writing from either cache/disk
- Because we need to create a new HFile everytime we flush we need to compact the HFiles together by first sorting all HBlocks and then merging them in O(n)

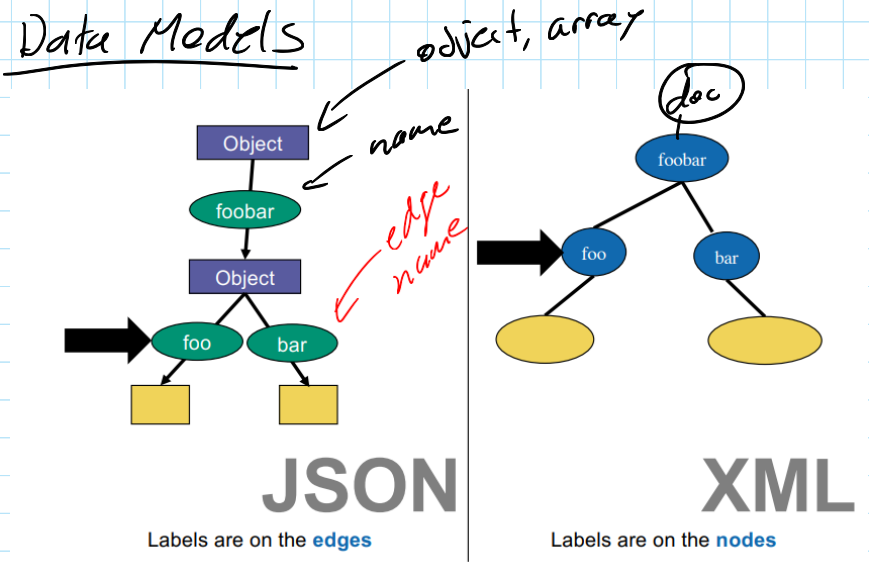


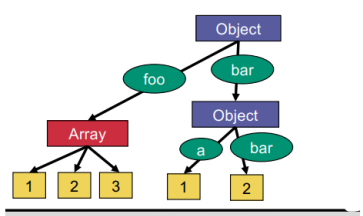
- Needed to be merged in a 2048 type of fashion (Log-structured Merge-Trees (=LSM))
- LRU HBlock cache => Bloom => bucket cache => HDFS
  ↳ except random access or batch processing



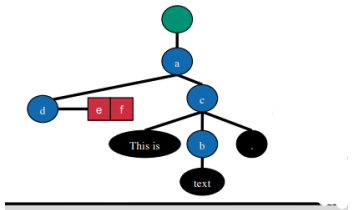
- Row Store: · Good for retrieving small set of rows
  · When scanning over one column, whole row is loaded
- Column Store: · Scanning very efficiently
  · Update a row, lots of accesses
- Wide Column Store: · Middle between Row/Column Store
  · Frequently together accessed columns => column family
  · Sparsity
  · Flexible
  · Storage overhead
- Denormalization: · No joins, only scans/lookups
  · Duplicated data
  · Maintain consistency difficult
- Bloom Filters: · Allow us to quickly determine if key is not in set
  · Speed up queries
  · Before looking into HFile, HBase checks the key against bloom filter associated to that HFile
- Logical View: · Parsed data model
- Physical View: · Syntax e.g. CSV
- Shards: · Plenty of files in same directory
  · Split dataset when uploading in case upload fails

Data Models






```
ew          {
             "foo" : [ 1, 2, 3 ],
             "bar" : { "a" : 1, "bar" : 2 }
            }
```




```
<a>
  <d e="f"/>
  <c>This is <b>text</b>.</c>
</a>
```

- Structured types: maps (object) & lists (array)
- Validation: checked against schema

- <xs:simpleType name="fixed_length">
  <xs:restriction base="xs:string">
  <xs:length value="3" />
  <xs:union memberTypes="xs:Integer xs:Boolean" />
  <xs:list itemType="xs:string" />
  </xs:restriction>
  </xs:simpleType>
- <xs:element name="foo" type="xs:string">
  fixed_length

mixed="true"
↳ <test> Dies <a> b

```
<xs:complexType name="complexContent">
  <xs:sequence>
    <xs:element name="twotofour" type="xs:string" minOccurs="2" maxOccurs="4"/>
    <xs:element name="zeroorone" type="xs:boolean" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>
```

```
<foo>
  <twotofour>foobar</twotofour>
  <twotofour>foobar</twotofour>
  <twotofour>foobar</twotofour>
  <zeroorone>true</zeroorone>
</foo>
```

```
<xs:complexType name="emptyType">
  <xs:sequence/>
</xs:complexType>
```

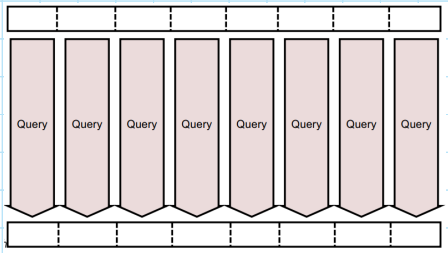
```
<foo/>
```

```
<xs:complexType name="withAttribute">
  <xs:sequence/>
  <xs:attribute name="country"
                type="xs:string"
                default="Switzerland"/>
</xs:complexType>
```

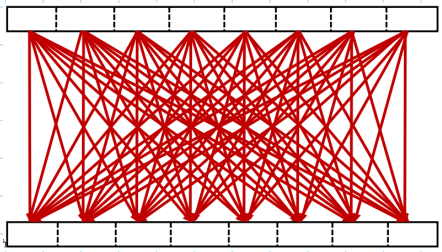
```
<foo country="Switzerland"/>
```

Map Reduce

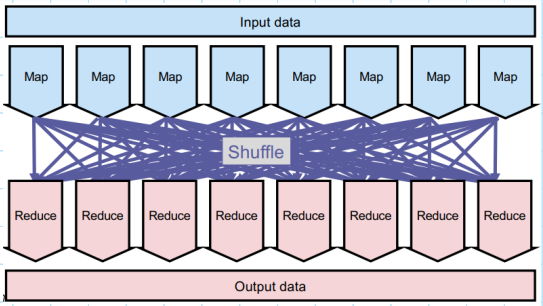
- Best case:



