Big Data

Tuur Vanhoutte

April 24, 2021

Contents

1 Understanding Data Intensive Applications

1.1 Why Big Data?

1.1.1 Use case: data intensive application RouteYou

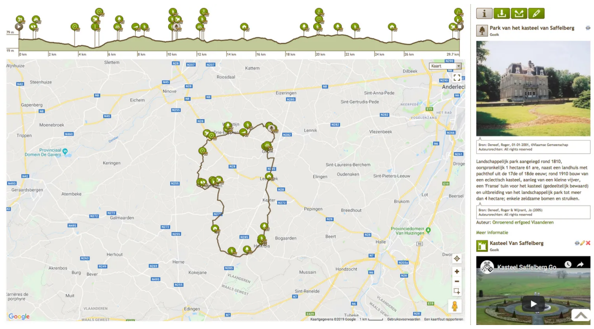


Figure 1: RouteYou

- Routes - user preferences & interests
- Searcheable Text data
- Geospatial data
- Community driven
  - Exponential user growth is necessary to make the application posssible
  - Server power/bills should grow linearly

1.2 Data Intensive Application: RAMS!

- **Reliable**
  - tolerating human mistakes
- **Available**
- **Maintainable**
  - Easy to adapt (evolvability)
  - Easy to deploy & operate (operations/sys admins)
- **Scalable**
  - User growth while maintaining low response times

1.2.1 Common similar abbreviations

- Infrastructure: RAS (Reliable, Available, Serviceable)
- Developer: RMS (Reliable, Maintainable, Scalable)

1.2.2 Methods to improve Maintainability

- Github
- Error handling
- Relative paths (not absolute)
- Abstraction (REST API, . . .)
- Documentation

1.2.3 RAMS applied to RouteYou application

- Geospatial data (longitude, latitude)
- Available & scalable
- Scalable & low response time
- Community driven - unstructured text
- Maintainable: automatic classification of community input (ML)

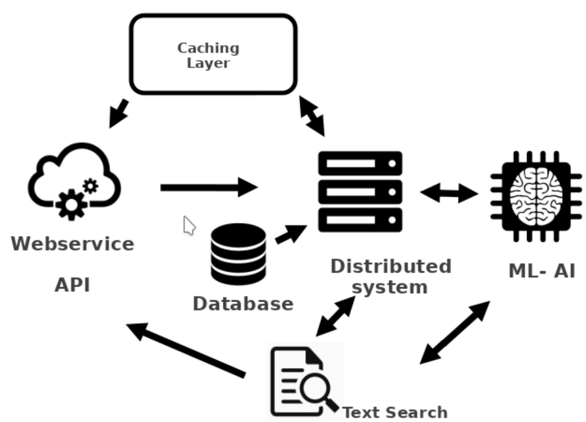


Figure 2: To support many users, you need a caching layer

1.3 Learning outcome for this module

Being able to make infrastructure & software choices to build a Reliable, Available, Maintainable & Scalable (RAMS) data intensive application.

- Deep insights into database technology & cloud services
- Connecting with Machine Learning & AI
- Configuring a data back-end (in the cloud or locally)

1.4 Scaling

1.4.1 MySQL scaling

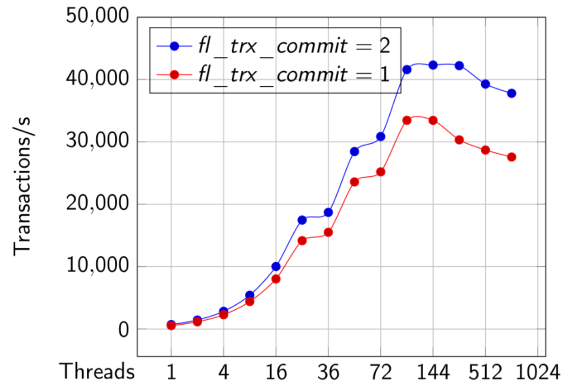


Figure 3: Transactions/sec

- Processing power of 16-64 = slightly less then 4x
- Real performance: 2.3x
- = scaling up: add more processing power to the system

1.4.2 ElasticSearch Scaling: distributed system

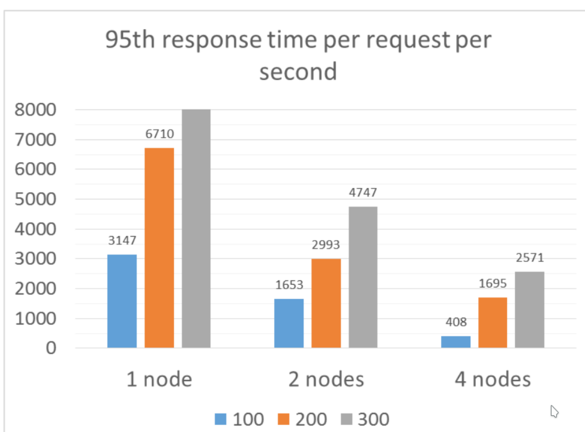


Figure 4: Response time per request

- Scaling out: add more servers to your data system

1.4.3 Professional architecture (Dev oriented)

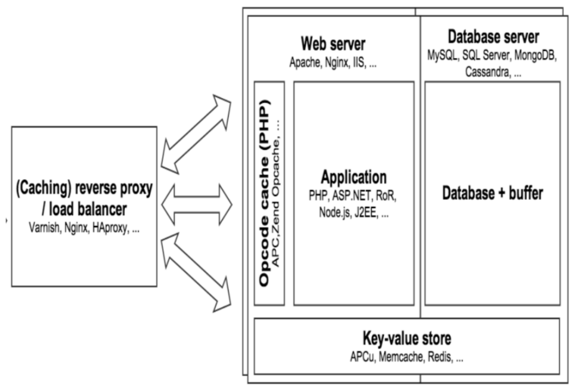


Figure 5: Professional architecture diagram

- **Reverse proxy / Load balancer:** improves scalability
- **Opcode/app/Webserver:** webservice + API
- **Key-value store:** ‘caching layer’
- **Database server:** distributed storage system + relational database

1.4.4 Time series Distributed database (OpenTSDB, InfluxDB)

Location	DateTime	maxTemp	minTemp	prcp	dewpoint	gust	seaLevelPres	windSpeed
3927	2011-09-16T00:00:00	82	80.1	0	75.9		1015.9	5.9
3864	2011-09-16T00:00:00	82.4	71.6	0	70.2			4.2
3860	2011-09-16T00:00:00	86	75	0	68.7	22	1019	12
3858	2011-09-16T00:00:00	84	79	0	71.4		1018.5	4.9
3855	2011-09-16T00:00:00	84.9	78.1	0	70		1019.1	9.8
3853	2011-09-16T00:00:00	82	62.1	0	69.8	25.1	1014.9	12.5
3847	2011-09-16T00:00:00	87.1	78.1	0	56.6		1013	5.3
3846	2011-09-16T00:00:00	93.9	81	0	55.2		1006.4	2.9
3845	2011-09-16T00:00:00	91	78.1	0	62.4			3.3
3843	2011-09-16T00:00:00	59	52	0.05	39.2		1024.1	5.9
3837	2011-09-16T00:00:00	54	42.1	0.07	40.3	20	1017.7	5.3
3835	2011-09-16T00:00:00	57.9	53.1	0	40.9	28	1017.9	13.5
3833	2011-09-16T00:00:00	55.9	52	0.01	38.8	24.1	1019.5	14
3830	2011-09-16T00:00:00	57.2	48.9	0.04	42	20	1016.7	10.1
3827	2011-09-16T00:00:00	63	46.9	0	41.8	22	1010.8	13

Figure 6: Data from windmill sensors. Most sensors log about every second

- Losing data is not that big a problem
- Massive amount of data to write

1.5 Scalability & application performance management

Response times and percentiles rule the web

1.5.1 The need for speed: some insights from Google

- Speed is a ranking factor
- When your site has high response times, less URLs will be crawled from your site
- 53% of visits are abandoned if a site takes longer than 3 seconds to load

- Slow websites will be labeled by Google Chrome

1.5.2 Response times for websites

- **Ideal:** 'blink of an eye' is 300-400 ms
- **Excellent:** 500ms to 1.5 seconds at most
- **Barely acceptable:** 3 seconds

Response time = Network latency + processing

- 2.9 seconds is faster than 50% of the web
- 1.7 seconds is faster than 75% of the web
- 0.8 seconds is faster than 94% of the web

1.5.3 4 components of network latency

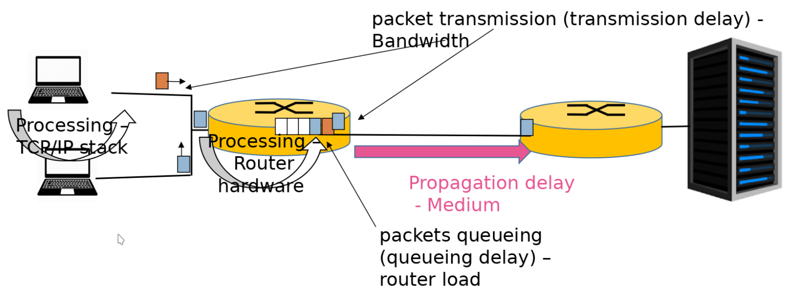


Figure 7: Network latency diagram

- **Processing delay**
  - Processing network software stack (TCP/IP layers)
  - Routing decisions
- **Transmission delay**
  - Bits on physical link (Bandwidth plays a big role, ex: 1Gbit/s)
- **Propagation delay**
  - Speed of EM signals in fiber: 200.000 km/s (67% of lightspeed)
  - Changes with distance and medium (Copper: 64% of lightspeed)
- **Queing delay**
  - Time spent in router & NIC buffers

1.5.4 TCP Congestion Window - slow start

- Network congestion = a network node or link is carrying more data than it can handle
- The internet is built around dropped packages

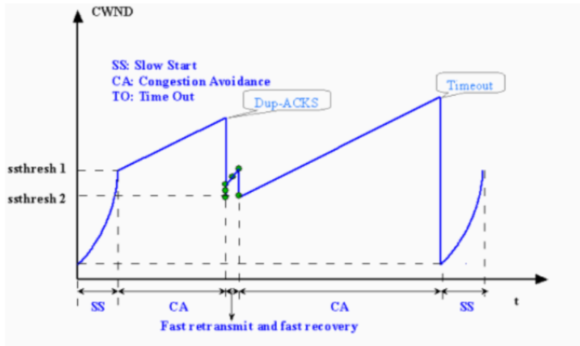


Figure 8: TCP Congestion window

- 4-8-16-32 TCP segments (Win 2008, Win7)
- 10-20-40 (Linux 2.6+, Windows Server 2016 / Windows 10)

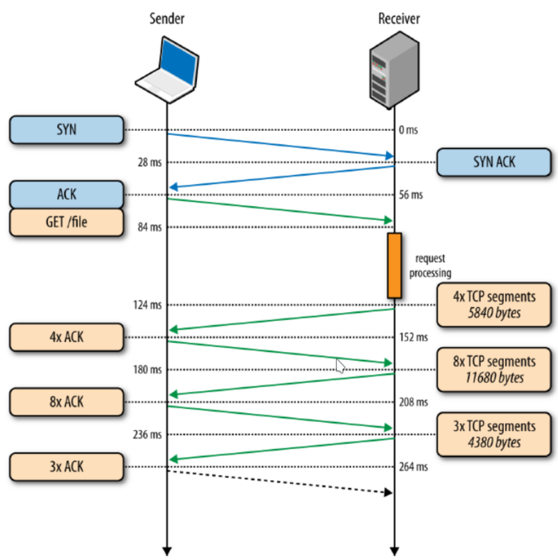


Figure 9: Because of many handshakes, there is a lot of latency

- Solution: KeepAlive of a HTTP Persistent Connection
  - Only one 3-way handshake for many requests
  - Lower network & CPU load
  - Lower response times
  - **Downside**: more connections open $\Rightarrow$ more memory, more connection failures, app crashing, ...
- Measure parallel requests of a website using `https://www.webpagetest.org/`
- Get a waterfall view of a webpage

1.5.5 Long tail latency

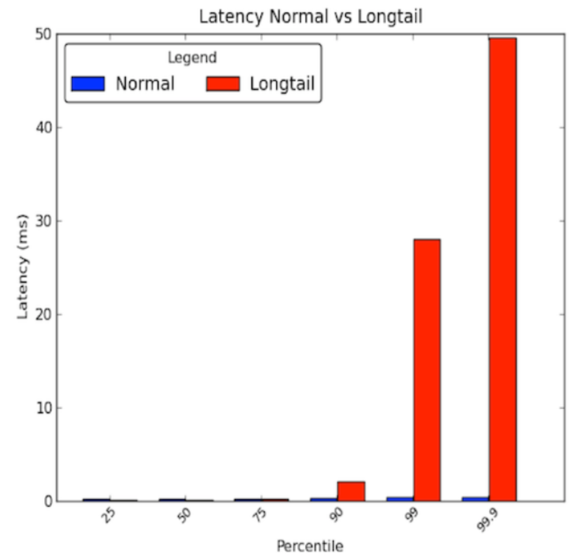


Figure 10: Long tail latency vs Normal latency

- The average latency doesn't show the whole picture
- Long tail latency = 99th percentile highest latency
  - To be experienced by a lot more than 1% of users!
- Best customers encounter highest percentiles
- URL consists of many requests

1.6 Conclusion

- Our goal is RAMS (or RASS)
- Many data models & stores: transactional, timeseries, text search
- Website 99th percentile + DNS + TCP $\Rightarrow$ < 2s response time
  - Efficient caching
  - Think about your architecture (infrastructure + software) before coding

2 Professional storage

2.1 Cloud MIPS

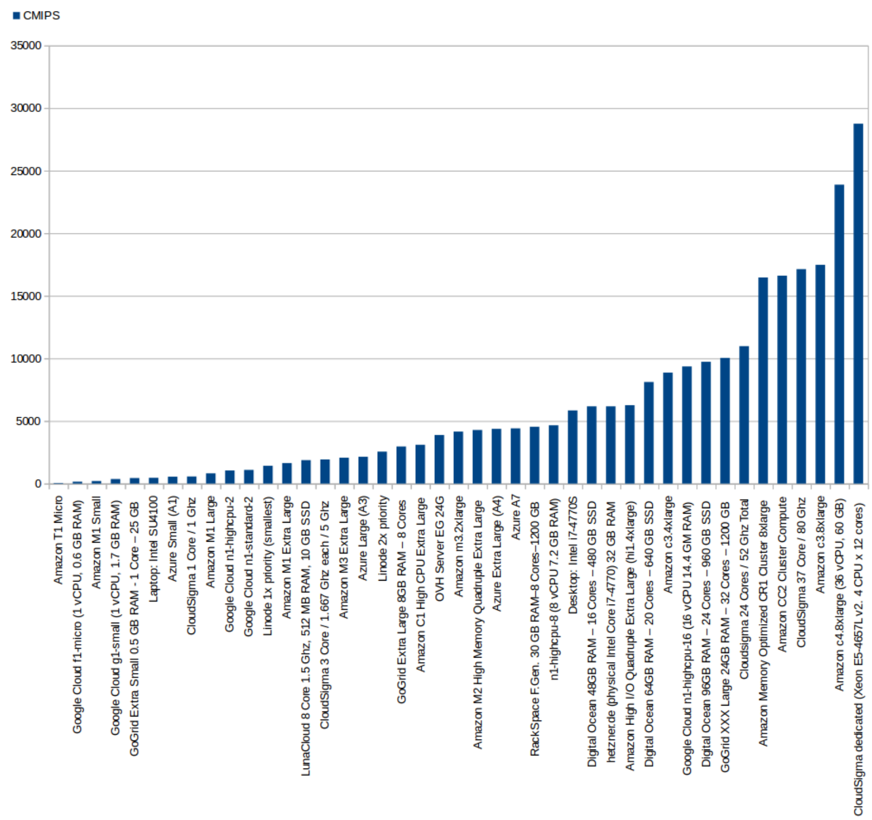


Figure 11: MIPS = Million Instructions Per Second

2.2 Latency vs storage space pyramid

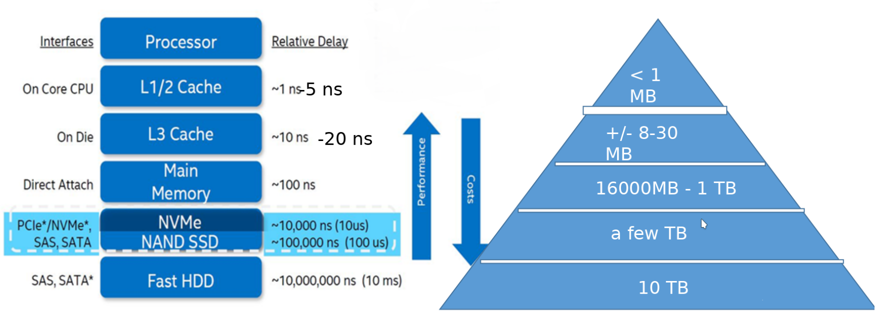


Figure 12: The higher the performance, the higher the cost per byte of storage

2.3 Storage media

2.3.1 Magnetic disks

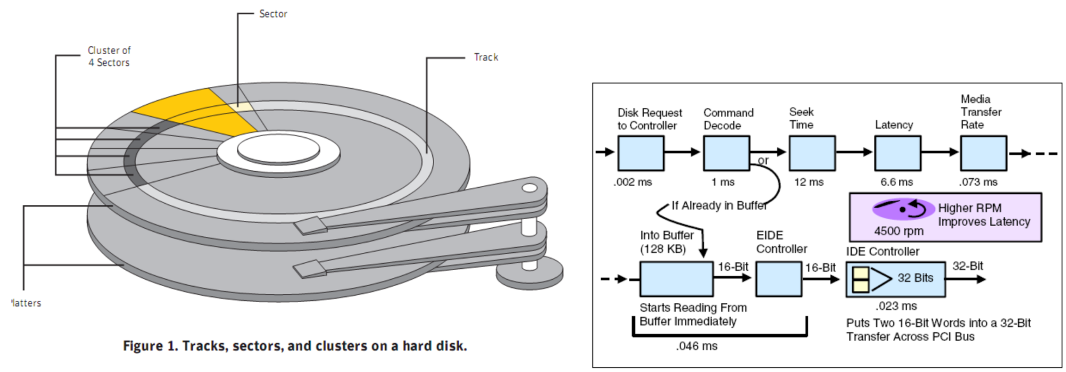


Figure 13: Massive capacity but mechanical latency

- Seek time and latency are the key bottlenecks
- Need large quantity of disks for good server performance

2.3.2 Flash (NAND) / SSDs

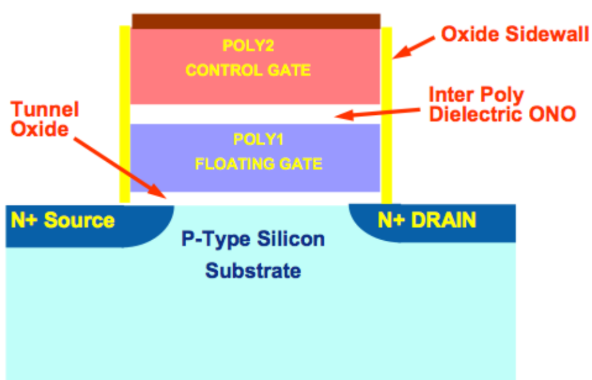


Figure 14: Flash storage

- SSD = Solid State Drive
- NAND = MOSFET + floating gate
- Voltage between control gate and N+ : electrons in floating gate
- This works very quickly

Architecture

- Page = 4 KB, pages are in block
- Block = 128 pages (4KB * 128 = 512 KB)
- You can read or write page per page

- Erasing has to erase the entire block

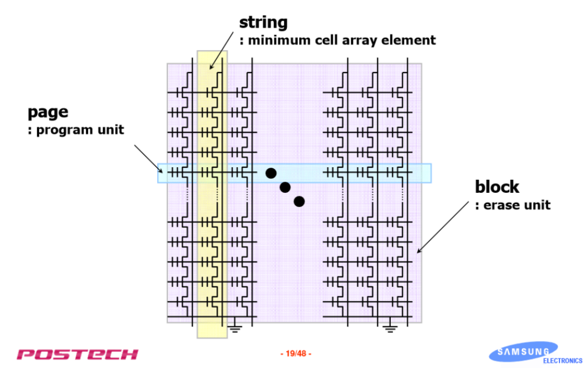


Figure 15: Diagram of a flash Block

2.3.3 Big difference between read and writing

	MLC NAND flash
Random Read (page)	50-100 µs
Erase (block)	1000-2000 µs per block
Programming (page)	40-250 µs

Figure 16: Reading & writing on Flash storage

- Limited number of writes
- Slow block write
- Limited ‘normal’ write (programming)

2.3.4 IOPS vs Bandwidth

- Transactions & virtualized workloads: lots of random access
- Timeseries fileserving: mostly sequential
- HDD: random performance can be extremely low to medium
- IOPS = Input/Output Operations Per Second

Storage device	Seagate Enterprise HDD	Intel SSD NVMe
	ST8000NE0001	DC3700
Capacity	8 TB	800 GB
Spindle speed (rpm)	7200	N/A
Max. BW (MB/s)	**230**	**600**
Latency (ms)	4,16	N/A
Seek time	8	N/A
Total Random read time ms	12	0,08
Random Performance	1000 Random 4 KB blocks	1000 Random 4 KB blocks
Total Random read time (ms)	12000	80
Transfer time (ms)	17,4	6,7
Sustained Transfer rate (MB/s)	**0,33**	**46,15**
IOPS	83	11538
Sequential Performance	1x 4 MB block	1x 4 MB block
Total Random read time (ms)	12	0,08
Transfer time (ms)	17	7
Sustained Transfer rate (MB/s)	**136**	**593**

Figure 17: An enterprise HDD vs an NVME SSD

2.3.5 Storage options

	Media Type	Interface	Read Latency (µs)	Write latency (µs)	Random IOPS	BW (MB/s)
HDD	Magnetic	SATA	10.000	10.000	100	1-200
Low-end SSD	NAND Flash	SATA	100-300+	40-2000+	5k-20k	100-550
High-end SSD	NAND Flash	NVMe	100-200+	20-1000+	50-200k	100-1800
3D-Xpoint	Electric resistance	NVMe	10-40	10-60	500+k	200-2000

Figure 18: Storage options

2.3.6 Performance Conditions

Type	Queue depth	Random?	Write vs Read	Perf consistency
HDD	As low as possible (1-2)	Sequential ! Random as low as 50 IOPS	Write slightly slower	Terrible (1 -200 MB/s)
Low-end SSD	8-16	Random	Write can be a lot slower	IOPS writes can vary 2-4x
High-end SSD	16+	Both	Write can be a lot slower	IOPS writes can vary 10-30 percent
3D-Xpoint	2+	Both	Does not matter	Very good

Figure 19: Performance Conditions

2.4 RAID

Definition 2.1 ***Redundant Array of Inexpensive Disks*** *(RAID) is a storage technology that combines multiple physical drives into one logical unit.*

Purpose:

- *Data redundancy*
- *Performance improvement*
- *Both*

2.4.1 Raid levels

- RAID 0
- RAID 1
- RAID 5
- Combinations are possible (RAID 10, 01, 51, 15)

Level	Benaming	Schijven	Capaciteits verlies	Availability	Lezen sequentieel	Schrijven sequentieel	Lezen random	Schrijven random
RAID 0	Striping	Min. 2	geen	Slechter!	Sneller	Sneller	gelijk	Gelijk
RAID 1	Mirror	Min. 2	50%	Beter	iets Sneller	Gelijk	Sneller	Iets trager
RAID 10	Stripe + Mirror	Min. 4	50%	Beter	Sneller	Sneller	iets Sneller	Gelijk
RAID 01	Mirror + Stripe	Min. 4	50%	Beter	iets Sneller	iets Sneller	Gelijk	Gelijk
RAID 5	Stripe+ Parity	Min. 3	33%	Beter, slechter tijdens rebuild	Sneller	iets Sneller	iets Sneller	Gelijk

Figure 20: RAID level choices

2.4.2 Caching & BBU

- RAM caching: to allow more users to access your data at a time
- RAID = lower latency by caching
- Not always durable: backup solutions needed like Battery Backup Unit (BBU)
- RAID = more bandwitdth, +- same latency
  - Latency does not increase as fast when load increases (vs single disk)
  - More bandwidth & capacity available

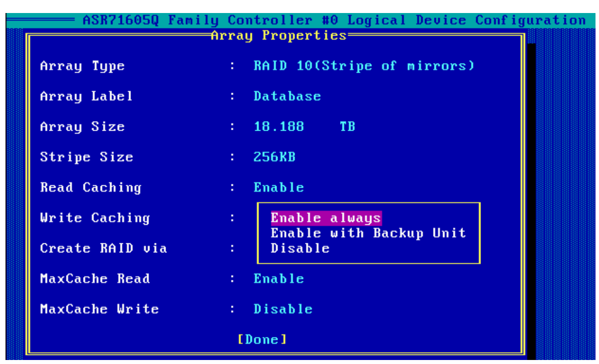


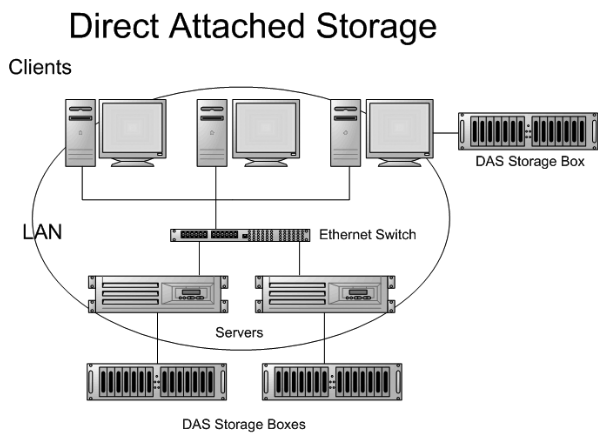
Figure 21: RAID configuration

2.5 Professional Storage Topology

2.5.1 Components

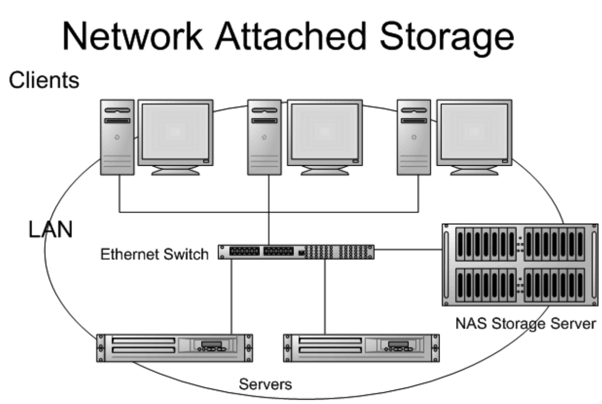
- Enclosure
- Controller
- Disk Array
- HotSpare (=backup disk if a disk fails)
- LUN (logical unit number) / Volumes (= logical storage areas)

2.5.2 DAS - Block storage



- Up to 122 disks per SAS controller (Serial Attached SCSI)
- Similar to disks inside the server
- No centralized back-up

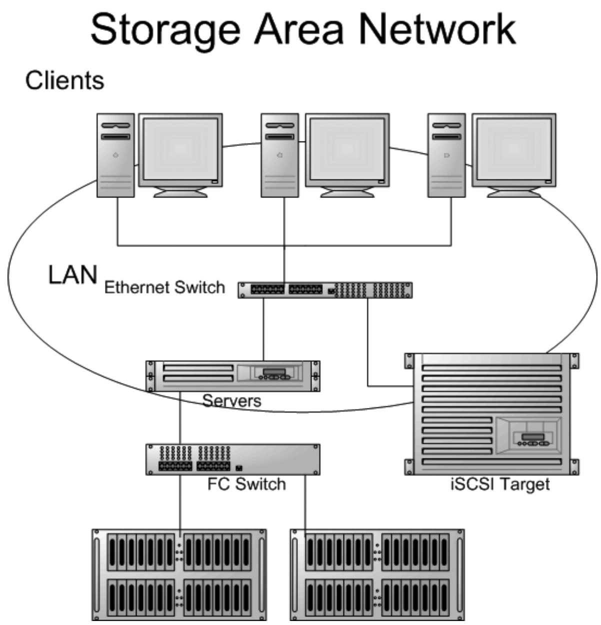
2.5.3 NAS - File storage



- Common Internet File System (CIFS) for Windows ⇒ SMB protocol

- Network File System (NFS) for UNIX ⇒ mounting via network
- SMB also available in Linux

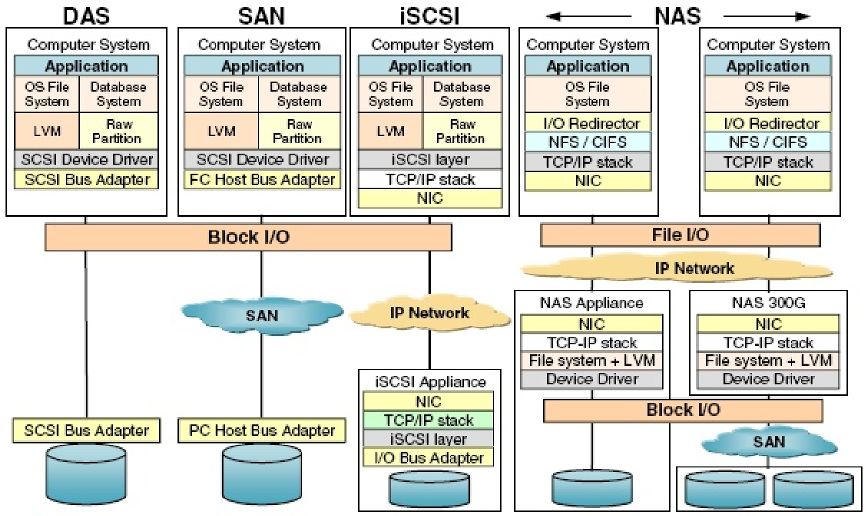
2.5.4 SAN - Block storage on a network



- Seperate Block storage network
- Centralized backup & management
- Good scaling, no load on LAN
- But:
  - No standards - proprietary
  - Expensive

2.5.5 iSCSI terminology

- **iSCSI Target** = the iSCSI 'server'
  - IP + port = Portal
  - Portal: LUNs / Volumes
  - Volume = IQN
- **iSCSI Initiator** = the iSCSI 'client'
  - Connects targets
  - Find LUNs/Volumes



Block I/O compared to file I/O

Storage characteristic	iSCSI SAN	Fibre Channel SAN	NAS
Protocol	Serial SCSI	Fibre Channel Protocol	NFS, CIFS
Network	Ethernet, TCP/IP	Fibre Channel	Ethernet, TCP/IP
Source / target	Server / Device	Server / Device	Client / Server or Server / Server
Transfer	Blocks	Blocks	Files
Storage device connection	Direct on network	Direct on network	I/O bus
Embedded file system	No	No	Yes

Figure 22: Overview iSCSI layers

2.5.6 Object storage

Definition 2.2 *Object storage (or object-based storage) is a data storage architecture that manages data as objects, as opposed to other storage architectures like file systems which manages data as a file hierarchy, and block storage which manages data as blocks within sectors and tracks.*

`https://en.wikipedia.org/wiki/Object_storage`

- Uses NAS hardware
  - But much easier to use for web developers: approachable through URL instead of NAS server IP
  - The hardware is distributed over multiple datacenters
- Object Data
  - For example: images
  - Metadata to support additional capabilities, like better indexing, better data management, . . .
- Objects have a Globally Unique Identifier (GUID)
  - GUID is in URL

- Uses a RESTful API
- Examples of technologies that offer object-based storage:
  - AWS S3
  - Ceph - Lustre
  - Google Cloud storage

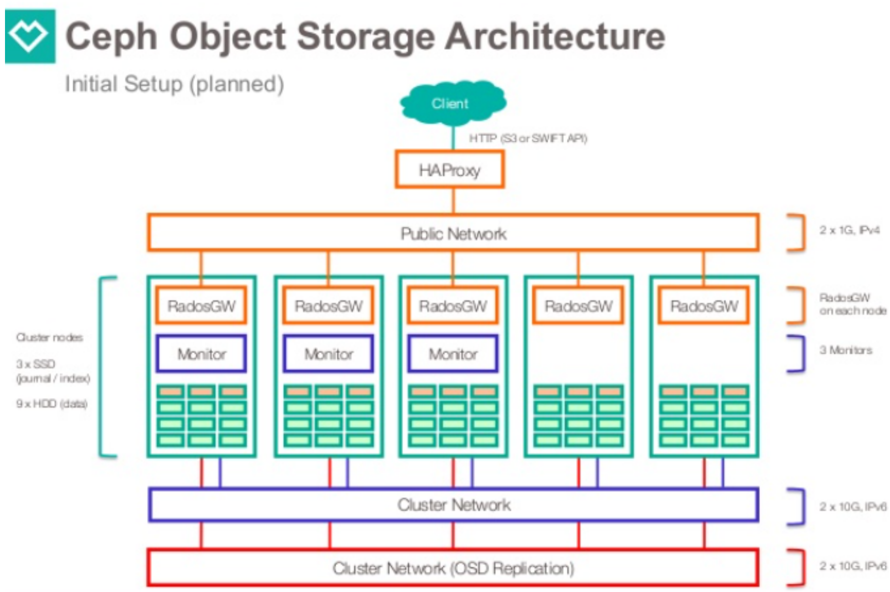


Figure 23: Object storage

2.5.7 Link with Databases & other data storage

- Transactional database: needs block storage
  - Performance
  - Durability
  - Consistency
- Block storage best for ‘raw data’ (no metadata involved)
- NAS = ‘file based’ services like Sharepoint by Microsoft
- Static objects on Object Cloud storage
  - good match for OOP & ‘unstructured data’
  - highly available
  - ‘Eventually’ consistent

3 Relational databases

Data intensive application: needs RAMS!