- Worst case



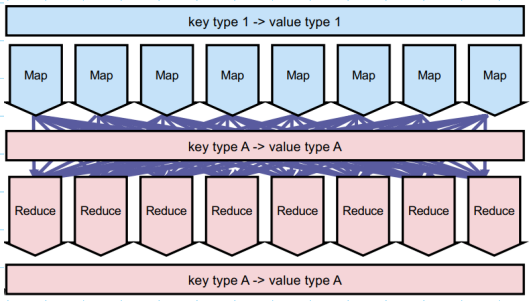
- Map Reduce:



← Key-Value pairs

← Key-value pairs

← Key-value pairs

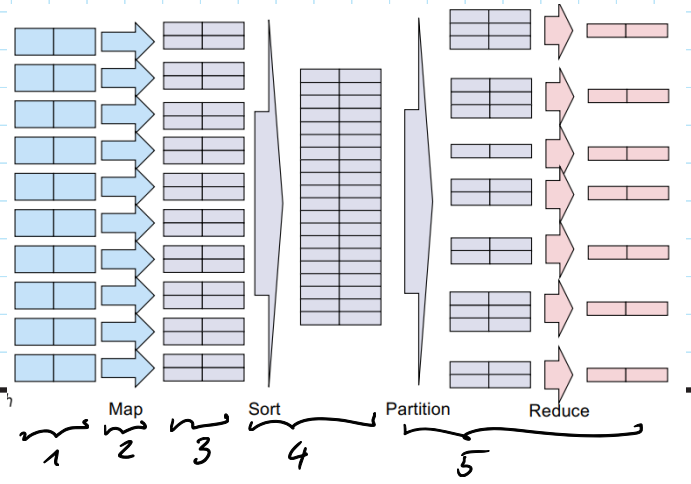


} Elements

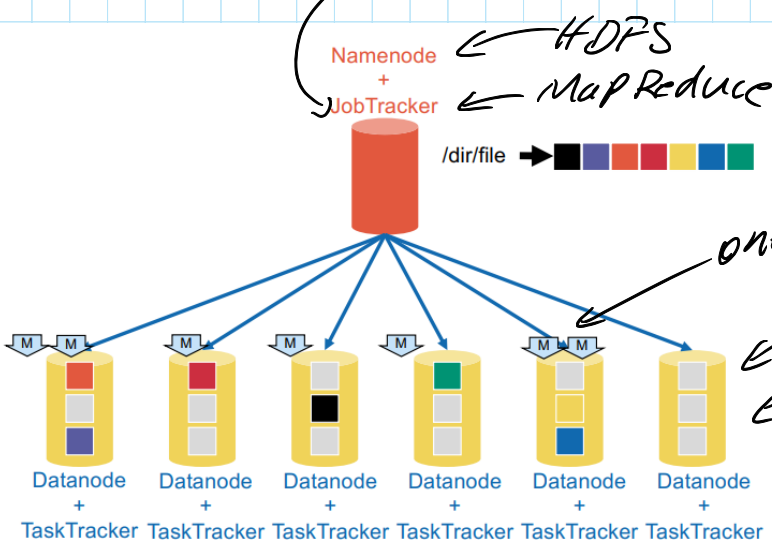
} Number of elements

} Total per element

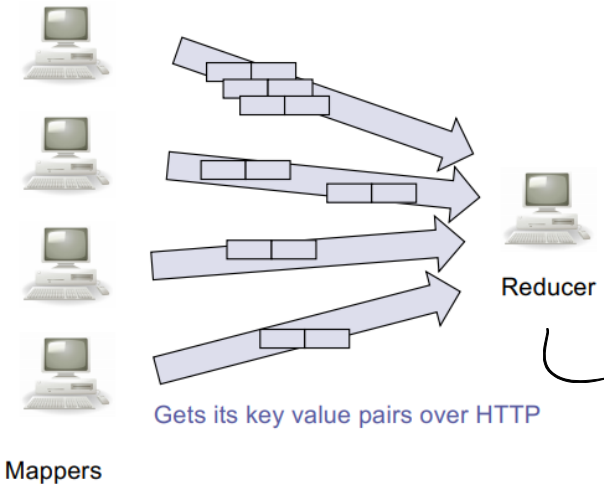
1. Split dataset
2. Map data-split according to a function in parallel
3. Put all mappings together
4. Sort by key
5. Split by key and reduce it to one or more keys



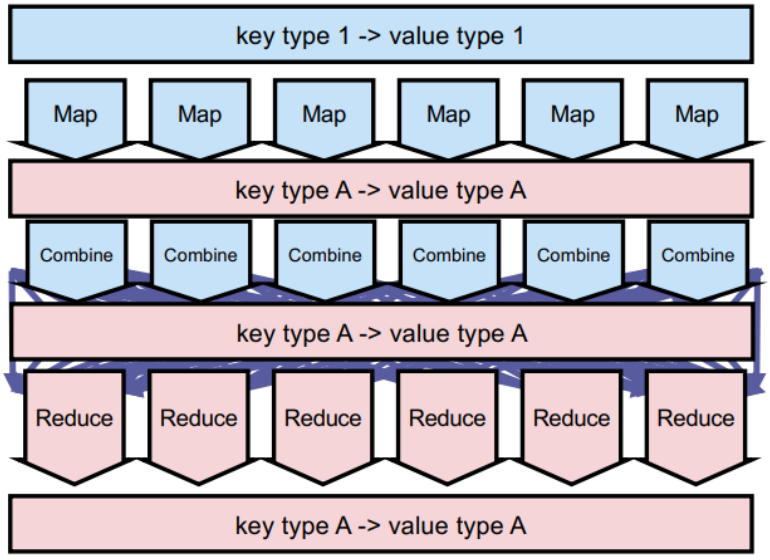
- Resource management
- Scheduling + Monitoring
- Fault-Tolerance



- Key-value pairs are flushed to disk when memory is full and later compacted in a 2048 style of manner