- **Reliable**
- Available
- Maintainable
- Scalable

3.1 Components of a relational database

- **Tables** = Relations are saved in the format of tables
- **Relationships** = a logical connection between different tables
  - Join, key, foreign key
  - Relation schema
- **Tuple** = A single row (record) of a table, which contains a single unordered record for that relation
  - A dataset representing an object, an item (‘person’)
  - Columns represent the attributes
  - Tuples are unique
  - Tuples are similar to Python dictionaries or JavaScript objects

SNAME	AGE	MAJOR	ID	SEX	ADDRESS	CITY	STATE
Anderson B.	19	CS	55555501	M	101 Rocket Way	Atlantis	CA
Barnes D.	17	MATH	55555502	M	1402 Elf Lane	Ruston	LA
Bronson P.	26	MATH	55555503	M	1 Web Master	Ruston	LA
Brooks D.	18	CS	55555504	F	900 Baird Street	Dallas	TX
Garrett D.	20	PSY	55555505	M	BGB Consulting	Dallas	TX
Howard M.	21	CS	55555506	M	5 Scarborough	Dallas	TX
Huey B.	20	CS	55555507	F	1 Historic Place	Jackson	MS
Kleinpeter J.	24	CS	55555508	M	69 Watson Lane	Ruston	LA
Kyzar D.	18	CS	55555509	M	49 Animae Way	Hammond	LA
Moore D.	19	MATH	55555510	M	No. 7 Seagram	Ruston	LA
Moore L.	20	MATH	55555511	F	2 Pot Place	New York	NY
Morton M.	30	ACCT	55555512	M	2010 Skid Row	Compton	CA
Pittard S.	22	ACCT	55555513	M	111 Easy Street	Ruston	LA
Plock C.	22	MGT	55555514	M	13 NSF Road	Ruston	LA
Slack J.	28	PSY	55555515	M	1 Pirate’s Cove	Ruston	LA
Talton J.	19	PSY	55555516	M	666 Microsoft	Redmond	WA
Teague L.	18	PSY	55555517	F	Fern Gully Farm	Terry	LA
Tucker T.	45	MGT	55555518	F	Prop Wash Way	Eldorado	AR
Walker J.	23	CS	55555519	M	42 Ocean Drive	Venice	CA
Walker R.	21	CS	55555520	M	9 Iron Drive	Monroe	LA

Figure 24: 1 relation ‘student’: 20 tuples, 8 attributes

3.2 Reliability problems

- Applications crash
- Client (website) - network - database
  - $\Rightarrow$ network can be very unreliable
- Multi-threaded code: race conditions $\Rightarrow$ who gets access to 1 piece of data
- Disks can fail

3.3 Example

1 database: bank

- Checking account = table 1
- Savings account = table 2

3.3.1 The problem

```
SELECT saldo FROM checking WHERE customer_id = 10233276;
UPDATE balance SET balance = balance - 200.00 WHERE customer_id = 10233276;

# CRASH: -200 but not on savings account!

UPDATE Savings SET balance = balance + 200.00 WHERE customer_id = 10233276;

# Crash: +200, and application might try again: +400
```

3.3.2 The solution: Transactions

= multiple operations are executed on multiple objects as one unit

```
START TRANSACTION;
SELECT balance FROM checking WHERE customer_id = 10233276;
UPDATE checking SET balance = balance - 200.00 WHERE customer_id = 10233276;
UPDATE savings SET balance = balance + 200.00 WHERE customer_id = 10233276;
COMMIT;
```

VERY IMPORTANT! Every transaction is ACID

- **Atomic**
  - Each transaction is treated as a single 'unit', which either succeeds completely, or fails completely.
  - If all succeed ⇒ Commit transaction
  - If at least one fails ⇒ Rollback transaction
- **Constistent**
  - Data cannot get 'magically' deleted or added
  - Example: when sending money to another bank account, the money cannot exist on both accounts after a transaction
- **Isolated**
  - Transactions cannot interfere with each other
- **Durable**
  - Data is written in a reliable way
  - Storage medium must be reliable

Commit / Rollback does not protect against threads that overwrite each other! It only protects against crashes from one thread.

3.4 Single object entry

Situation:

- Input = 1 record - row - object
  - What if the network fails while sending the input
- Single Object Atomicity & isolation:
  - Create log entry (WAL = Write Ahead Log)
  - Write lock when writing
  - Create log entry if successful
  - Restart if fail
- (Almost) all database - storage engines support this
- This is not a transaction!

3.5 Concurrency Control

Definition 3.1 *Concurrency control ensures that correct results for database operations are generated while getting those results as quickly as possible.*

There are many problems that can occur when executing database operations:

- *Dirty Reads*
- *Dirty Writes*
- *Read skew*
- *Lost updates*
- *Write skew*
- *...*

3.5.1 Dirty Reads

Definition 3.2 (Dirty Reads) *Dirty reads (aka uncommitted dependency) occur when a transaction is allowed to read data that has been modified by another running transaction, and not yet committed.*

Example:

```sql
-- 1: start the transaction
START TRANSACTION;
-- 2: check the current balance
SELECT balance FROM checking WHERE customer_id = 10233276;
-- 3: money is taken from the balance account
UPDATE checking SET balance = balance - 200.00 WHERE customer_id = 10233276;
-- 4: money is put on the savings account
UPDATE savings SET balance = balance + 200.00 WHERE customer_id = 10233276;
-- 5: Commit the transaction
COMMIT;
```

If the data is read after #3 happens, the savings and total values will be wrong

Tx	balance	savings	Total
1	1000	1000	2000
2	1000	1000	2000
3	800	1000	1800
4	800	1200	2000
5	800	1200	2000

Dirty read

Figure 25: Dirty read example: every row Tx is the state after a command

Solutions:

- Read locks (=very bad performance)
- Remember the old value until commit
  - Until the commit happens, every value will be what it was at Tx = 1

3.5.2 Dirty Writes

Definition 3.3 (Dirty Write) *A dirty write happens when a transaction writes data that has been changed on disk by another transaction. The last transaction will overwrite what the first transaction wrote.*

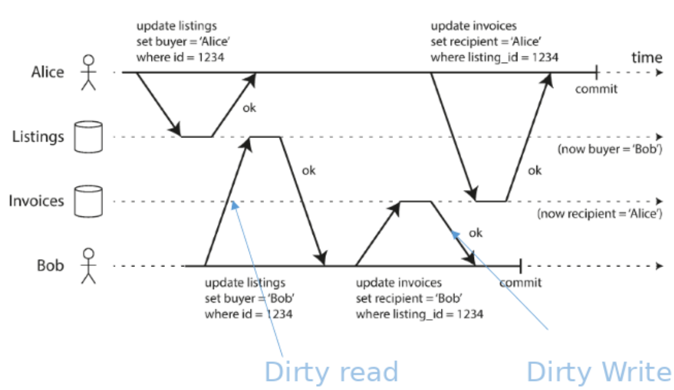


Figure 26: Dirty write example

1. Alice buys a car from a dealership
2. Bob buys the same car from the dealership
3. Bob gets an invoice before Alice because his internet is faster
4. Alice gets an invoice after Bob. Two people now own the same car?

Solution: Write lock:

- If a row is claimed by a transaction, that row should be locked until commit
- Bob cannot write to the invoices, because it has been locked by Alice.

3.5.3 Read skew

Definition 3.4 (Read skew) *Read skew happens when a commit reads the same data twice, with different results because another transaction updated the data.*

1. Alice checks the balance of the first account
2. Bob updates the balance of the first account
3. Bob updates the balance of the second account
4. Alice checks the balance of the second account

Result: Alice ‘loses’ $100 in one commit, because another transaction changed data.

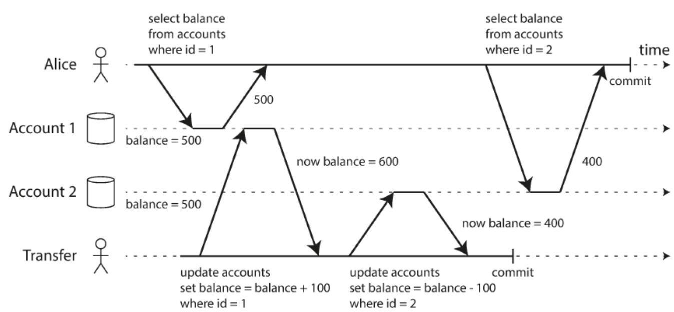


Figure 27: Read skew example

Solution:

- Reading the values again solves the problem
- Except for backups: If a backup saves data while another transaction changes it, you will come across problems.

3.5.4 Lost updates

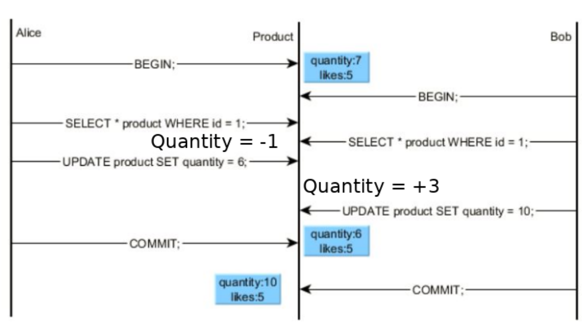


Figure 28: Lost updates: example

1. Alice checks the quantity of the product (quantity = 7)
2. Bob checks the quantity of the product (quantity = 7)
3. Alice buys the product (quantity = 6)
4. Bob thinks the quantity is 7 and he wants to add 3: he sets the quantity to 10 (7+3)
5. Alice commits her changes. According to her, the quantity should be 6
6. Bob commits his changes. The quantity is 10, overwriting Alice's changes. The actual quantity should be 9 (7-1+3)

Solution: Atomic updates

- Problem: Two read - modify - write transactions with different outcomes
- Repeatable read does not fix this
- Solutions: 'atomic updates' or manual lock
  - = Exclusive read lock on the data
  - = No reads or update object until commit
  - Update 'X' SET value = 'X2' $\Rightarrow$ (Read - modify - write in one operation)

3.5.5 Write Skew

Atomic Updates don't protect against everything:

- Multi object updates
- Lost updates

Pattern:

1. Read something
2. Make decision
3. Write new data
4. By the time the write is committed, the premise of the decision (step 2) is no longer true, because some other transaction also changed the data

3.5.6 2-phase lock - Serial execution

With weak isolation levels:

- Readers never block writers
- Writers never block readers (you can read the old value while it is being overwritten)

With 2-phase lock, there are two phases (duh):

1. Exclusive read-lock on data
2. Exclusive write-lock on data

Problem: **Deadlocks**

- Transactions keep waiting on other transactions' locks
- Similar to gridlock: `https://en.wikipedia.org/wiki/Gridlock`

- Result: the whole database can crash because of this

Examples that support 2-phase locking:

- MySQL InnoDB
- SQL server
- DB2 (but DB2 mistakenly calls this ‘Repeatable read’)

3.6 Isolation levels

= Choose between strong isolation or strong performance

- Modern processing 8 - 100+ threads
- Choose an isolation level (sorted from weakest to strongest isolation):
  - Read Uncommitted (weakest isolation, most performance)
  - Read Committed
  - Repeatable Read (=snapshot isolation)
  - Serial Execution (strongest isolation, least performance)
- Isolation problems are hard to debug:
  - It’s a timing problem
  - Very hard to reproduce
  - No errors are logged

	Default	Max	Source
SQL Server	Read Committed	Serializable	https://docs.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql?redirectedfrom=MSDN&view=sql-server-ver15
MySQL InnoDB	Repeatable read	Serializable	https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html
MySQL MyISAM	No Transactions!	No Transactions!	
Oracle	Read Committed	Snapshot Isolation (“Serializable” (*))	https://docs.oracle.com/cd/B14117_01/server.101/b10743/consist.htm#i17856
MongoDB/Cassandra	No Transactions!	No Transactions!	

Figure 29: (default) isolation levels in current databases. (*) Wrong, Oracle does not comply with ANSI

3.6.1 Isolation level 1: Read Uncommited

- Read Uncommitted offers no protection against concurrency threats
- Fastest performance, lowest isolation
- One transaction may see not-yet-committed changes made by other transactions

3.6.2 Isolation level 2: Read Committed

Offers protection against:

- Dirty reads

- Dirty writes

Solutions

1. Read locks (bad performance)
2. Remember the old value until 'commit' (better performance)

3.6.3 Isolation level 3: Repeatable read or Snapshot Isolation

- Also called 'Multi Version Concurrency Control (MVCC)'
- Solves dirty reads, dirty writes and read skew
- If a commit happens before everything is fully backed up, every commit started after the start of the backup will be ignored.
- To accomplish this, every transaction gets a number
- 'Readers do not block writes, writers do not block reads'

3.6.4 Isolation level 4: Serial execution

- One single fast thread (in RAM) for writing
- Multiple threads for reading
- Not very fast, definitely not very scalable
- Only use if your database is not too complex
  - Redis
  - VoltDB
  - Other databases that can be kept in memory...
- No write locks necessary, no overhead from thread synchronisation, ...
- Limited by a single thread on your CPU
- How to use multiple threads?
  - Partition data
  - Multiple threads, one thread per partition
  - Speed will be much slower if a transaction accesses multiple partitions
- Complete transaction in one serial stored procedure (=piece of code, already compiled and ready to execute)

3.6.5 Conclusion

- Isolation levels are a complex trade-off between...:
  - Consistency
  - Scalability
- Check your application: which level is the best for your usecase?
-

3.7 ACID: Durable

= A database should be durable: every write transaction has to be written to disk, and should be stored safely and reliably.

But durability is also a trade-off:

- Higher durability ⇔ lower performance
- Higher performance ⇒ more risks

3.7.1 Caching & BBU

- If we choose the highest isolation on sofware, the hardware can still fail.
- For a RAID configuration:
  - Most RAID configurations use a RAM cache before writing changes to disks
  - RAM cache should have a Battery Backup Unit (BBU) in case the power goes out
  - This is because RAM is by definition not durable, but volatile
- Disks also have RAM caches
  - This is mostly to sort the data before it gets written
  - Use professional storage: disks with capacitors!
  - RAM caches in SSDs have to be Non-Volatile (NV)!

3.7.2 The Transaction chain

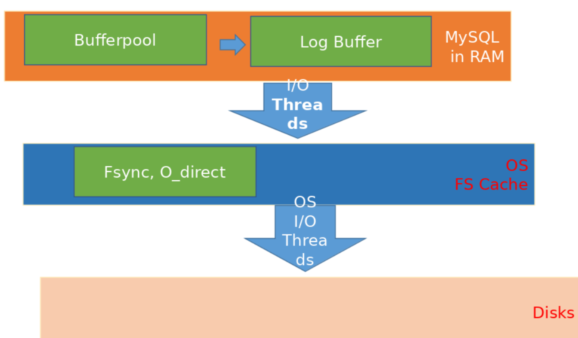


Figure 30: The transaction chain when writing to disk

Steps a transaction takes to write to disk:

1. The transaction gets buffered in the buffer pool
2. The data gets written to the log buffer
3. Using multiple I/O threads, the log buffer gets flushed to the OS
4. The OS chooses how the data is written (cache first, or write immediately)
5. The OS writes the data to the disks

3.7.3 The transaction chain: innodb_flush_log_at_trx_commit

= a setting in MySQL InnoDB with three options:

- 0: Write the log buffer to the log file and flush the log file **every second**, but do nothing at transaction commits (fastest)
  - Fastest
- 1: Write the log buffer to the log file and flush it to durable storage **at transaction commits**
  - This is the only option that is fully ACID compliant
  - It is also the slowest
- 2: Write the log buffer to the log file **at every commit**, but flush it every second

3.7.4 innodb_flush_method

= a setting that tells the OS how the data has to be written

- fdatasync
  - InnoDB uses fsync() to flush both data and log files (unix)
- O_DIRECT
  - This setting still uses fsync() to flush the files to disk, but it instructs the operating system not to cache the data and not to use read-ahead. Avoids double buffering
- async_unbuffered
  - Default value on Windows
  - Causes InnoDB to use unbuffered I/O for most writes
  - Exception: it uses buffered I/O to the log files when innodb_flush_log_at_try_commit = 2

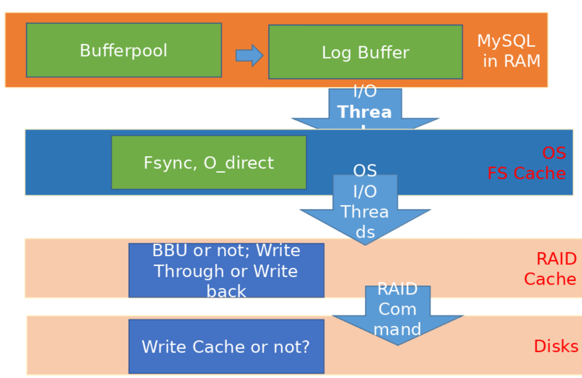


Figure 31: The full transaction chain, with RAID

4 NoSQL

4.1 SQL

4.1.1 Possibilities:

- Relational
- Column store
- Document store
- Graph
- Key-value
- Special:
  - Time series
  - Text search

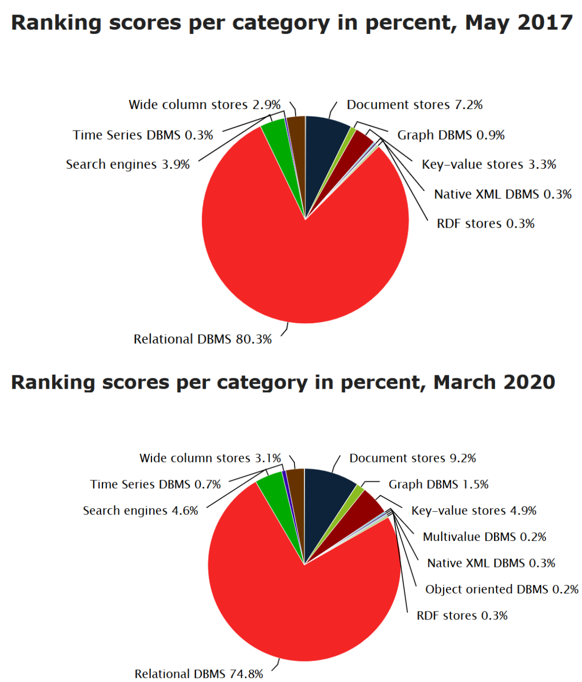


Figure 32: Popularity: Relational DBs are the most popular

4.1.2 Imperative languages vs Declarative languages

```
Public class TokenizerMapper extends Mapper<Object, Text, Tekst>{
    private final static IntWriteable one = new IntWriteable(1);

    public void map(Object key, Text alue, Context context) throws
                IOException, InterruptedException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while(itr.hasMoreTokens()){
            word.set(itr.nextToken()); context.write(word, one); }}}
        .... Much more code
```

```
SELECT word, count(*) FROM lines
    LATERAL VIEW explode(split(text, ' ') Table as
words
    GROUP BY word;
```

Figure 33: Imperative (left) vs Declarative languages (right)

- Imperative: tell the system how to retrieve/handle/mutate the data, in what order
  - C#, Java, python, ...
- Tell the system the structure of the data you're looking for. Don't tell the system how it has to happen.
  - SQL (uses the query optimizer), HTML + CSS (the browser figures it out)

4.2 B-tree index

4.2.1 Index

Index of a book:

- Summary/copy that allows to search faster in the main structure (book/database)
- Redundant (copy), needs disk space (needs 'pages')

Index of a database:

Definition 4.1 (Index) *A copy of some columns from a table, sorted, that improves the speed of data retrievel operations at the cost of additional writes and storage space. Indexes are used to quickly locate data without having to search every row in the database.*

- `https://en.wikipedia.org/wiki/Database_index`
- `https://use-the-index-luke.com/sql/anatomy`

4.2.2 B-tree index

Definition 4.2 *A B-tree index stands for 'balanced tree index' and is a type of index that can be created in relational databases. It works by creating a series of nodes in a hierarchy. The nodes cover a range of values of row IDs. It's called a tree because it has a root, several branches, and many leaves.*

The leaves make up a doubly linked list: *https://en.wikipedia.org/wiki/Doubly_linked_list*

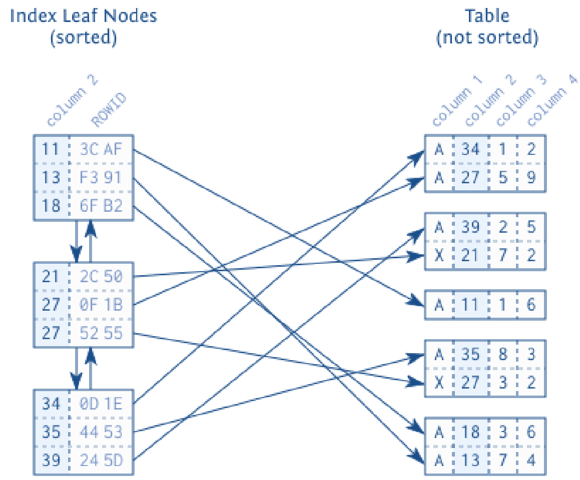


Figure 34

- The leaf nodes make up a doubly linked list
  - Every block refers to other blocks: the next and previous block
  - Every block has key-value items: Index + Row ID (or index + value in 'key value' DBs)
  - Insert = add new links to the list
- This index is stored in RAM:
  - If you have to jump from one block to another block, sequential disks are too slow for random access: random read/writes are much faster with RAM
  - If you have 10 million records: serial search is way too slow!

4.2.3 Tree architecture

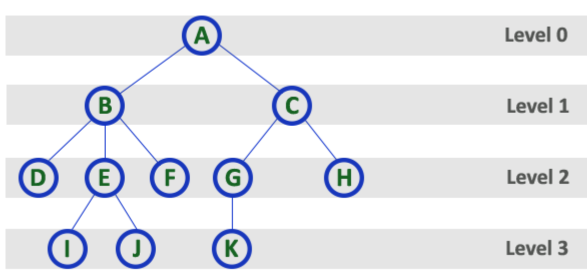


Figure 35: A balanced tree

- A = root
- B & C = child (pages)
- AC = edge
- Depth A to K = 3 (=amount of edges)
  - From root to leaf

- I, J, K = leaf nodes (point to data)

4.2.4 Searching for an index

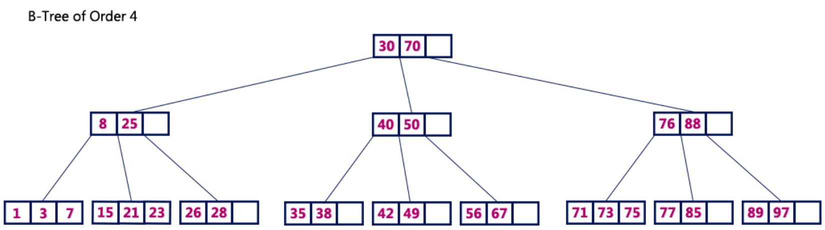


Figure 36: Balanced tree example: how do you find index 38?

1. Start at the root node and search from node 30
2. Go to the next level and find node 40
3. Search from node 35 (depth 3), now read serially
4. Find row id in index 38

4.2.5 Size

- Block size = 16KB
- Branching factor = 100 (=number of children for each node)
- Depth = 3
- $100 \cdot 100 \cdot 100 \cdot 16$KB $= 16$GB
  - Typical depth: 4
  - Typical branches = 100s

4.2.6 B-trees: getting faster & more reliable

Definition 4.3 (WAL) *A Write Ahead Log is a type of file (you can also call it a type of sequential database) where database changes are first recorded (append-only). These changes must be written to stable storage, before the changes are reliably written to the database.*

Another name for WAL is a redo log: each log file consists of redo records. A redo record holds a group of change vectors, each of which represents a change made to a single block in the database

- 1 Update = 2 writes: when data gets updated, the index also needs to get updated:
  1. Update the WAL or REDO log
  2. Page update
- If something fails when it's updating the page, the DB will try again using the data in the WAL/REDO log
- Less levels vs more branches:
  - Each level can be a disk seek

– Using more branches means less disk seeks

4.2.7 Dataminded: Python + Postgres

Dataminded = consultent in big data technology with lots of experience using big data tools.

But: Python + Postgres can handle almost any analytics challenge

Figure 37: Use python libraries + a relational database for most analytics

4.2.8 When is SQL not the answer

- **Volume** = When you have petabytes of data (rare)
- **Velocity** = Too many writes per second (less rare)
- **Scalability**
  – Want to avoid expensive servers
  – Want to avoid expensive SANs (Storage Area Networks)
- **Variety** = When you don't want to turn an object (with unstructured data) into a relational row
  – = Object - relational database mismatch
  – `https://en.wikipedia.org/wiki/Object%E2%80%93relational_impedance_mismatch`

4.3 Key-Value

- Key-value DBs are relatively simple databases
- Adding to the end of a file (=appending) is the fastest way to write
- Example: a list of videos with their watch time
  – Key = video ID
  – Value = watch time, gets incremented often

4.3.1 Hash index

= a hashed key that will be searched for

- the hash (stored in RAM) will point to an offset on the disk
- if you find the key, you'll find where on the disk the data is stored
- if you want to update data, it will get appended to the end (=segment 2)

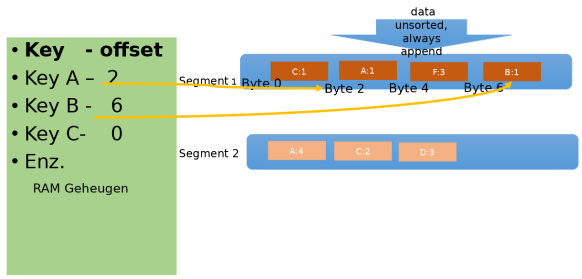


Figure 38: Log structure + hash index

4.3.2 Compacting

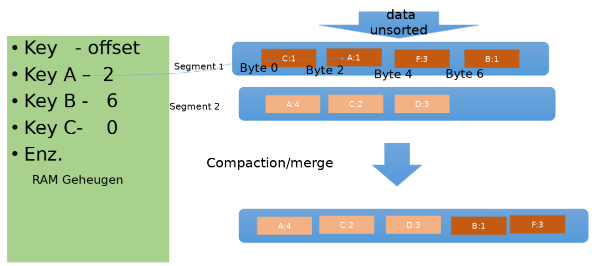


Figure 39: Compaction/merge of 2 segments

- In the above example: $A = 1$ in segment 1, but $A = 4$ in segment 2
- Segment 2 is newer: we ignore the first segment's A
- Same for C ($C = 1 \Rightarrow C = 2$)
- We compact/merge the segments until stored the data is correct
- We do this often

4.3.3 Principles

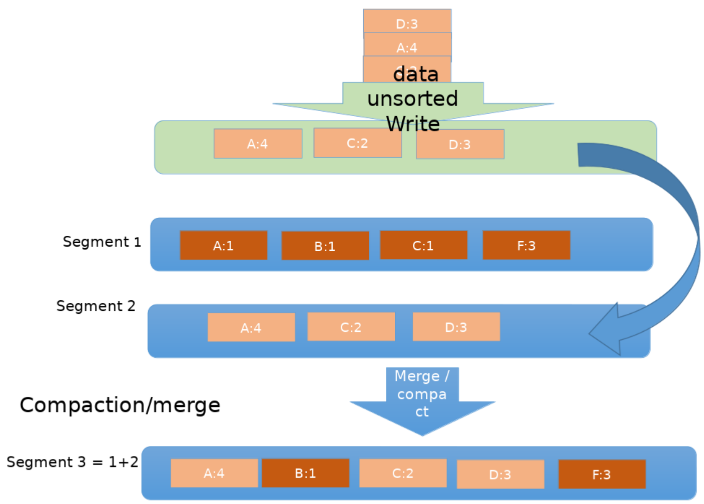
- Very fast writes (append only, so you can write sequentially because disks won't need to change tracks)
- Very fast reads if (sorted) hash index is in RAM
  - if not: very slow
  - SELECT * FROM A TO ZZZ (=slow)
- On crash: no corruption because of wrong update
- Example key-value hash index: **Riak Bitcask** (`https://en.wikipedia.org/wiki/Riak`)

4.4 LSM: Log Structured Merge Tree

Apart from the hash index, there is another Log Structured data store: the **Log Structured Merge Tree** with **Sorted String tables**

Definition 4.4 *A **sorted-string table** is a file used in LSM trees that contains key-value string pairs, sorted by key. An index is kept at the end of the file, which is kept in RAM.*

- Unsorted data now gets sorted first in a memory buffer (RAM)
  - Log segment file as backup
- Then it gets written to a segment
- Just like a hash index: values are not updated, but appended in a new segment
- Write to disk after several MB (to sorted string table file)



4.4.1 Log Structure Merge + Sparse tree index

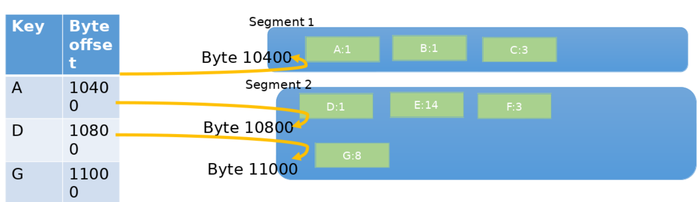


Figure 40: Sparse tree index

- We can simplify the LSM
- You don't need to keep every key in RAM
- Sparse tree index ⇒ remember where some milestones are
  - If you need F, and you know where D and G are
  - Start reading from D (byte-offset 10800)
  - Read sequentially until G (byte-offset 11000)
  - This sequential read is very quick, because it's a small amount of keys
- Merge, compact & sort every time = string sorted table

- Every delete: create new segment and merge.
- 'Tombstone the old segment' = marking key/value pairs for deletion

4.4.2 Applications of Sorted String & LSM-tree

- First application: Google Big Table (`https://en.wikipedia.org/wiki/Bigtable`)
- LevelDB (also by Google), RocksDB (fork of LevelDB)
- MyRocks (=a RocksDB storage engine for MySQL)
- Hbase, Cassandra (Facebook)
- ElasticSearch: based on Lucene text search (key = text, value = document) or inverted index

4.4.3 Advantages LSM

- Very fast writes (append-only)
- Quite fast reads (sparse index)
- Easily scaled over nodes (segments)

4.4.4 Disadvantages LSM

- Write (merge & delete) in background can influence speeds
- Deletes are costly because you don't actually delete: you tombstone a segment (=mark for deletion, it only gets deleted at the compaction step)
- You have to specify the rate between compaction & write/read (ongoing action) yourself

4.4.5 Summary: B-Trees vs LSM trees

Explain:

- Why do Write-Ahead Logs (WAL) exist?
- What is a compactor? Compaction planning?
- Advantages & disadvantages with much or little compacting?

4.5 Time Series

= LSM with a twist

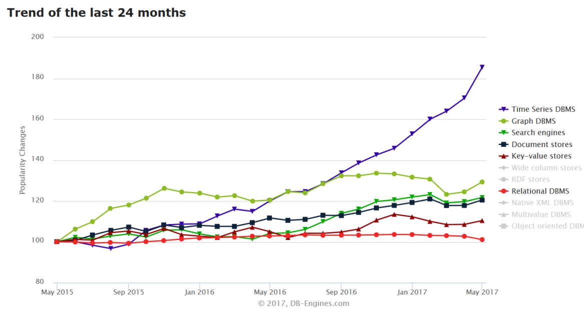


Figure 41: Popularity time series

Why the sudden rise in popularity?

- IoT devices use lots of sensors that need to be logged
- That sensor data is time based

4.5.1 Properties

- Lots of individual data points: 'a row is not important'
  - If you lose a data point, that's not a problem
  - You can guess what the data point would be, based on the data around the same timeframe
- High write throughput
- High read throughput (aggregation per hour/day)
- Large deletes (data expiration)
- Mostly an insert/append workload, very few updates

4.5.2 Use case: windmill sensors

Situation: a windmill has many sensors that produce data that needs to be logged

- Turbine sensor data needs to be stored every second
  - 30 sensor readings per second
  - > 300GB per windmill per year
- Both aggregation and realtime
- Issue with MySQL databases: read locks during queries ⇒ INSERT fails ⇒ causing data loss

4.5.3 Case study: influx DB

`https://docs.influxdata.com/influxdb/v1.4/concepts/storage_engine/`

Exercise: 'read the entire page, understand everything, and answer these questions:'

- Why important for MCT?
- How can you scale?
- TSM?

- WAL? How durable? What is it?
- What is a compactor? Compaction planning?
- Give one example of unique functionality typical for a time series environment
  - Tip: say you need to aggregate every hour

4.5.4 Object - relational mismatch

Definition 4.5 *Object-relational mismatch is a set of technical difficulties that are often encountered when an RDBMS is being served by an application program written in an object-oriented programming language, because objects or class definitions must be mapped to database tables defined by a relational schema.*

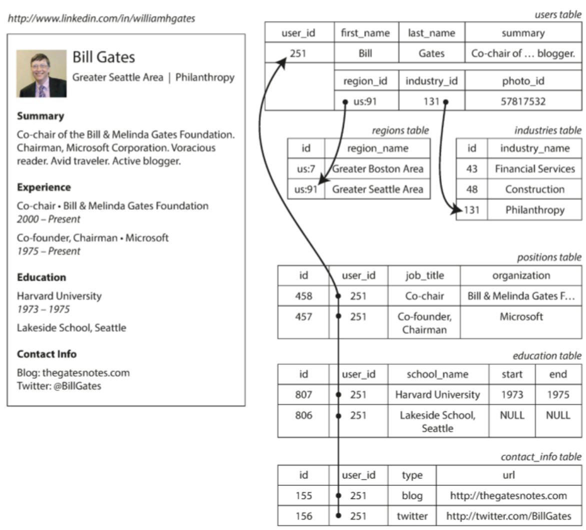


Figure 42: Representing an object in relational rows

- Example: a LinkedIn-like application on an RDBMS with lots of 1 to n relations
- These relations are prone to object-relational mismatch problems
- A better match for objects with unstructured 1-n data: JSON documents:

```
{
    "user_id": 251,
    "first_name": "Bill",
    "last_name": "Gates",
    "positions": [
        {"job_title": "Co-chair", "organization": "Bill & Melinda Gates Foundation"},
        {"job_title": "Co-founder, Chairman", "organization": "Microsoft"}
    ]
}
```

4.5.5 JSON in PostgreSQL

- Mid 2014: PostgreSQL 9.4 natively supports JSON

- The speed to ingest documents as quickly as MongoDB, but ACID!
- Fully indexable

4.6 ElasticSearch

- = a document Store
  - So naturally suited for describing objects
  - JSON serialized
- Easy access to an advanced fulltext search-engine library
- Lucene is very complex but very advanced
  - Lucine = LSM sorted string table that works with segments
  - Automatized sharding (and thus scalable) in containers
- Has a RESTful API: you can use commands like curl, wget, . . . to interact with the database as if it's a website
- Data ingest is not very fast ('index')
  - Can be used for time series, but a real time series DB is better

4.6.1 Elastic Search architecture, the basics

- Document = JSON data
- Index = a collection of documents (ElasticSearch index ! = database index)
- Shards = scalable pieces of index
- Segments = sequential pieces of a shard

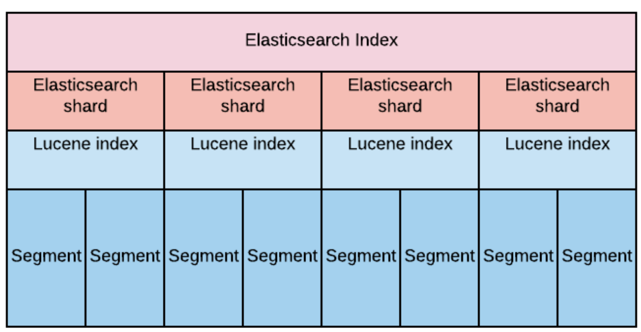
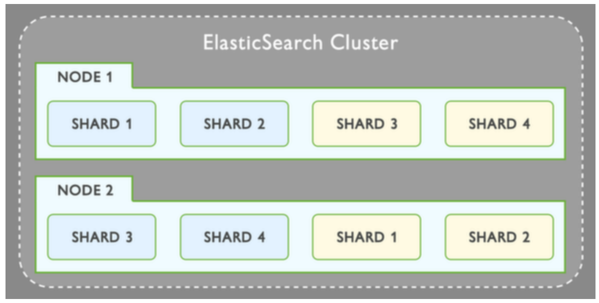


Figure 43: Elastic Search architecture

4.6.2 Elastic Search Cluster

- Index is split over shards - nodes: scalability
- Shards can be replicated over nodes: availability



4.6.3 Inverted index

Definition 4.6 *An inverted index is a database index storing a mapping from content, such as words or numbers, to its locations in a table or document.*

This is in contrast to a 'forward index', which maps from documents to content.