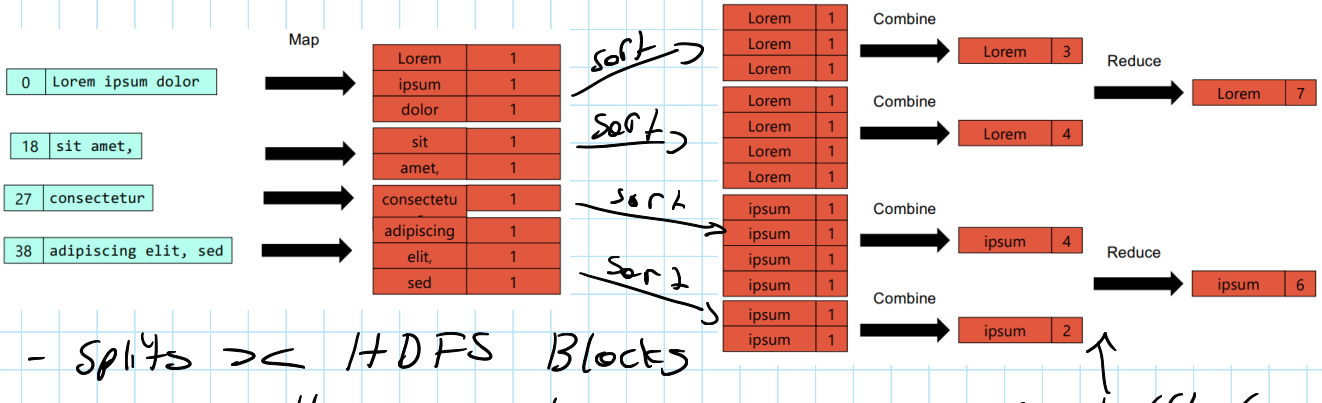
- Mappers are on same TaskTracker than reducer
- # Mappers ≥ # Reducers
  - one shard per reduce task



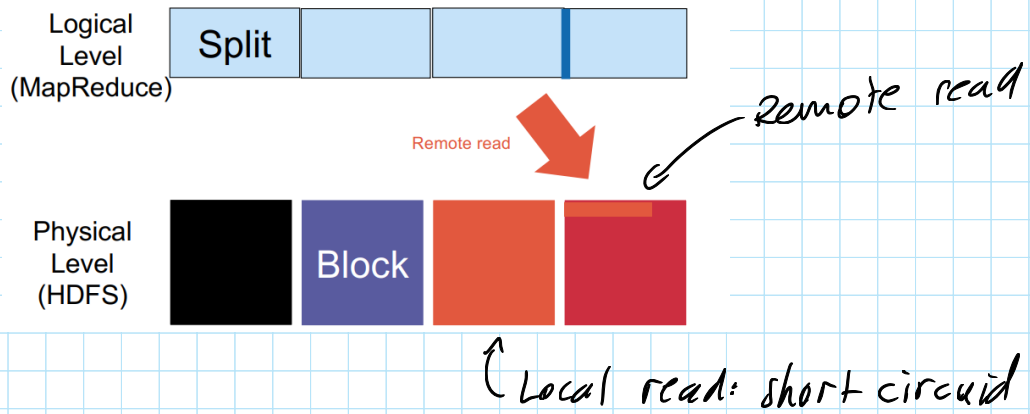
combine when flushing

combine first locally and then shuffle in order to reduce data transmission

- Often, reduce function = combine function
  - Key-value types must be identical for reduce input/output
  - Function must be commutative + associative
    $(a+b)=(b+a)$ $a+(b+c)=(a+b)+c$



- Splits >< HDFS Blocks
  - Normally one split = one HDFS block
  - Splits are in terms of key-value
  - HDFS blocks are in terms of bytes (128 MB)
  - After exceeding 128 MB, the block is splitted, not looking at key-value pairs

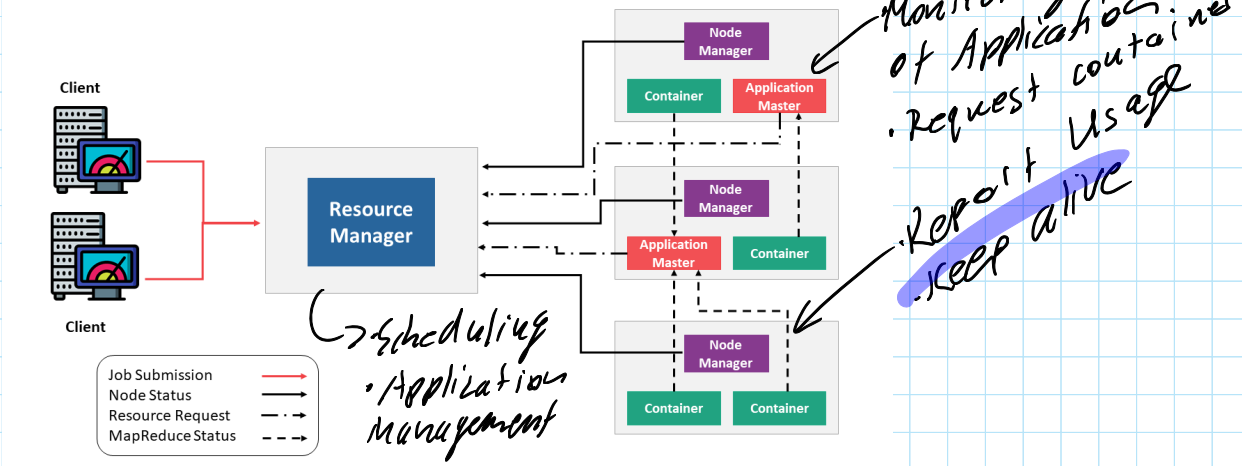


map (k1,v1) --> list(k2,v2)

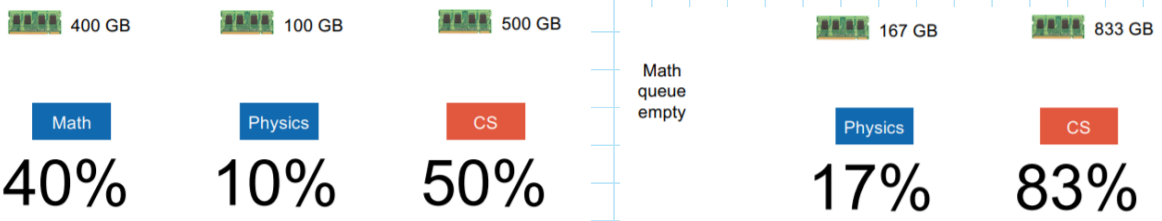
reduce (k2,list(v2))--> list(k2, v2)