- The power of ElasticSearch is text search using Lucine
- = 'Lucene Index'
- A document will be seen as important if it contains a relevant word many times
- If a word is very common, Lucine will see it as a stopword (a, and, for, is, in, it, . . .)
- Lucine can work with misspelled words, using a certain 'distance' of characters (= amount of characters that can be wrong)
- Disadvantage: increased processing when a document is added to the database

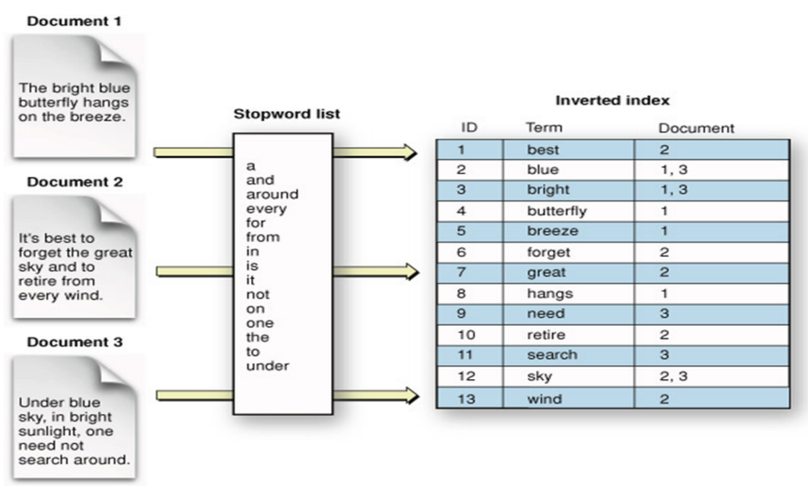
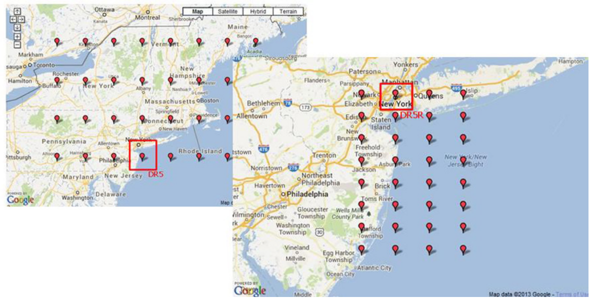


Figure 44: Inverted index: an index where words are mapped to the location where that word appears

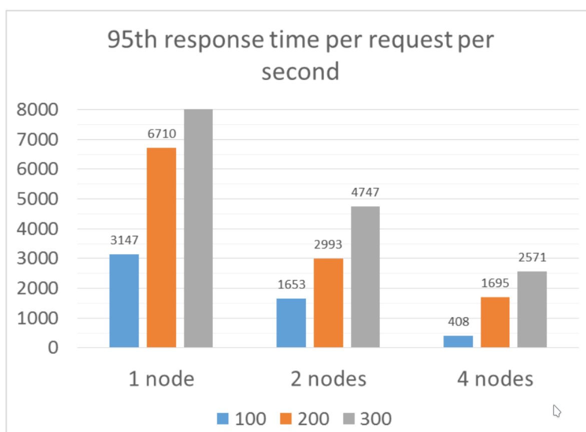
4.6.4 GeoHashes: Representing Geospatial data in ElasticSearch



How it works is less important for this module, just know that it exists and uses ElasticSearch's text search capabilities

- Since ElasticSearch 0.90
- Base32 encoded strings, interleaving the latitude and longitude
- Max resolution: 40mm * 20mm
- Each extra symbol divides the grid in 26 cells

4.6.5 ElasticSearch scaling



- ElasticSearch is very scalable
- 1 node ⇒ 2 nodes: almost double the performance

4.7 Which storage engine is the best and the worst

Which storage engine is the best/worst for the following situations:

- High amount of writes every second (sensor data)
- High amount of updates every second: Web analysis data (Marketing campaign)
- Continuous updates and reads of personal data?

- Full scans on structured data?
- ACID compliant OLTP?

4.8 Summary

4.8.1 Hash index vs LSM datastore

Hash index

- Fast reads when index is in RAM
- Very fast writes (append)
- Slow scans (select *, group by, . . .)
- Example: Riak Bitcask

Log Structured Merge Tree

- Fast read speed (offset + sequential scan), **a bit slower than hash index**
- Very fast writes (append)
  - But because of sorting in RAM, we need to first write to a read-ahead log
  - Or else data can be lost if the data is still being sorted in the memory buffer (for example due to power failure)
- **Fast scans** because of sorted string tables
- Examples: Google Big Table, LevelDB, MyRocks, Hbase, Cassandra, ElasticSearch

4.8.2 B-tree vs LSM

B-tree

- **Very** fast in random reads
- **Fast updates**
- Not useful for high volume writes
- Very fast scans when index is on correct column

LSM Tree

- Fast read speeds
- Slower in updates because of tombstone: mark for delete + append
- **Very fast writes** (append)
- Very fast scans (sorted!)

5 Distributed Stores

Definition 5.1 *A distributed data store is a computer network where information is stored on more than one node, often in a replicated fashion.*

It is the perfect match for big data: volume, velocity, variety

5.1 Terminology

5.1.1 Shard/partition

Definition 5.2 *A shard or partition is a small subset of the database that can be assigned to a node*

- The database is divided into nodes
- ElasticSearch, MongoDB, MySQL: 'Shard'
- Hbase: 'Region'
- Cassandra, Riak: 'vnode'

5.1.2 Replica

Definition 5.3 *A replica is a copy of a shard/database that is kept on a different machine*

- If a shard gets corrupted or lost, replicas serve as a backup solution
- The replicas are kept on **different** nodes: never on the same node

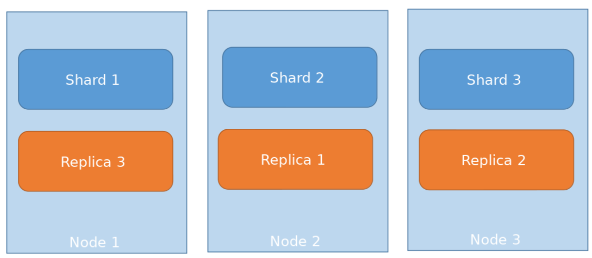


Figure 45: Shards and Replicas are split over multiple nodes

5.2 Elastic Search Cluster

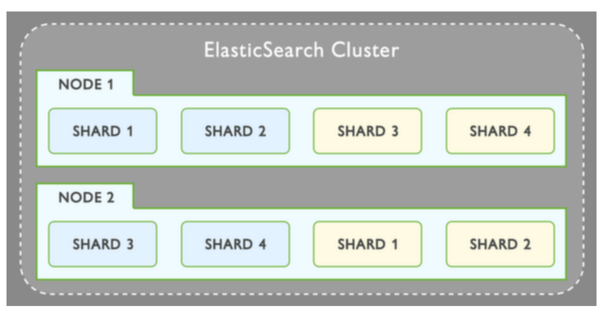


Figure 46: Example cluster with 2 nodes, 1 replica shard for each node

- Index is split over shards - nodes: **Scalability**
- Shards can be replicated over nodes: **Availability**

5.3 Replication: master/slave or leader/follower

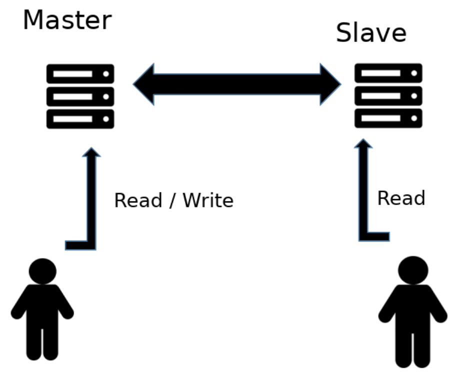


Figure 47

- Multi-node or distributed data systems can be quite complex
- The master (one server) does all read and write operations
- The data gets copied to the slave (the replica) for redundancy
- You can only read to the slave

5.3.1 Communication between master & slave, two ways:

Synchronously

- The master writes new data to the slave
- The slave must acknowledge the data
- The master must wait for the confirmation by the slave
- Consistent data
- Slow & unreliable with many slaves

Asynchronously

- No acknowledgment of slave
- Inconsistent data
- Fast, even with many slaves

5.3.2 Consistency choices

Strong Consistency

- At a certain point in time after a write, all replicas return the same update record/document (almost realtime)
- Consistent with order in which write operations are submitted by clients

Eventual Consistency + High availability

- Faster, easier to be 'available'

- Hard for devs: when updating a row, you can not be sure when it will be updated in other nodes
- Every request received by a non-failing node must result in a ‘non-error’ response
- Nodes can read/write - even if that means that not all replicas have the same content

5.3.3 Amount of leaders

Single leader

- Only one node (the leader) can write
- The follower gets the updates as fast as possible, synchronously
- Only for systems with few write operations and many read operations
- Strong consistency is possible

Multi leader

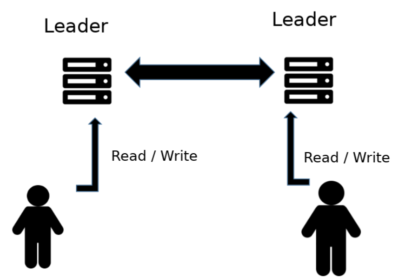


Figure 48: Multi leader

- More than one node can write
- ⇒ write conflicts are possible, because the same data can be updated on different nodes
- These conflicts must be solved ⇒ system is much more complex
- Not consistent = replica may be out of sync

5.3.4 Split-brain and Network partition

Definition 5.4 *Network partitioning happens when a network is split due to the failure of network devices. The network devices stop communicating and synchronizing their data to each other.*

Definition 5.5 *Split-brain is a problem that indicates data or availability inconsistencies originating from the maintenance of two separate data sets, for example because of network partitioning.*

What if the network connection drops?

- Do we allow the follower to become the leader?
- Or do we block from writing to follower?
- But leader might still be alive and serving
- Result: if we allow data to be written to the follower, and the leader is still serving, both nodes might have different data

5.4 CAP Theorem

Definition 5.6 *The CAP theorem states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees:*

- ***Consistency****: every (later) read operation will always return the last version (*) of the data, or an error message (single object)*
  - *(*): last written by a write operation X older than this read operation*
- ***Availability:*** *there is always a server available, if necessary: with older data*
- ***PartitionTolerance:*** *the system must keep working, even if the nodes can't communicate with each other ('network partitioning')*

We can use the CAP theorem to describe the problems originating from network partitioning.

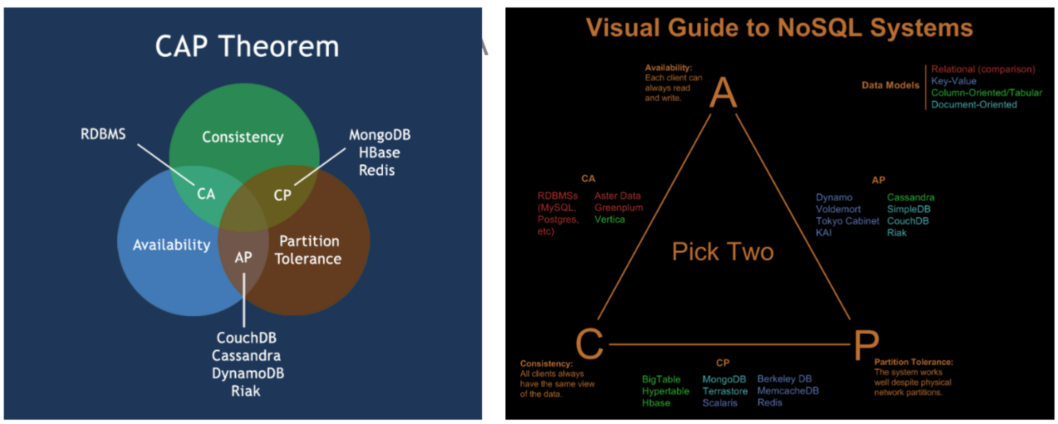


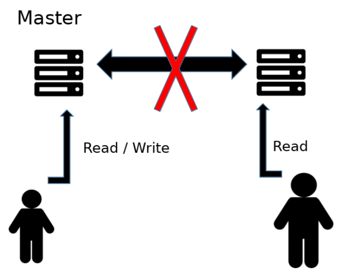
Figure 49: Oversimplification of CAP theorem

5.4.1 CAP: either consistent or available when partitioned

- When we come accross network partitioning, we will have to choose between 2 properties if we want to keep partition tolerance:
  - Consistency
  - Availability
- Both at the same time is NOT possible
- When there is no network partitioning, then both Consistency and Availability are satisfied

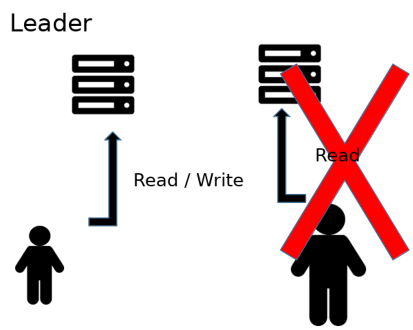
⇒ We will look at a couple possible situations, using a relational database as an example:

5.4.2 Relational DB: CA



- If we want to keep consistency, we will only allow reads to the slave
- Only consistent if perfectly synchronized
- In reality, we only allow reads to slave
  - $\Rightarrow$ no ‘A’ for writes
  - $\Rightarrow$ CAP theorem has no value here

5.4.3 Relational DB: CA: single node (no P possible)



- = simpler system: single node: we only read / write to one node
- Partition tolerance is not possible because there is no network between nodes
- $\Rightarrow$ CAP theorem also has no value here

5.4.4 AP systems

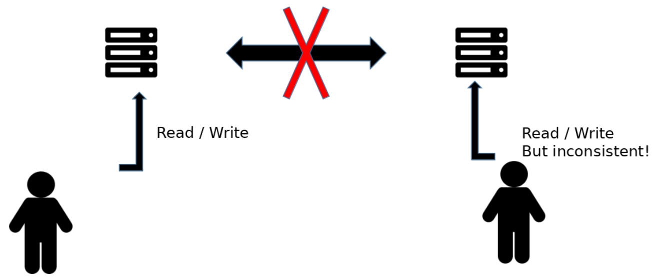


Figure 50: AP: ensure Availability and PartitionTolerance when there is network partitioning

- The system will always return data, but not always consistent (if network issues between nodes)
- If we read or write data and this fails, another node will take control, no error messages
- We choose AP in situations where reading data is more important than writing
- But how available is this system in reality?
  - It might take a while before replica's know they can't synchronize
  - There might be considerable lag when waiting for a response
  - $\Rightarrow$ not as much availability after all $\Rightarrow$ not so useful

5.4.5 CP

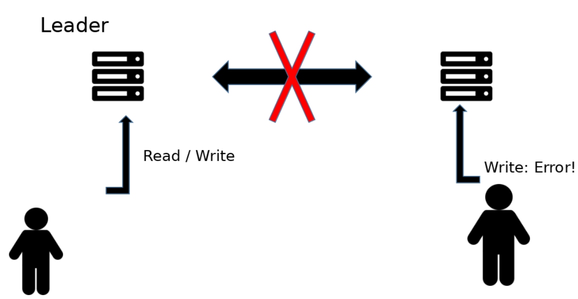


Figure 51: Consistency and PartitionTolerance: The system will always return the correct data

- When network partitioning happens, consistency is kept: it will either not return data, or return the correct data
- We will not be able to read the data as long as these were not written to all nodes
- If we can't write data, we will see an error message $\Rightarrow$ no availability
- We can use CP if data consistency is very important

5.4.6 Conclusion 1

CAP theorem is not clear way to make architectural choices. The creators even admitted this.

- CA doesn't exist: >1 node & partitioning == no real availabilty anymore (only for read)
- When CAP theorem describes consistency, it talks about a single object, but in many applications consistency over many rows (in many tables) is important
- No CP: with many replicas, you should choose for asynchronous replication:
  - So no strong consistency
  - No availability (no writes possible)

5.4.7 Conclusion 2

CAP theorem only serves to clarify the difference between AP vs Single node consistency

- When using multiple nodes, there is only 'eventual consistency' and 'read availability' ⇒ CP is impossible
- Multi node systems are always a little inconsistent
- AP is possible but only for partition problems
  - Is being available in 1 situation (network partition, rare) even that useful?

5.5 How does Elastic Search handle network partitioning?

5.5.1 Consistency and Network partitioning

`https://www.elastic.co/guide/en/elasticsearch/reference/2.4/docs-index_.html#index-consistency`

- To prevent writes from taking place on the 'wrong' side of a network partition
- By default, index operations only succeed if a quorum(=voting procedure) (>replicas/2+1) of active shards are available.
- This default can be overridden on a node-by-node basis using the action.write_consistency setting.
- To alter this behavior per-operation, the consistency request parameter can be used.
- Valid write consistency values are one, quorum, and all.

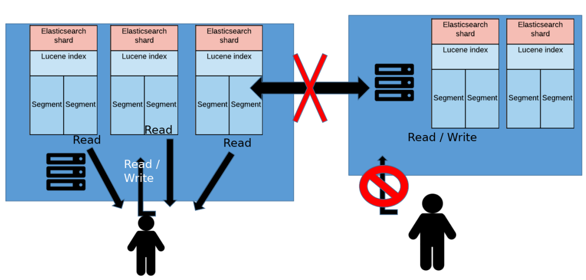


Figure 52: ElasticSearch

- In the above example: we have 5 replicas
- If 3 out of 5 replicas respond, we will still be able to read and write
- This is because the majority of the quorim is available

- This way, ElasticSearch ensures a certain amount of consistency and availability

5.5.2 Architectural choices

If you have to choose and configure a datastore for your application, let it be clear that CAP theorem is not a good way to choose between data stores. The following questions are more useful:

Do you need transactions? (Multi-object transactions?)

- = Do i need to keep multiple objects consistent?
- If very important: pursue high isolation (atomic)
- How important is availability: choose reliable (expensive) hardware
- How important is performance: very expensive hardware for high load (limited scalability)
- If not very important: Read committed + hardware dependant on load
- If availability very important: fast network (synchronized!) and expensive reliable hardware + few (1) replicas

How important is scalability and performance:

- Very high load = low consistency & limited availability
- Rather choose for databases that are easily sharded (so no transactional DBs)
- Availability: high amount of replicas + 'normal hardware' ⇒ lower consistency!

	Garanties	**Hoe?**	**Nadeel**
Read committed	Geen dirty writes	Row-level locks (or table locks)	Inconsistente data: Read Skew
	Geen dirty reads	Oude waarde weergeven zolang transactie bezig is	Lost updates : 2 updates tegelijkertijd
snapshot isolation	+ Geen read skew	+ multiple versions of a row (or object)	Lost updates : 2 updates tegelijkertijd
		Resultaten van latere transacties zijn niet zichtbaar voor vroegere	
Atomic updates	Geen lost updates	Exclusive lock - geen Read tijdens transactie	"write skew"
Serial execution	"perfect"	Single threaded execution	Traag naarmate DB groter wordt - Single Thread is vooral interessant voor "perfect" gepartitioneerde DBs. Beperkte complexiteit van data (bvb. eenvoudige Key-value)
Two Phase Lock	"perfect"	Multi threaded, goed voor ruime "checks"	Werkt beter voor "Complexe data", nog altijd zeer traag: Blokkeert potentieel hele veel data

Figure 53: Isolation overview

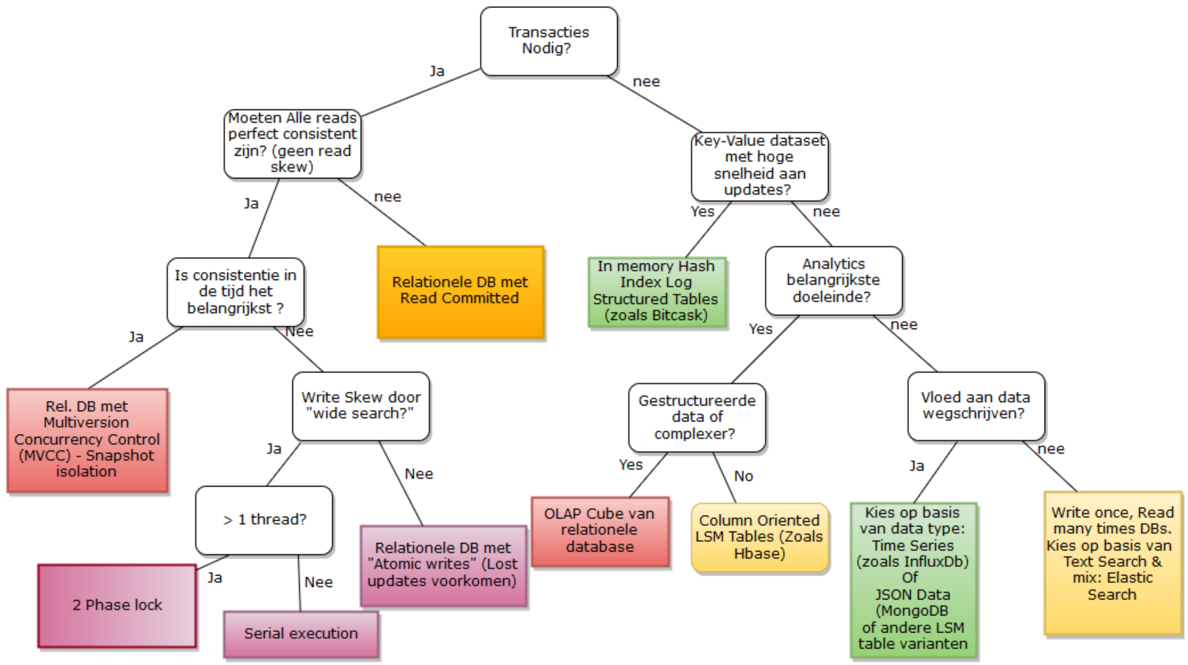
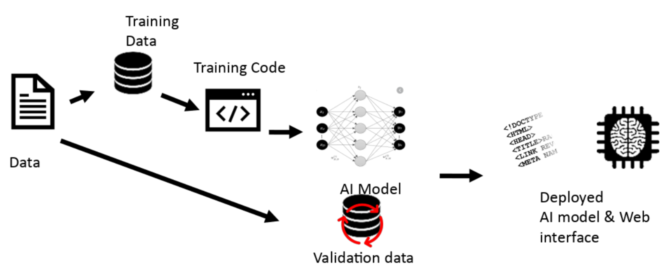


Figure 54: Simplified overview datastore choices

6 Batch Processing



- In data driven programming, one of the steps is to download data
- To use the data, it needs to be processed first

6.1 Data processing

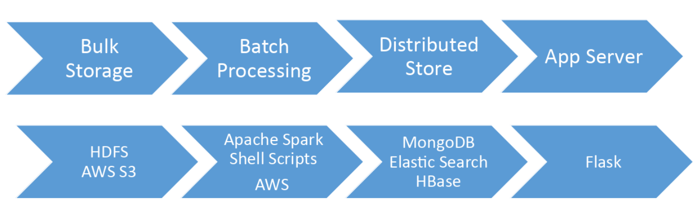


Figure 55: The data pipeline (can also be a chain of simple unix commands)

Data processing can be done in 3 ways:

6.1.1 Request - Response

- HTTP/REST API
- SQL - Relational DB
- Result: **online** 'life data'
- Response time: milliseconds - seconds

6.1.2 Batch Processing

- Unix tools - Map/Reduce - Online Analytical Processing (OLAP) with ETL
- **Offline** - Throughput, high latency - Full scan reporting
- Result: 'derived' data
- Response time: minutes - days

6.1.3 Stream Processing

- Processing events - Twitter, Kafka
- **Near Real Time**: low latency, sliding window & continuous results

6.2 Batch Processing

6.2.1 Basic principles

- Dataset is limited in length, size, time, ... For example:
  - One big log of all web activity of the last month
  - One log of all system activity the last day
- Datasets are immutable: not changed by processing
  - Original file/log does not get changed
  - Result of batch processing: new dataset
  - New dataset with new calculations (group by)
- Reason: processing can always start again if you make a mistake

6.2.2 Unix style

- Piping: |
- Redirection: & or >

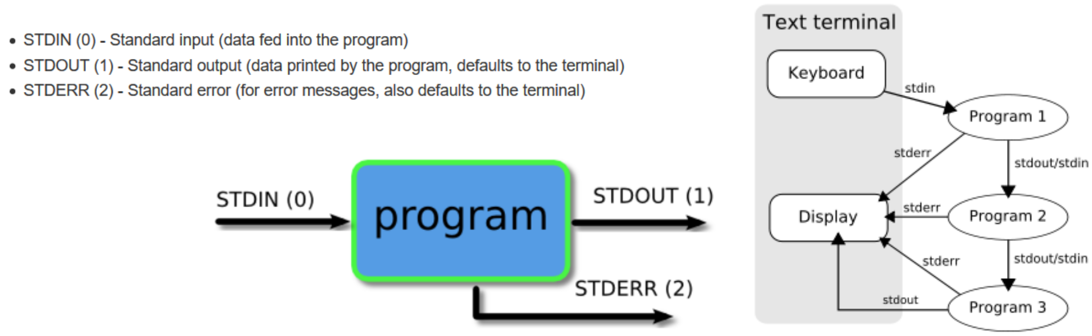


Figure 56: Linux streams

A linux program always has three standard streams:

- Standard input = STDIN (0)
- Standard output = STDOUT (1)
- Standard error = STDERR (2)

6.2.3 Why not everything is possible with Unix tools

- What if the data does not fit on one (logical) drive?
- Do you even want all data on one (logical) drive? (Throughput/latency/. . .)
- For both storage and processing:
  - Scale out: more processor cores, . . .
  - Possible with cheap(er) hardware

6.3 Map/Reduce

= A programming model for processing big data sets with a parallel, distributed, algorithm on a cluster

`https://en.wikipedia.org/wiki/MapReduce`

6.3.1 Hadoop

Definition 6.1 *Hadoop is a software framework by Apache for distributed storage and processing of big data using the Map/Reduce programming model. Hadoop is made out of modules, each of which carries out a particular task essential for a computer system designed for big data analysis.*

`https://en.wikipedia.org/wiki/Apache_Hadoop`

Modules

- **Distributed filesystem:** allows data to be stored in an easily accessible format, across many linked storage devices.
- **MapReduce:** the basic tools for data analysis
- **Hadoop common:** provides the tools (in Java) needed to read data stored under the Hadoop file system

- **YARN:** manages resources of the systems storing the data and running the analysis

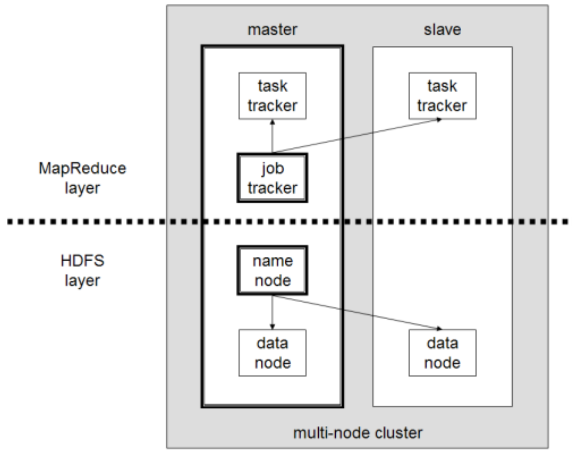


Figure 57: A multi-node Hadoop cluster

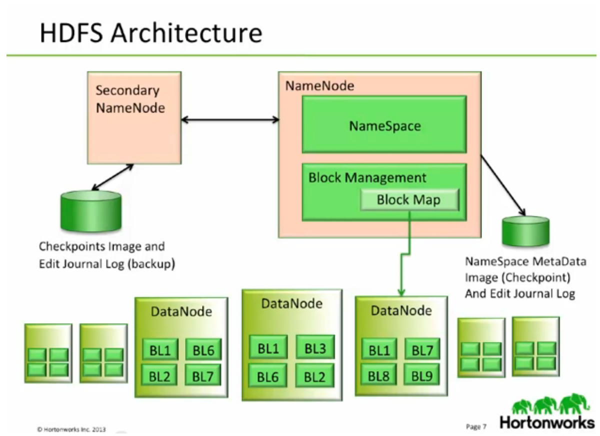
Two layers:

- MapReduce layer
  - The job tracker is responsible for the distribution of tasks to ‘task trackers’
  - Task trackers = compute nodes
- HDFS layer
  - Name node tracks ‘files on which node’. It functions as entrance to the file system, which consists of many data nodes.
  - 3+ replicas (availability)

6.3.2 HDFS

Definition 6.2 *The Hadoop distributed file system (HDFS) is a distributed, scalable, and portable file system written in Java for the Hadoop framework. It is used for storing the data that will be processed by MapReduce.*

- = ‘Network attached storage on steroids’
- Big datablocks (64+ MB) + replication over many nodes, many disks and standard network.
- HDFS consist of a daemon process running on each machine, exposing a network service that allows other nodes to access files stored on that machine
- HDFS creates one big filesystem that can use the space on the disk of all machines running the daemon
- A central server called the **NameNode** keeps track of which file blocks are stored on which machine
- The blocks are replicated on multiple machines (for availability and redundancy), similar to RAID.



6.3.3 Map reduce

Map Reduce is named after the two basic operations this module carries out:

- Map = read data from a DB, put it in a format suitable for analysis
- Reduce = perform mathematical operations

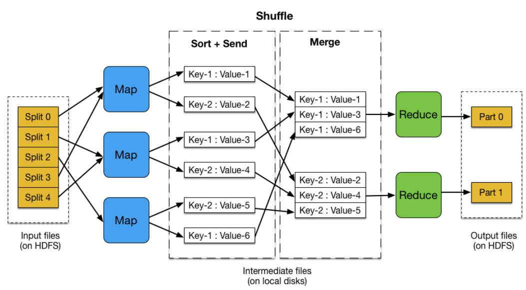


Figure 58: Map reduce schematic

1. Read input files (HDFS input parser)
2. Map fase
    (a) Get a key and value log from your total log
    (b) Sort and send per key
3. Reduce fase
    - Merge all equal keys (key2 - value1, key2 - value2)
    - Process over all equal keys (already sorted)
        – Only one record per key
        – Ex: count how may (uniq -c)

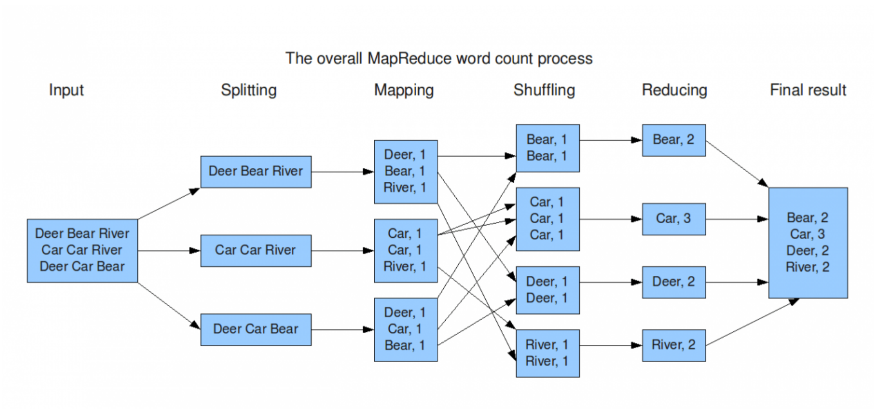


Figure 59: Example with wordcount

Ideally every mapper and reducer work on their own disks.