YARN (=Yet another resource Negotiator)

- Jobtracker of MapReduce becomes bottleneck as well as only static allocation

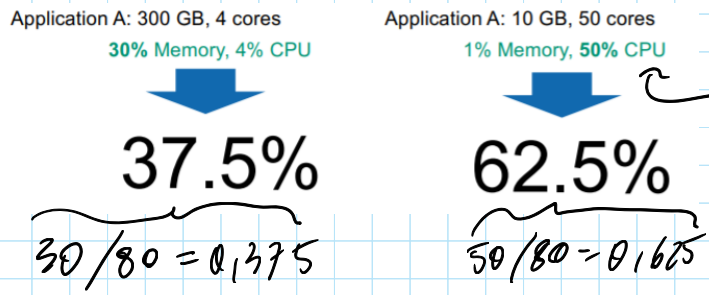


- Resource Manager:
  - Does not monitor/restart tasks
  - FiFo: Disadvantage for small applications
  - Steady fair share: Percentage without looking at current utilization
  - Instantaneous fair share: Percentage by considering an elastic queue
  - Preemption: Claim resources back from elastic queue and get back to steady fair share



40/60 = 0,66 → 33%, 67%

- Dominant Resource Fairness
  - Used for calculation two resources



look at dominant resource

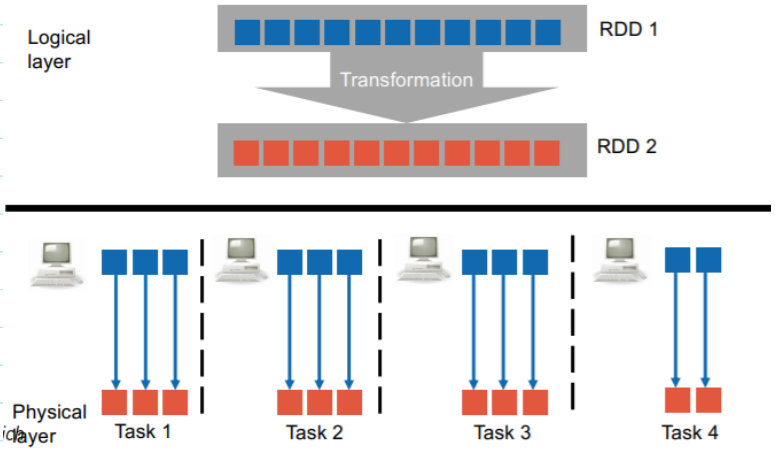
30/80 = 0,375 — 50/80 = 0,625 } Normalization

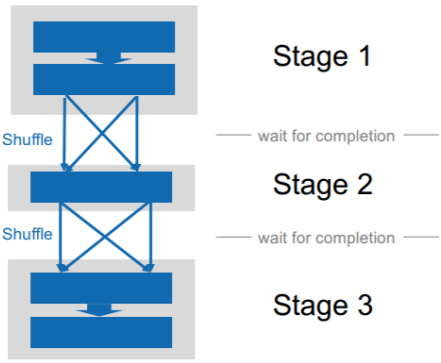
  - Then, we are back to previous problem
  - One resource is underutilized
- Fair Scheduler: Fair distribution, same priority to all
- Capacity Scheduler: Instantaneous fair share
  - Min. guaranteed usage

Spark

- YARN supports any DAG
- RDD: Collection of data which is partitioned (can be recreated when memory is lost)
- Transformations:
  - fruits.filter(lambda fruit: len(fruit) < 5) ("banana" in fruit)
  - fruits.map(lambda fruit: fruit + "test")
  - fruits.flatMap(lambda fruit: [fruit, "test"])
  - fruits.distinct()
  - fruits.map(lambda tuple: tuple[0]) (key)
  - fruits.map(lambda fruit: (fruit[0], fruit)).groupByKey()
  - reduceByKey(lambda v1, v2: v1 + v2)
  - sortByKey
- Transformations on two RDD:
  - fruits.union(yellowThings) => Union on values
  - fruits.intersection(yellowThings) => Intersect on value
  - subtract
  - cartesian
- Actions: (Job is run, when action is called) (one action per Job)
  - fruits.collect() => Outputs to list
  - count()
  - take (first x)
  - top (last x)
  - keys (only take keys)
  - values (only take values)
- dir(fruits) => All available methods
- help(fruits.map) => Description of method

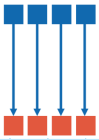
- Execution
  - One task per HDFS block

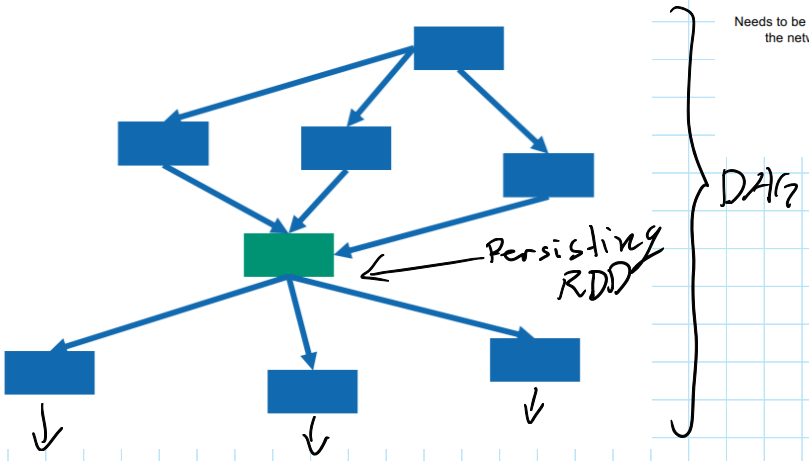


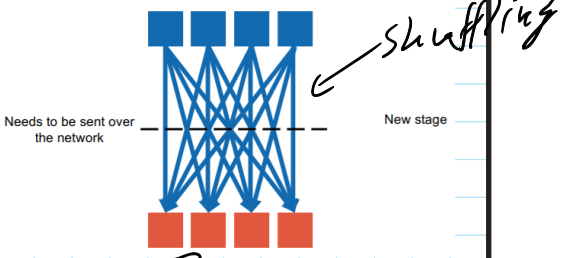


  - Stage: Sequence of parallelizable transformations

- Dependencies
  - Narrow: Better parallelization
  - Wide: Bandwidth utilization







Avoid this, Key-Values on same machine

- Jobs: · Main function that has to be done and is submitted to Spark
- Stages: · Jobs are divided into stages depending on how they can be seperately carried out
  · Mainly on shuffle boundaries
- Tasks: · Stages are divided into tasks
  · Has to be done by the executor
- Executor: · 1 Executor ≙ 15 cores ↯ (overhead)
  · 1 Executor ≙ 1 core ↯ (less memory)
  · 1 Executor ≙ √Total Number Cores
- Spark RDD

```
lines = sc.textFile("data.txt")
lineLengths = lines.map(lambda s: len(s))
totalLength = lineLengths.reduce(lambda a, b: a + b)
```

→ lineLenghts.persist()
cache it

```
lines = sc.textFile("data.txt")
pairs = lines.map(lambda s: (s, 1))
counts = pairs.reduceByKey(lambda a, b: a + b)
```

↳ Values are aggregated

- Pair RDDs: contain key + value
- DataFrame
  - printSchema()
  - select('cust. first_name').show()
  - count()
  - filter(order["name"] == "Jonas").limit(5).show()
  - select(explode("items").alias("i"), "i.product")
    ↑ Generates as many rows as there are elements
  - groupBy().sum()
  - orderBy("sum(total)")

```
temp = orders_df.select(explode("items").alias("i"), "i.product", "i.price", "i.quantity", "order_id")
temp.select((temp["price"] * temp["quantity"]).alias("total"), "order_id").groupBy("order_id").sum().orderBy(desc("sum(total)")).show()
```

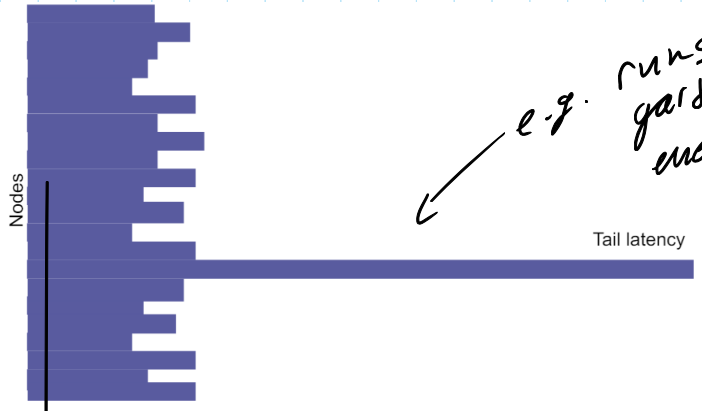
- Spark SQL
  - SELECT orders.customer.last-name ← nested items with dot

```
SELECT order_id, explode(items) AS i, i.product, i.price, i.quantity, i.price*i.quantity AS total
FROM orders
```

```
SELECT i.product, MEAN(i.quantity) AS meanQuantity
FROM (
  SELECT order_id, explode(items) AS i, i.product, i.price, i.quantity, i.price*i.quantity AS total
  FROM orders
)
GROUP BY i.product
ORDER BY meanQuantity DESC
LIMIT 10
```

Performance at large scale

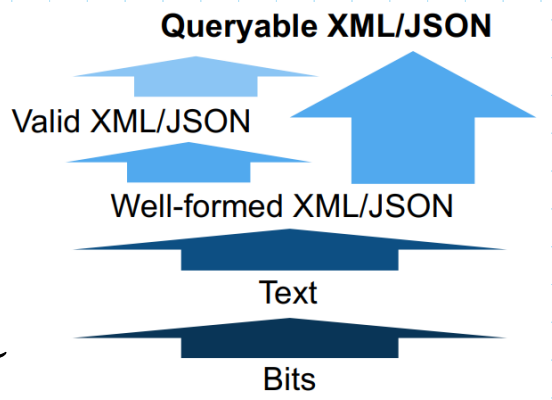
- Bottleneck:
  - Disk
  - Memory
  - CPU
  - Bandwidth
- Total response time: Latency + Transfer Time
- Speedup: $\frac{\text{Latency Old}}{\text{Latency New}}$
- Amadahl's Law: $\frac{1}{1-p+\frac{p}{s}}$ (p ← % parallelizable; s ← speedup of parallelizable part)
  - Constant problem size
- Gustafson's Law: $1-p+sp$
  - Constant computing power
- Improvements:
  - Reduce loops
  - Use efficient formats
  - Use compression
  - If highly structured data, can be represented in binary
  - Before shuffling, prefilter, preproject, preaggregate
- Spark:



e.g. runs garbage collection, energy management, hyperthreading, ...

→ Duplicate task, first task wins
↳ → When I am slower, launch duplicate

Document Store



- NoSQL for trees
- Flat tree: one level
- If there is a schema, I can create table in SQL
- But if there are differencies per row I need to use NoSQL
- NULL when no value available
-
```
{
  "foo":     1,
  "bar":     [ "foo", "bar" ],
  "foobar" : true,
  "a" :      { "foo" : null, "b" : [ 3, 2 ] },
  "b" :      3.14
}
```
} Typical small document

- Document store: billions of CSV/JSON objects
- BSON: Storage syntax of JSON
- MongoDB
  - db.scientists.find({}) (=return everything)
  - db.scientists.find({"Theory": "Relativity"}, {"First-name": 1}) → And -id
    Only include first-name

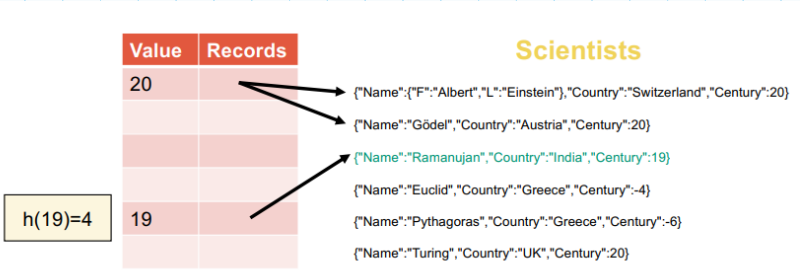
```
db.users.find(
   { age: { $gt: 25, $lte: 50 } }
)
```