6.4 Spark - Data processing framework or Dataflow engine

Definition 6.3 *Apache Spark is an analytics engine for large-scale data processing. It uses Resilient Distributed Datasets (RDDs): an immutable distributed collection of elements of the data. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster.*

Spark and its RDDs were developed in 2012 in response to limitations in MapReduce.

https://en.wikipedia.org/wiki/Apache_Spark

6.4.1 Why Spark is more powerful than Hadoop

Problems with Hadoop:

- Doing everything with Map & Reduce: good for simple problems like wordcount, but not very flexible for complex tasks
  - $\Rightarrow$ No integration with Machine Learning
- Next step (for example Reduce) can only start if **all previous tasks** are finished.
  - Previous task creates an 'intermediate' file that the next task needs
- Good throughput (TBs of data), but long response time (minutes, hours) before you get a result because of constant disk activity
- Hadoop only has Map & Reduce
- Hadoop sorts even when not necessary

What Spark does better:

- Spark has many more operators than just Map & Reduce
- Sorting is a seperate operator $\Rightarrow$ only sort if actually necessary
- Uses a DAG (Directed Acyclic Graph) as an execution plan, similar to a database query plan
- Only writes to disk if necessary: makes more use of memory

6.4.2 Execution plan using a Directed Acylic Graph (DAG) in Spark

Spark creates a DAG to figure out the best way to execute a job

- Similar to ‘query planner’ from RDBMS
- A job is associated with a chain of Resilient Distributed Dataset (RDD) dependencies organized in a DAG

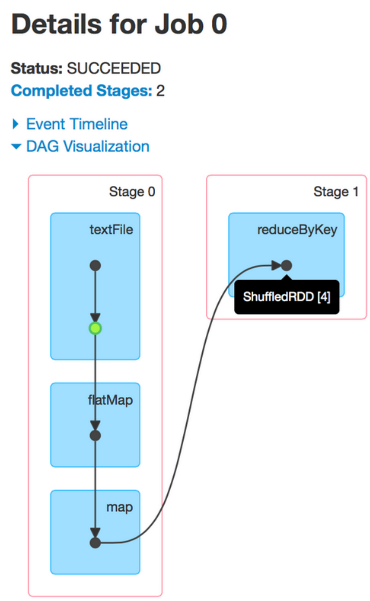


Figure 60: Example of a Spark job

- This job performs a simple word count:
  - First, it performs a textFile operation to read an input file in HDFS
  - Then a flatMap operation to split each line into words
  - Then a map operation to form key value pairs {word: amount of occurences}
  - Finally a reduceByKey operation to sum the counts for each word
- The blue boxes = the spark operation that the user calls in their code
- The dots in the boxes = RDDs created in the operations. The operations are group by the stage they are run in.
- This visualisation shows 2 interesting things:
  1. It reveals the Spark optimization of pipelineing operations that are not seperated by shuffles. After reading from HDFS, each executor directly applies the flatMap and map functions to the partition in the same task, removing the need to trigger another stage
  2. One of the RDDs is cached in the first stage (green dot) $\Rightarrow$ future computations on this RDD can access at least a subset of the original file from memory instead of from HDFS

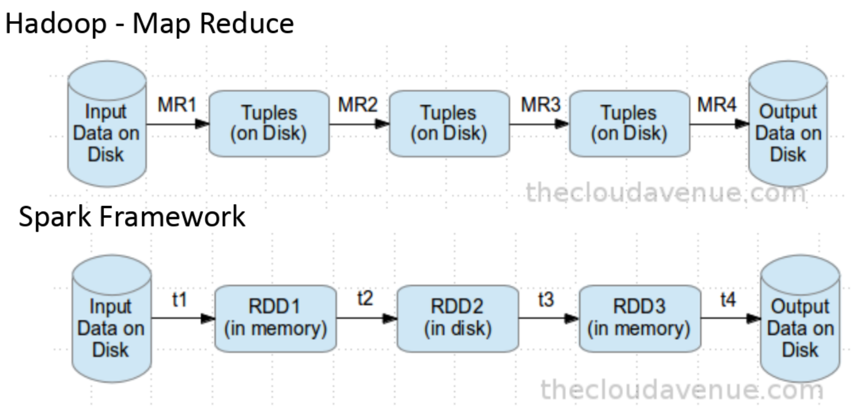


Figure 61: Hadoop (Map Reduce) vs Spark Framework: Only 1 write to disk, everything in memory

6.4.3 RDD or Resiliant Distributed Dataset

- RDD is a read-only, collection of records partitioned across the nodes of the cluster
- Fixed number of partitions
- They can be operated on in parallel
- firstRDD = sc.textfile('file.txt')
  - sc = spark context
  - .textfile = method

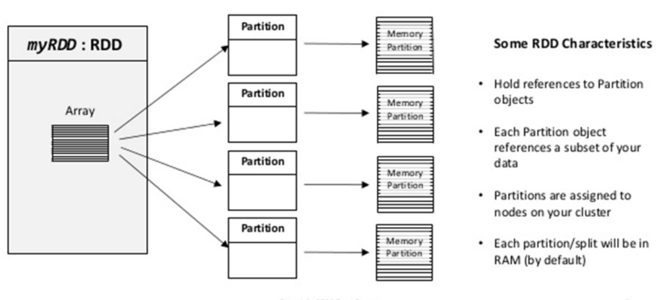


Figure 62: We divide the RDD into partitions, which are in turn also divided into memory partitions

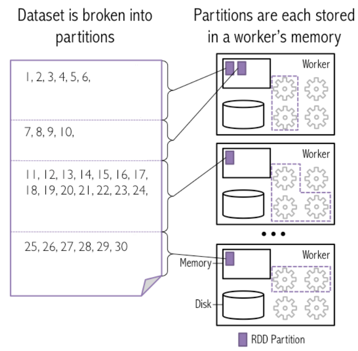


Figure 63: Large textfile example

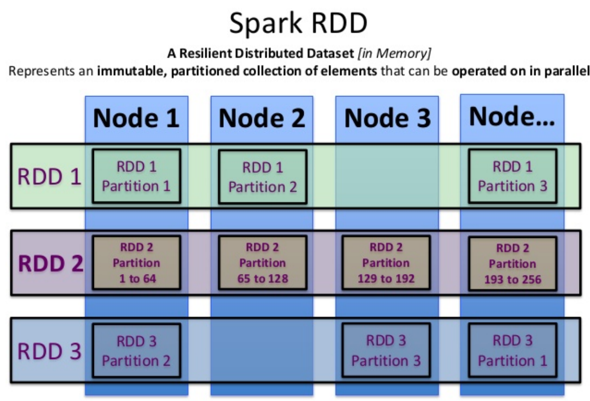


Figure 64: The RDDs are partitioned over multiple nodes (servers)

- RDD is the fundamental data structure of Spark
- Allows a programmer to perform in-memory calculations on large clusters in a fault-tolerant manner
- $\Rightarrow$ speeds up the task
- Spark Dataframe APIs:
  - Unlike an RDD, data is organized into named columns
  - Immutable distributed collection of data
  - Allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction

6.4.4 Complete Spark framework

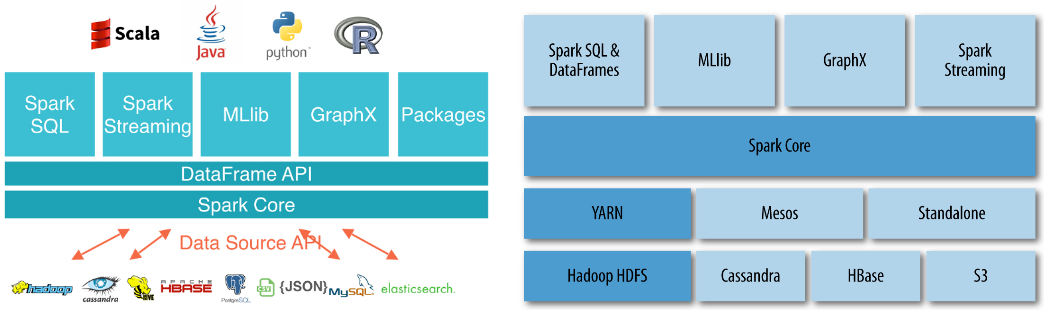
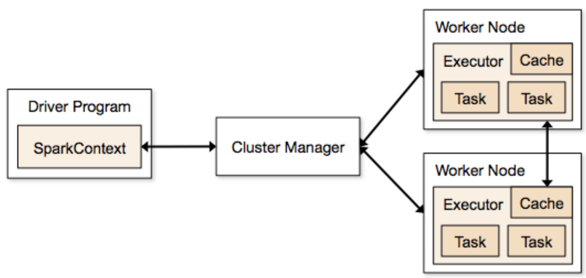
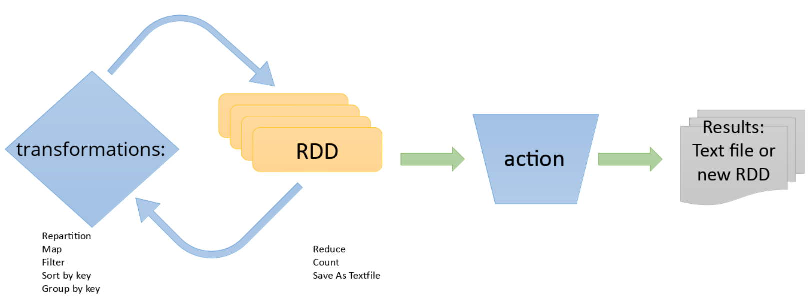


Figure 65: A complete framework that uses Spark

6.4.5 Realtime in-memory processing with Spark



- SparkContext = programmable object
  - Local or cluster
  - Starts with a session
- Every executor processes tasks (+ in memory storage/cache)
  - 2-3 tasks per CPU



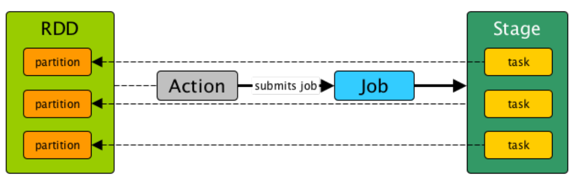
- RDDs are divided into memory partitions
- You run transformations on these RDDs (happens in-memory)
  - Repartition (when you add a server and you want to repartition the data so it is also partitioned on the new server)

- Map: key values
- Filter
- Sort by key
- Group by key
- Only when you run an action (reduce, count, save as textfile, . . .), you will write to disk
- Result: text file or new RDD

6.4.6 Jobs

= RDD + action

- Stages = parts of the execution plan of the job
- Contains a sequence of transformations that can be completed without shuffling the data
  - Get data (stage 1)
  - Group by (stage 2)
  - Map & union (stage 3)
  - Join everything (stage 4)
- Every stage = set of parallel tasks
  - Same code - different subset of data



One job, multiple tasks: For example: counting words in a file:

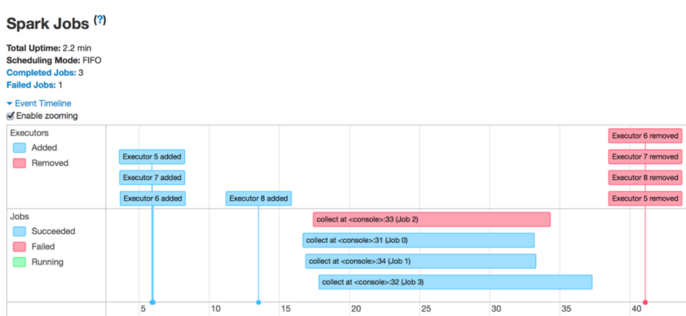


Figure 66: Overview of multiple jobs

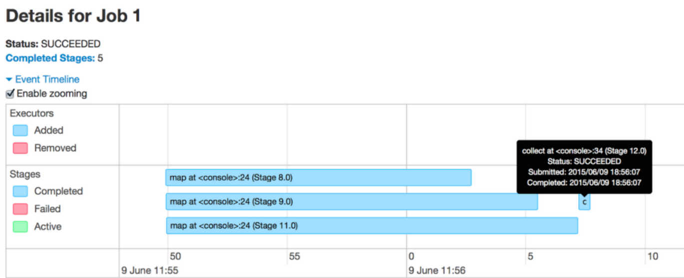


Figure 67: Details of one job: multiple maps per job

- This job runs word count on 3 files and joins the results at the end
- From the timeline, it's clear that the 3 word count stages run in parallel (because they do not depend on each other)
- However: the join at the end does depend on the results from the 3 stages
- Consequence: the collect stage at the end does not begin until all previous stages have finished.

6.5 Conclusion

- Batchprocessing = producing derived data in several steps without changing the source
- Unix pipeline can be very powerful - more than typical sort-transform in programming languages
- Use spark for multi-node batch processing

7 Stream Processing

7.1 The problem

Say you need to create an application for a windmill park

- What if you need to detect the situation: 'In the last minutes the temperature is rising while the power is dropping'
- Not very suitable for batch processing
- If you would use a database instead:
  - Would need to poll a lot: overhead!
  - Processing raw data is a lot slower than just ingesting

7.2 Batch vs Stream

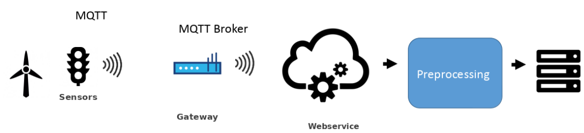
7.2.1 Batch

- Collect chunk/batch (one day) of data
- Process until all data is processed
- Work with averages, percentiles, . . . over longer time
- Example: ETL process from OLTP data to Data cube

7.2.2 Stream

- Little chunks (events) are fed continuously to the ‘stream processor’ (=software)
- Never stop processing
  - Process sliding window (for example: 1 hour)
  - This allows you to do trend analysis very efficiently
- Detect trends quickly (sharp decline, ...) near realtime
- Example: Windmill Sensor data trend analysis

7.3 IoT use case



- Preprocessing of (time series) sensor data takes longer than sending the sensor data
- Connect more than 1 processing app
- IoT device - MQTT - Webservice problem situations:
  - What if too many IoT devices connect and send data?
  - What if a service fails?

7.4 Summary: What needs to be solved?

- Many producers of data (windmills): can consumer/processing keep up?
  - Need for load balancing, tracking orders, ...
- Many consumers:
  - Polling at low delay = overhead - rather have ‘notifications’ (push instead of pull)
  - How to make sure you do not need to change the API all the time
  - How to keep the number of interfaces low? (Errors!)
- Syncing between producers and consumers
  - Crashed - heartbeat
  - Error handling
  - ‘Flow control’

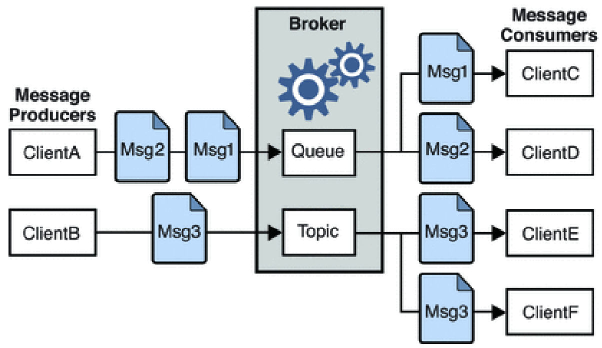
These challenges cannot be solved properly with classic database solutions.

We need a software solution: a message broker

7.5 Message broker

= server software that connects message producers (=data producers) and message consumers (=data consumers) as clients

The message producers will publish events in the queues of a message broker.



7.5.1 Publish/Subscribe model

- Producer = sends 'events' or 'messages'
- Topic = consumer is subscribed to this queue and gets notified
- **Load balancing:** distribute the load between multiple message consumers
  - = each message is delivered to *one* of the consumers
- **Fan-out:** data of a certain topic is broadcast to all data processors
  - = each message is delivered to *all* of the consumers

7.5.2 Classic reason for a message queue

Message queues have been used for much longer than since the IoT boom, for example:

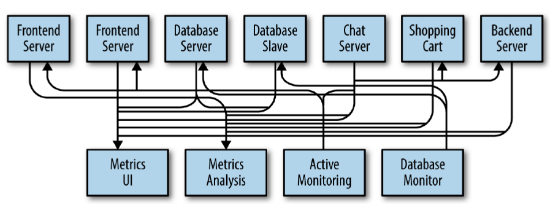


Figure 68: To avoid 'spaghetti services architecture', you can use a message broker inbetween the servers and data consumers

7.5.3 Database vs Message queue

Database:

- Stores data
- Keeps data until it is deleted
- Fast Random Search by index

- Only if a trigger is programmed, app is notified

Message queue:

- Stores data
- Keeps data until the data is processed, then deletes it
- Consume in order
- All subscriber consumers get notified

7.6 3 types of messaging systems

Messaging systems always have to make the trade-off between:

- Latency = how fast the data gets sent and processed
- Durability = how sure I am that the data gets processed at all

We start with the lowest latency and lowest durability, and move our way up:

7.6.1 Direct messaging

- UDP multicast - videostream, financial data, stock broker monitoring
- It's important to get the latest data as fast as possible
- Very low latency
- When too much data is sent at once:
  - Solution: use back pressure / message dropping

7.6.2 Message brokers (RabbitMQ, Azure Service Bus)

- Message brokers save the messages in a message queue in RAM
- Sometimes the message is backed up to a log on the disk
- Once message is received, message queue entry is deleted
- In memory, or if RAM is full, let it 'spill over' on the disk disk
- Problem: If a message needs to be resent to a consumer, it is not certain that it arrives in the correct order.
- Low latency, a bit higher than Direct Messaging

7.6.3 (Partitioned) Log message queue (Kafka, AWS Kinesis)

- Message stays for a while in an append-only log
- Good latency (but higher)
- High throughput (parallel partitions on different disks, or even on different nodes)

7.7 Advantages Message Queue / Broker

- Producers can crash
- Producers can change the format of the message (eg.: make it longer)
- Producers don't have to wait on consumers
  - Fire and forget
  - Buffering
  - Lower latency
- Consumers can crash: message is buffered

7.8 (Partitioned) Log Message Queues

= Message queues that work with an append-only log file

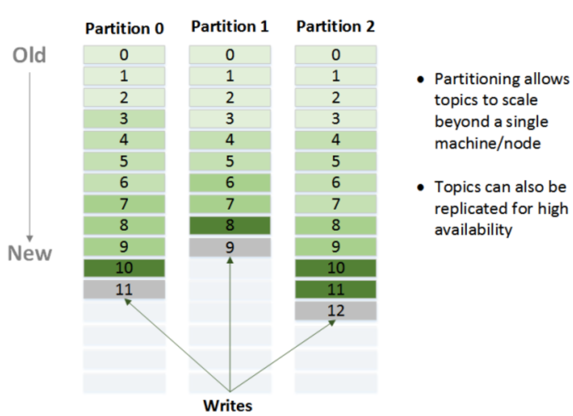
- **Topic** = all events that are similar, for example: all windmill sensor readings
- We can make topics scalable by writing these events to different partitions
- All the partitions for a topic listen to the same producer
- Topics can also be replicated for high availability