  - db.scientists.find ($OR: [{"Last": "Einstein"}, {"Last": "Bokstaller"}])
  - db.scientists.find ({"Age": {$gt: 100}})
  - db.scientists.find ({"Name.Age": 50})
  - [...].sort({"founded": -1}) ↑ asc/desc
- Atomicity: one object

```
db.users.find(                          ← collection
   { age: { $gt: 18 } },                ← query criteria
   { name: 1, address: 1 }              ← projection
).limit(5)                              ← cursor modifier
```

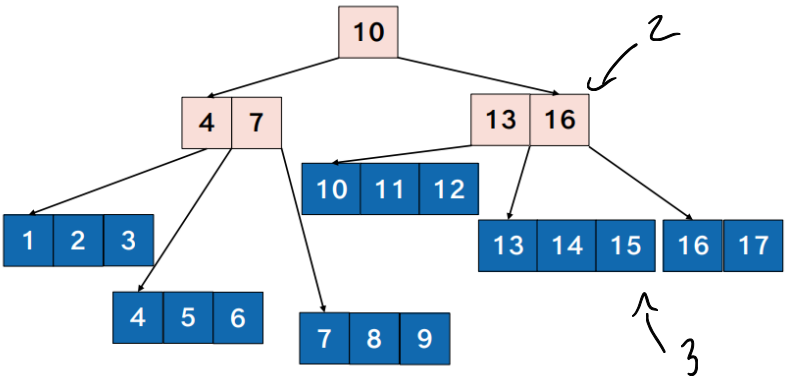
Indices

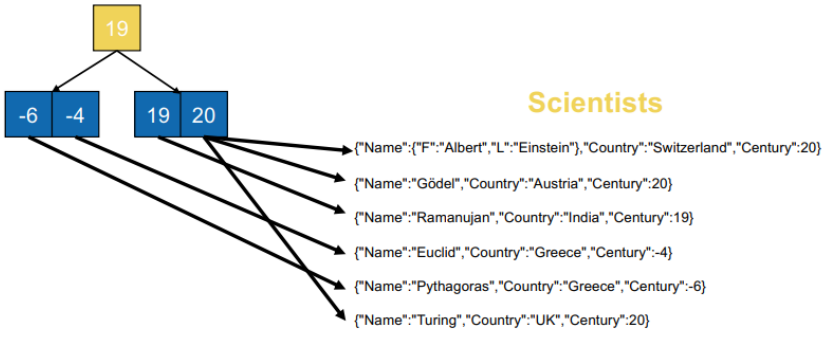
- Hash Indices:



- fastest
- no range queries

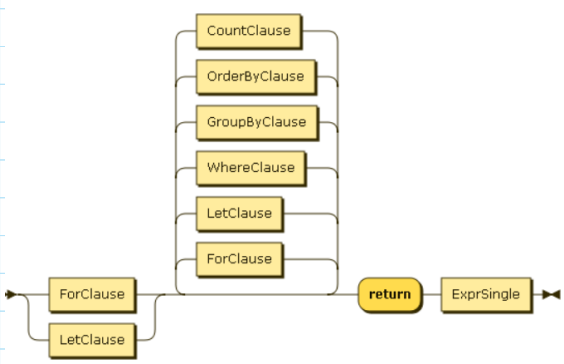
- B+ Tree: · All leaves at same depth
  · 3-5 children





JSONiC

- SQL is good at most one level of nestedness
- JSONiC for highly nested data
- concat()
- substr()
- string-length()
- eq, ne, gt, lt, and, or, not
- if then else
- let $x := 5
  return $x
- for $x in $collection
  where $x.name = "Test"
  group by $x.lastname
  order by $x.name
  return [$x.lastname, count($x),
  [for $l in $x.lastname
  return $l]




```
for $x in collection("captains")
order by $x.name
count $c
return { "id" : $c, "captain" : $x }
```

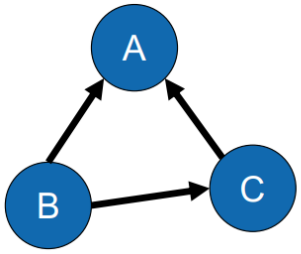
- [] => to sequence
- distinct-values(..)
- join:

```
let $seq1 := ...
let $seq2 := ...
for $i in $seq1, $j in $seq2
where $i.attr1 eq $j.attr2
...
```

- Parsing: Query => Asstract Syntax Tree
  => Expression Tree
  => Iterator Tree

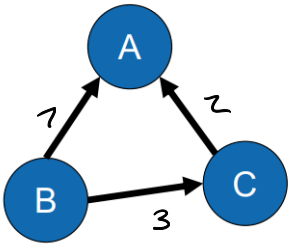
Graph Databases

- Key-value store: Dynamo
- Column Store: HBase
- Object Store: HDFS
- Document Store: MongoDB
- Relations are expensive
  ↳ Graph for direct relations
- one node for each row
- Adjacency matrix

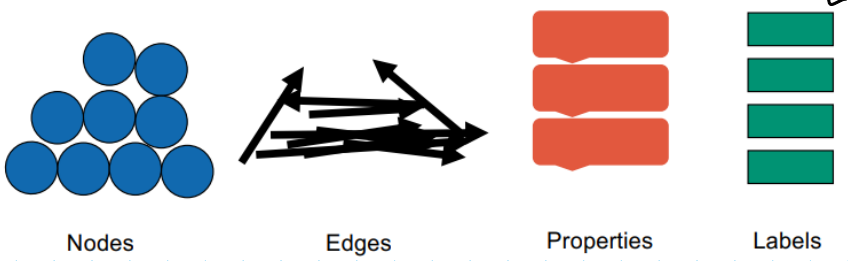


Node	Edges
A	[ ]
B	[A, C]
C	[A]

- Indices matrix:



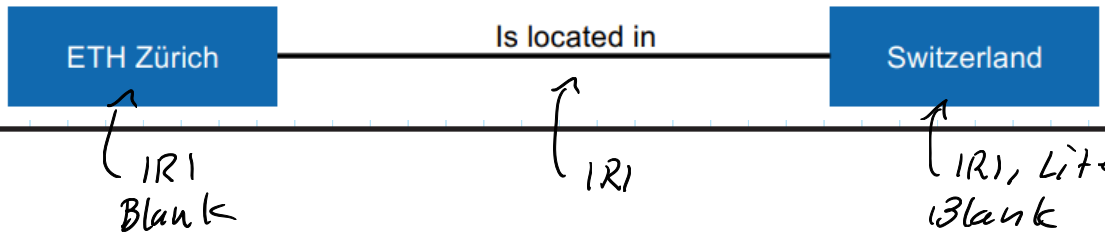
Nodes \ Edges	1	2	3
A	1	1	0
B	-1	0	-1
C	0	-1	1



← What type of node

↑ Like JSON object
Name: Einstein
First name: Albert

- RDF: (triple store)
  - stored as triples: Subject x Property x object



- Can be stored as XML (= RDF/XML)

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:geo="http://www.example.com/geography#">
  <rdf:Description rdf:about="http://www.ethz.ch/#self">
    <geo:isLocatedIn
      rdf:resource="http://www.example.com/Switzerland"/>
    <rdf:type
      rdf:resource="http://www.example.com/geography#school"/>
    <geo:population>25000</geo:population>
  </rdf:Description>
</rdf:RDF>
```

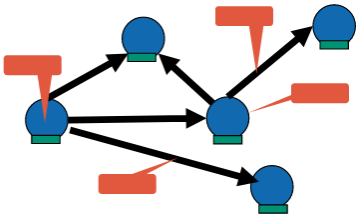
subject

property

subject-Type

object

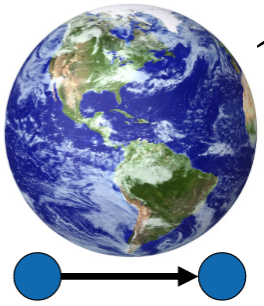
stored as graph



Property Graph

Neo4j

stored in database as triples

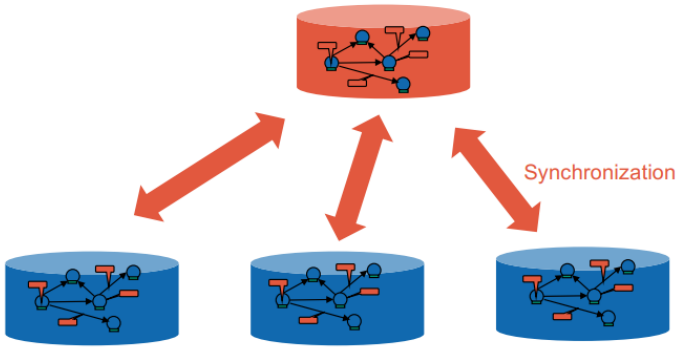


Triple stores (RDF)

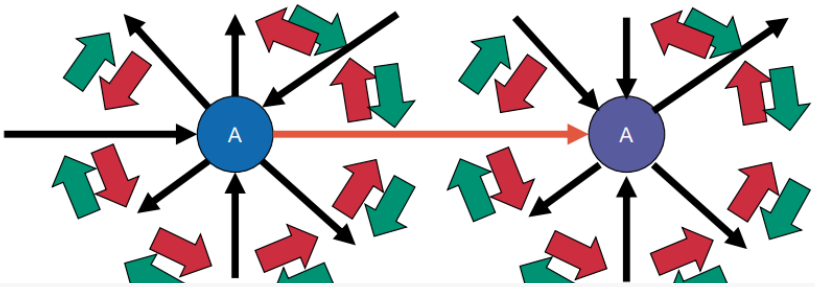
W3C Semantic Web

Neo4j

- No sharding => fast traversal



Allow users to read from replicas




```
MATCH (s:Student),(t:Student)
WHERE (s)-[:IS_FRIEND_OF]-(t) and (s)-[:LIKES]->(t) and not (t)-[:LIKES]->(s)
RETURN s.name, t.name
```

- Cypher is the graph language of Neo4j

name Label property

```
CREATE (einstein:Scientist {name: 'Einstein', first: 'Albert' }),
       (eth:University {name: 'ETH Zurich' }),
       (einstein)-[:VISITED]->(eth)
```

```
MATCH (alpha)-[:A]->(beta)-[:B]->(gamma)
WHERE alpha.name = 'Einstein'
RETURN gamma
```

```
(alpha)
-[*1..4]->(beta)
```

- return DISTINCT gamma.name order by gamma.name gamma.first

```
MATCH p = shortestPath((p1:Person)-[*..15]-(p2:Person))
WHERE p1.name =  "Keanu Reeves"
  AND p2.name = "Tom Cruise"
RETURN p1.name,p2.name,length(p)
```

```
MATCH (s:Student)-[:LIKES]->(m:Movie)
WITH s, count(*) as c  <- Zwischen resultat
RETURN s.name, s.grade, c
ORDER BY c ASC
LIMIT 4
```

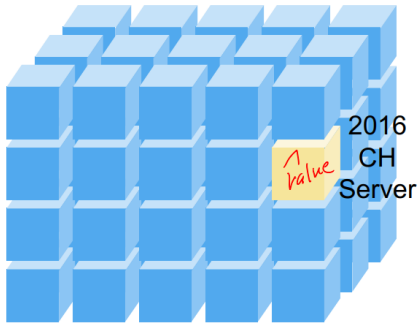