`https://www.youtube.com/watch?v=avi-TZI9t2I`

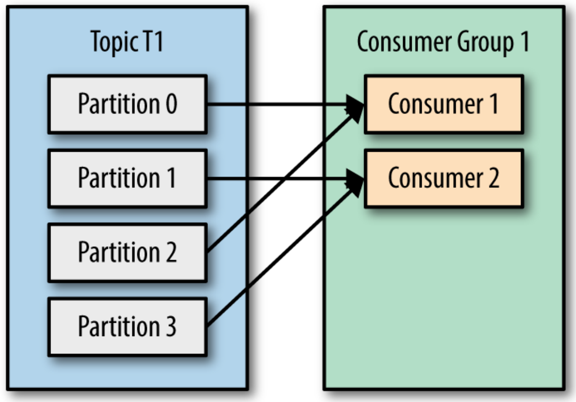
`http://www.benniehaelen.com/hadoop/using-apache-kafka/`

7.8.1 Advantages (Partitioned) Log Message Queues

- Very scalable
- Every time we add data to a partition, it gets a sequence number (= a 'message offset')
- A message consumer has its own offset that can be compared with the partition's offset
- Consumer can crash and then restart with the correct **offset**



7.9 Partitioning consumer



- Consumer can have 1 to n partitions
- No 2 consumers in the same group have any partition in common
- Every partition writes data to (maximum) one consumer
- $\Rightarrow$ loadbalancing

7.10 Message Queue vs Log based message queue

7.10.1 Message queue

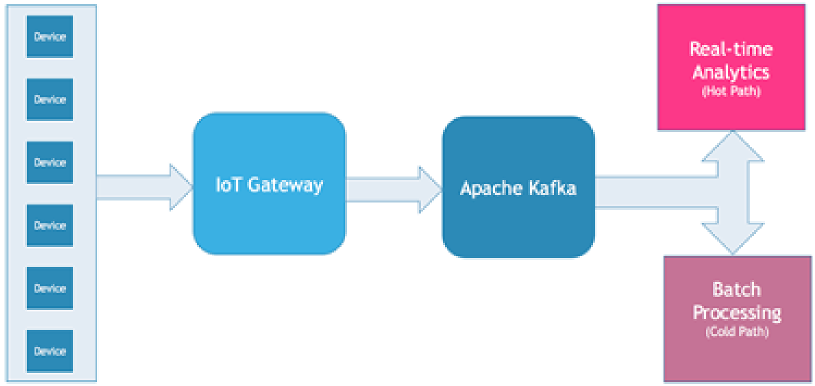
- Lower latency (in RAM, writes only on disk if necessary)
- Data delivered $\Rightarrow$ data deleted
- Scalable
- In order delivery is not guaranteed when using load balancing

7.10.2 Log Based Message Queue

- On disk
- One partition = one consumer (limitation of consumers)
- Data can be reread after delivery
- Scales very high: multiple disks, multiple partitions and nodes
- Message ordering & producer recovery

7.11 Use case: message queue for analytics

- Central hub for batch and stream
- All data is sent to the hub
- You can do both real time analytics (trend analysis) and batch processing at the same time
- Can also contain a database as producer/consumer

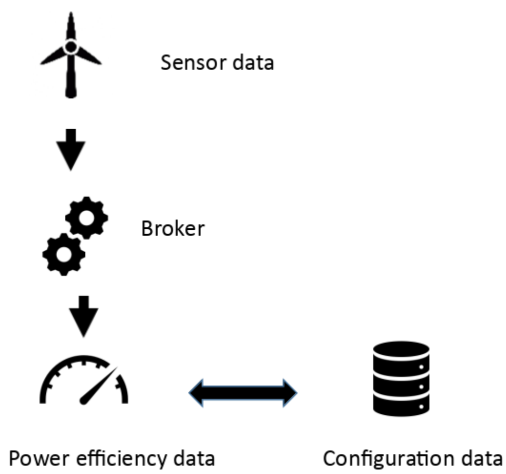


7.11.1 Some other use cases for message queues

- Software integration hub
- Start of Data pipeline for different services (Realtime analytics/'slow' services)
- Use a message broker for streaming data: Ingest/buffering

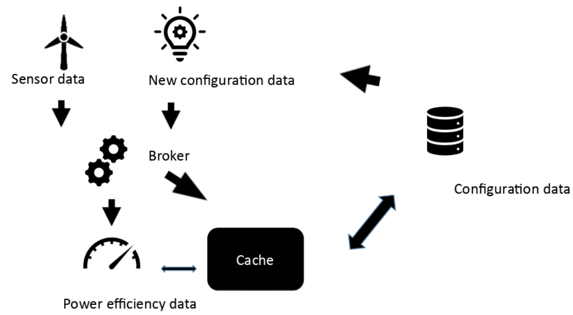
https://www.youtube.com/watch?v=UEg40Te8pnE&t=1510s

7.12 Events & data architecture



Situation:

- You need to process sensor data from a windmill park
- You also need to correlate that data with the configuration data from the windmills (which is on a database)
- **Goal:** train a machine learning algorithm to figure out what configuration is optimal for a certain windmill on a certain location
- **Problem:** if you have a lot of windmills, you would need to make lots of requests to the configuration database



Better solution:

- Sensor data is still sent to the message broker
- To correlate with the configuration data, we will not connect directly to the database
- Instead, we will create a cache (in RAM)
- **Problem:** cache is not always the most recent
- **Solution:** send changes made to the configuration data to the message broker, to update the cache with the new configuration data

7.13 Stream Analytics

Analysis with stream processing is very different than analytics with batch processing: with stream processing, we use 'rolling statistics':

- Multiple ways to do rolling analytics:
  - **Thumbling window** (1:00-2:59, 3:00-5:59)
  - **Hopping window** (1:00-2:59, 2:00-4:59)
  - **Sliding window** (last 2 minutes: 1:04-3:04, = the heaviest calculation)
- Apache Spark, Flink Kafka Streams, Azure Stream Analytics have services to do these calculations

8 Intro to message brokers and Kafka in more detail

The talk is available as a video link on Leho, it is recommended that you watch it:

`https://leho-howest.instructure.com/courses/8864/modules/items/260222`

8.1 Situation

Say you have two microservices A and B, where A sends data to B, to process it.

- A is faster than B
- If B processes data slower than A can send it, B will get overloaded
- Solution: partition the work amongst multiple instances of B
- This is difficult to program properly ⇒ use a message broker

8.2 Message brokers

- Most common model = Publish/Subscribe model
- Popular Message brokers = ActiveMQ, RabbitMQ
- Standard protocols: AMQP, MQTT, STOMP
- It handles new producers/consumers for you
- It removes disconnected/timed-out clients
- If a consumer fails to receive a message, the producer holds on to it for future retransmission, or delivers it to another instance

A message broker also allows you to set the way messages are delivered to your consumers:

- Divided = divide the message to one of the consumers
- All at once = deliver the message to all consumers

8.3 Kafka

Kafka is a logging service

- Kafka is not a message broker, but it has most of the functionality you would commonly expect from a message broker
- 2 main differences:
  - Messages are not removed as soon as they are acknowledged by a consumer
  - A consumer can go back in time to read messages that are older out of the queue
- Kafka works like a Partitioned Log based Message Queue

8.3.1 History

- Founded in 2011 by LinkedIn
- Open sourced and then futher developed by Apache
- Companies like Netflix, Spotify and PayPal use it
- Written in Scala
- Uses its own client protocol
- The official client is in Java, but there are many other stable clients for every programming languages
- Well supported

8.3.2 Sending a message to a partition

- You can let Kafka choose a partition
- You can choose an algorithm to pick a partition for you
- You can choose a partition by the partition number

8.3.3 Kafka message production

- The client contacts the Partition leader and sends the message
- You can choose whether to wait for acknowledgment (synchronously) or to send the message asynchronously
- What if a Kafka node dies?
  - If data was safely persisted to disk, Kafka will recover
  - Kafka only fsyncs after a certain interval $\Rightarrow$ you might lose some data

8.3.4 Kafka replication

- To protect a Kafka node against failure
- New nodes need to be added in increments of 2
- There are many configurations possible to increase the availability

8.3.5 Kafka consumer

- A consumer is always part of a consumer group
- A consumer group has its own offset: it defines how far a consumer has read in the queue

8.3.6 Consumer availability and load balancing

What happens if a consumer in a consumer group dies?

- Each consumer sends a 'heartbeat' to the Kafka node
- Kafka checks that every consumer sends a heartbeat
- If Kafka gets no heartbeat from a consumer, it declares it dead
- It then tells every consumer in the group that a consumer has died, and that every consumer needs to stop consuming

8.3.7 Consuming messages

3 basic steps:

1. You pull a message from the queue
2. You perform all the handling of the message (including handling the result downstream)
3. You mark the message's offset as done

8.3.8 How to handle lost messages

Because of operational events like network outages or component failures, some messages may need to be sent again. In message brokers, there are two policies to handle this:

- At-least-once: it is possible that you can read a message more than once
- At-most-once: it is possible that some messages may get lost

Kafka has an at-least-once guarantee, if correctly used. As a consequence, some messages are sent twice. Kafka solves this by doing deduplication or partial detection. The database will simply overwrite the existing entry if it already exists.

This makes Kafka **idempotent:** for the same input, the output should always be the same, independant of the machine or time.

8.3.9 How far can you go back in time?

Kafka has 2 settings:

- Log retention hours: specifies the amount of hours you want to keep data on disk
- Log retention size: specifies the maximum amount of disk space the partitions may use

By default, Kafka sets the log retention hours to 172 hours (1 week), and does not set a default size limit.

9 Overview data pipelines

A full stack data engineer/scientist has to work with many different pipelines, for example:

- A data pipeline for streaming
- A data pipeline for AI
- ...
- There are also many alternatives to a full blown data pipeline:
  - Linux tools + Python + MySQL/PostgreSQL (for smaller jobs)
  - Pandas/Nifi + Python + Libraries
  - Tick stack for time series
  - ES Stack for text search, geospatial data

9.1 Data pipeline for streaming

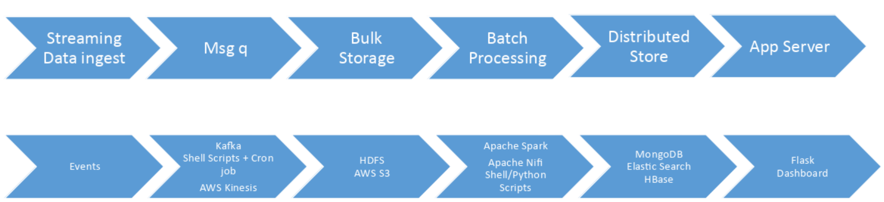


Figure 69: Top: the data pipeline for streaming data, bottom: examples for every step

- Events
  - logs
  - sensor data
  - 'long latency' actions

– Many events are JSON-documents (JSON or JSON lines: to process these, you can use the Python standard library)

- Message queue or collector
  – Collecting data from different sources, timing, formats
  – Example: Kafka-Python
- Bulk storage
  – Scalable, massive and high bandwidth storage
  – But relatively high latency
- Batch processing
  – immutable datasets taht are transformed to useable datasets
  – Possible with PySpark
- Distributed stores
  – To store the results
  – PyMongo, PyElasticSearch
  – Elasticsearch DSL is a high-level library, built on top of the official low-level client
- App server: lightweight simple web application for quick visualisation or prototype
- Flask

⇒ everything is possible with Python!

9.1.1 Batch vs Stream

- Batch if you can, stream if you must
- Stream can be collected in batches
- Kafka: Spark streaming (mini batches)

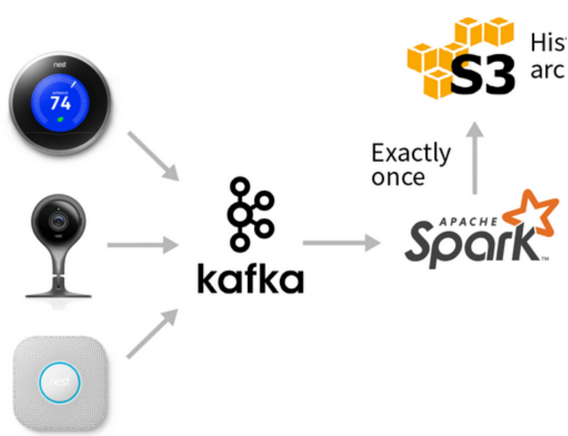


Figure 70

9.1.2 Batch schedule

Apache Airflow is a helpful tool to do batch processing

- To schedule
- To run periodically: daily, hourly, . . .
- Controlled using Python code
- Uses a DAG to create a sequence of jobs
- Similar to Linux's cronjobs
  - But if something goes wrong in a cronjob script, you won't realize until it is too late
  - Airflow has ways to detect if a step fails
- Backfill = command to reprocess older data, for example with a different ML model
- Installed through pip
- `https://en.wikipedia.org/wiki/Apache_Airflow`

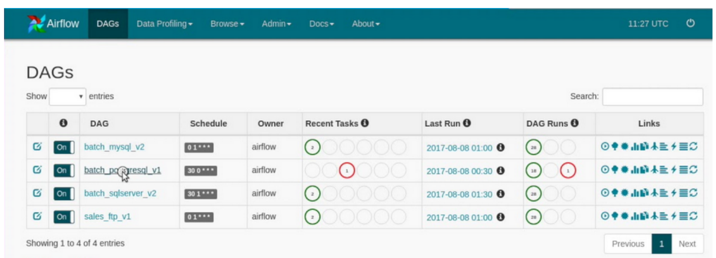


Figure 71: 'Airflow' web interface

9.1.3 JSON vs Parquet

JSON

Javascript Object Notation

- Compressable with gzip
- Document format
- Need and read the complete file
- Processing file with spark:
  - RDD.toJSON().saveAsTextFile(file)
- Supported by everybody
- Used everywhere

Parquet

Apache Parquet is a column-oriented data storage format for the Apache Hadoop ecosystem

- Compressed by default (smaller)
- Columnar format

- Can load only the columns we need
- Processing files with spark:
  - RDD.write.parquet(file)
- Limited support
- Used for performance

9.1.4 SQL

Use SQL to quickly access your batch processing

```
RDD = spark.read.format("csv")

RDD.options(header="true", inferSchema="true")
RDD.load("test.csv")

RDD.registerTempTable("table name")

# quickly use SQL to verify the data has been loaded
Spark.sql("SQL statement from tablename")
```

•

9.1.5 Spark to Elastic Search Cluster

```
RDD = spark.read.parquet("")
# save the dataframe to elasticsearch
RDD.write.format("org.elasticsearch.spark.sql")
   .option("es.batch.size.entries", "100")
```

9.2 Data pipeline for AI

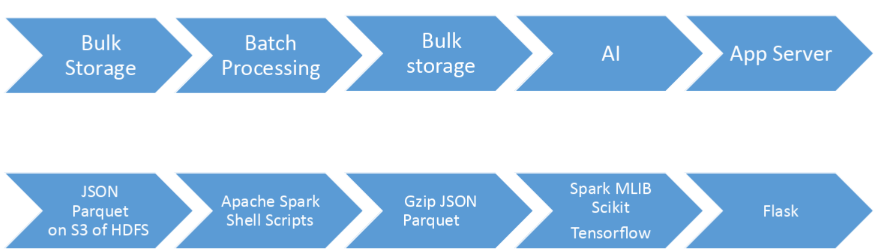
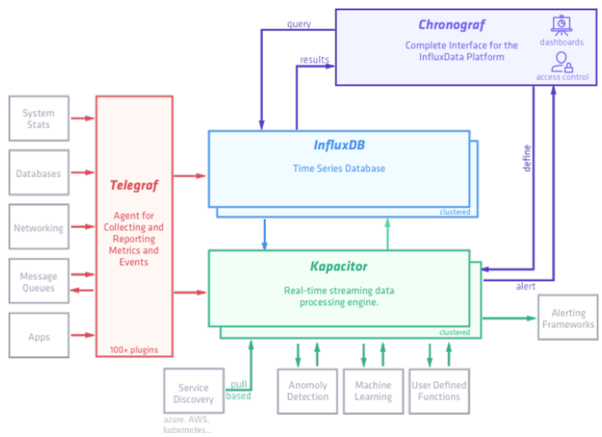


Figure 72: Top: the data pipeline for AI, bottom: examples for every step

9.3 Alternatives

9.3.1 Elastic stack

9.3.2 Tick stack

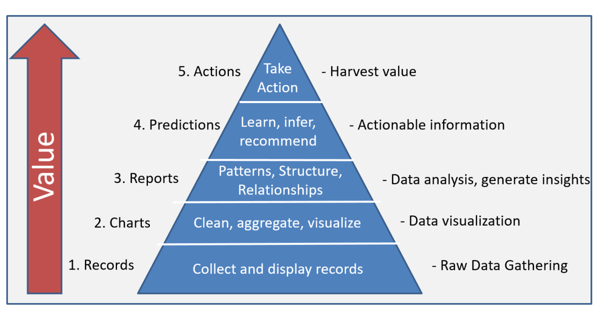


- Only time series
- Dashboarding
- For visualisation and prediction, you need more tools than just the standard Tick stack!

Components:

- **Telegraf:** Plugin-driven server agent for collecting and reporting metrics
- **InfluxDB:** Scalable data store for metrics, events, and real-time analytics
- **Chronograf:** Monitoring/visualization user interface for TICK stack
- **Kapacitor:** Framework for processing, monitoring, and alerting on time-series data

9.4 The data science pyramid



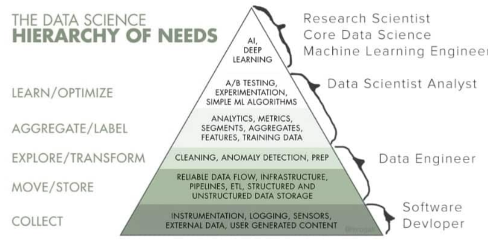


Figure 73: Many jobs in data science!