```
MATCH (s:Student),(t:Student)
WHERE (s)-[:IS_FRIEND_OF]-(t) and (s)-[:LIKES]->(t) and not (t)-[:LIKES]->(s)
RETURN s.name, t.name
```

Cubes

- **OLTP:**
  - Online Transaction Processing
  - Daily Database business
  - Normalized => Lots of writes
- **OLAP:**
  - Online Analytical Processing
  - Analysis over big chunks
  - Slow and not interactive
  - Materialized views (denormalized)

Data warehouse:
- Lots of reads
- Integrated: different sources
- Time variant: We save data to year/month
- Non-volatile: No updates to warehouse

- **Data Cube**
  - Each dimension is a variable



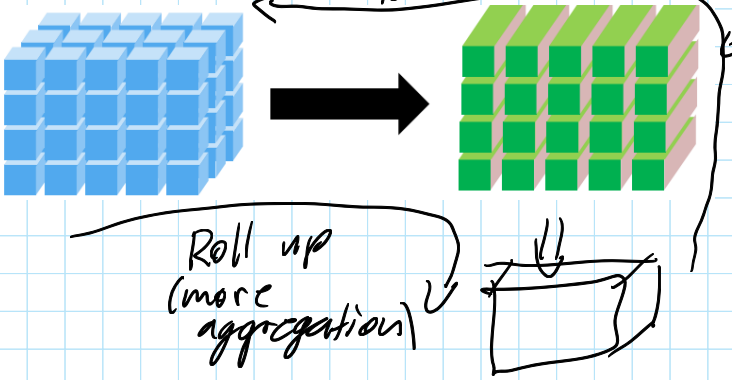
Cubes

Dimensions

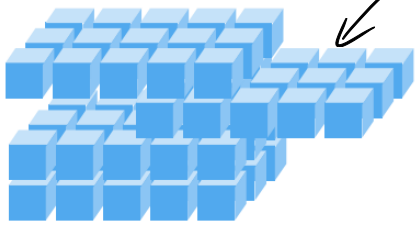
Wh			
Germany	2016	Peter	1,000$
Germany	2015	Mary	15,000$
Switzerland	2016	Mary	1,500$
Switzerland	2015	Peter	3,000$
Australia	2015	Peter	6,000$
China	2015	Mary	1,000$

↳ Fact Table

- **Aggregations**



- Slicer



only Switzerland

- Slicing in cube

≙

selecting in table

```
SELECT t.Year, s.Product, s.Customer
FROM Sales s, Time t
WHERE s.Date = t.Date
  AND t.Year = 2017
```

(Slicing

```
SELECT *
FROM facebook
JOIN linkedin
ON facebook.name = linkedin.name

SELECT *
FROM facebook
JOIN linkedin
WHERE facebook.name = linkedin.name

SELECT *
FROM facebook, linkedin
WHERE facebook.name = linkedin.name
```

- Dicer
  - Slice over two dimensions to make a new block

- Star-Schema
  - Satelite tables are denormalized

```
SELECT t.Year, s.Product, s.Customer, s.Quantity
FROM Sales s, Time t
WHERE s.Date = t.Date
```

- Snow-flake-Schema
  - Satelite tables are normalized
  - One satelite table corresponds to many tables due to normalization

- Union of drill-down >< roll-up

```
SELECT t.Year, p.Brand, SUM(s.Quantity)
FROM Sales s, Time t, Product p

WHERE s.Date = t.Date
  AND s.Product = p.Name

GROUP BY GROUPING SETS (
 (t.Year, p.Brand),
 (t.Year),
 ()
)
```

Year	Brand	Quantity
2017	Apple	6
2017	Samsung	5
2016	Apple	9
2016	Samsung	5
2015	Apple	5
2015	Samsung	4
2017	NULL	11
2016	NULL	14
2015	NULL	9
NULL	NULL	34

} completely drilled down

} completely rolled up

roll up

Year	Brand		
	Apple	Samsung	
2017	5	6	11
2016	9	5	14
2015	5	4	9
Total			**34**

drill down

drill down

} roll up

} Cross-Tabulation

Cheat Sheet

(Q1)
```
SELECT Model, Color, SUM(Value)
FROM CarSales
GROUP BY Model, Color WITH CUBE
```

(Q2)
```
SELECT Model, Color, SUM(Value)
FROM CarSales
GROUP BY Model, Color WITH ROLLUP
```

(Q3)
```
SELECT Model, Color, SUM(Value)
FROM CarSales
GROUP BY Model, Color WITH DRILLDOWN
```

(Q4)
```
(
    SELECT Model, Color, SUM(Value)
    FROM CarSales
    GROUP BY Model, Color
) UNION ALL (
    SELECT NULL AS Model, Color, SUM(Value)
    FROM CarSales
    GROUP BY Color
) UNION ALL (
    SELECT Model, NULL as Color, SUM(Value)
    FROM CarSales
    GROUP BY Model
) UNION ALL (
    SELECT NULL, NULL, SUM(Value)
    FROM CarSales
)
```

(Q5)
```
(
    SELECT Model, Color, SUM(Value)
    FROM CarSales
    GROUP BY Model, Color
) UNION ALL (
    SELECT Model, NULL, SUM(Value)
    FROM CarSales
    GROUP BY Color
) UNION ALL (
    SELECT NULL, NULL, SUM(Value)
    FROM CarSales
)
```

(Q6)
```
SELECT Model, Color, SUM(Value)
FROM CarSales
GROUP BY GROUPING SETS(
    (Model, Color),
    (Model),
    (Color),
    ()
)
```

(Q7)
```
SELECT Model, Color, SUM(Value)
FROM CarSales
GROUP BY GROUPING SETS(
    (Model, Color),
    (Model),
    ()
)
```

(Q8)
```
SELECT Model, Color, SUM(Value)
FROM CarSales
GROUP BY GROUPING SETS(
    (Model, Color),
    ()
)
```

(Q9)
```
SELECT Model, Color, SUM(Value)
FROM CarSales
GROUP BY GROUPING SETS(
    (Model, Color)
)
```

How many?	Common sign	Common adjective
One		required
Zero or more	*	repeated
Zero or one	?	optional
One or more	+	

- < / >
- no xml in tag
- no spaces in tag
- xml -> rs

```
db.restaurants.find({"address.zipcode" : "11225" , "grades.0.grade" : "C",
"grades.1.grade" : "A" },{"name" : 1 })
```

```
db.restaurants.find({"grades.grade" : { $ne : "A"}} , {"name" : 1 ,
"address.street": 1})
```

```
let $ratio := (for $i in $total, $j in $correct
               where $i.target eq $j.target
               return {"language": $i.target, "ratio": ($j.count div $i.count)})
```