

Big Data

Tuur Vanhoutte

22 februari 2021

Inhoudsopgave

1 Understanding Data Intensive Applications	1
1.1 Why Big Data?	1
1.1.1 Use case: data intensive application RouteYou	1
1.2 Data Intensive Application: RAMS!	1
1.2.1 Common similar abbreviations	1
1.2.2 Methods to improve Maintainability	2
1.2.3 RAMS applied to RouteYou application	2
1.3 Learning outcome for this module	2
1.4 Scaling	3
1.4.1 MySQL scaling	3
1.4.2 ElasticSearch Scaling: distributed system	3
1.4.3 Professional architecture (Dev oriented)	4
1.4.4 Time series Distributed database (OpenTSDB, InfluxDB)	4
1.5 Scalability & application performance management	4
1.5.1 The need for speed: some insights from Google	4
1.5.2 Response times for websites	5
1.5.3 4 components of network latency	5
1.5.4 TCP Congestion Window - slow start	5
1.5.5 Long tail latency	7
1.6 Conclusion	7
2 Professional storage	8
2.1 Cloud MIPS	8
2.2 Latency vs storage space pyramid	8
2.3 Storage media	9
2.3.1 Magnetic disks	9
2.3.2 Flash (NAND) / SSDs	9
2.3.3 Big difference between read and writing	10
2.3.4 IOPS vs Bandwidth	10
2.3.5 Storage options	11
2.3.6 Performance Conditions	11
2.4 RAID	11
2.4.1 Definition	11
2.4.2 Hardware <> chip	12
2.4.3 Raid levels	12
2.4.4 Caching & BBU	12
2.5 Professional Storage Topology	13
2.5.1 Components	13
2.5.2 DAS - Block storage	13
2.5.3 NAS - File storage	14
2.5.4 SAN - Block storage on a network	14
2.5.5 iSCSI terminology	15
2.5.6 Object storage	15
2.5.7 Link with Databases & other data storage	16
3 Relational databases	16
3.1 Components of a relational database	17
3.2 Reliability problems	17
3.3 Example	17
3.3.1 The problem	18

3.3.2	The solution: Transactions	18
3.4	Single object entry	18
3.5	Concurrency Control	19
3.5.1	Dirty Reads	19
3.5.2	Dirty Writes	19
3.5.3	Read skew	19
3.5.4	Repeatable Read or Snapshot Isolation	19
3.5.5	Multi Version Concurrency Control (MVCC)	20
3.5.6	Lost updates & Atomic updates	20
3.5.7	Write Skew and others.	20
3.5.8	Single Threaded - Serial Execution	21
3.5.9	2-phase lock - Serial execution	21
3.6	Isolation levels	21
3.6.1	Conclusion	22
3.6.2	Isolation level 1: Read Uncommitted	22
3.6.3	Isolation level 2: Read Committed	22
3.7	ACID: Durable	23
3.7.1	Caching & BBU	23
3.7.2	The transaction chain: innodb_flush_log_at_try_commit	23
3.7.3	innodb_flush_method	23
3.7.4	The Transaction chain	24

1 Understanding Data Intensive Applications

1.1 Why Big Data?

1.1.1 Use case: data intensive application RouteYou



Figuur 1: RouteYou

- Routes - user preferences & interests
- Searchable Text data
- Geospatial data
- Community driven
 - Exponential user growth is necessary to make the application possible
 - 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

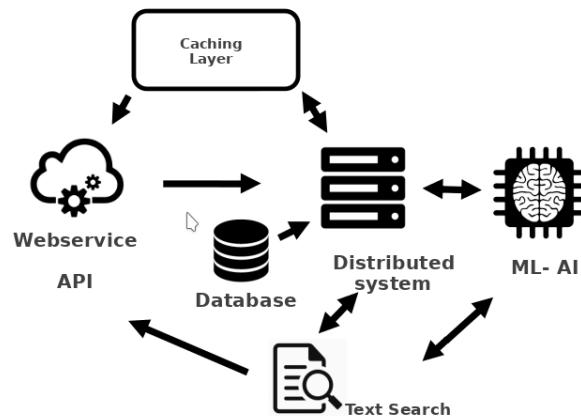
- Infrastructure: RAS (Reliable, Available, Serviceable)
- Developer: RMS (Reliable, Maintainable, Scalable)

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)



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

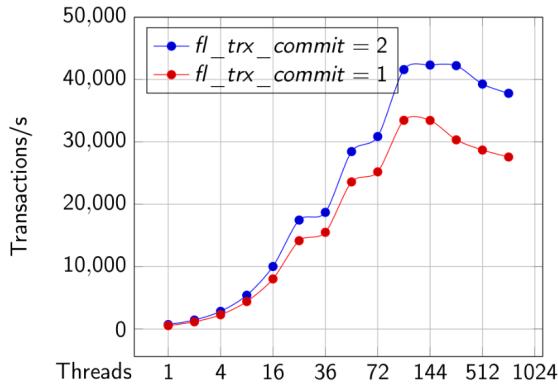
1.3 Learning outcome for this module

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

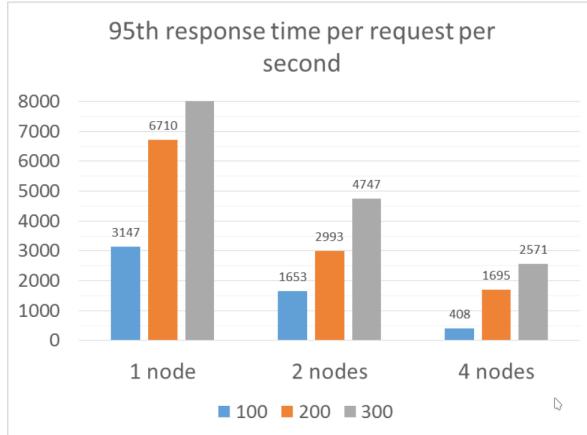
1.4.1 MySQL scaling



Figuur 3: Transactions/sec

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

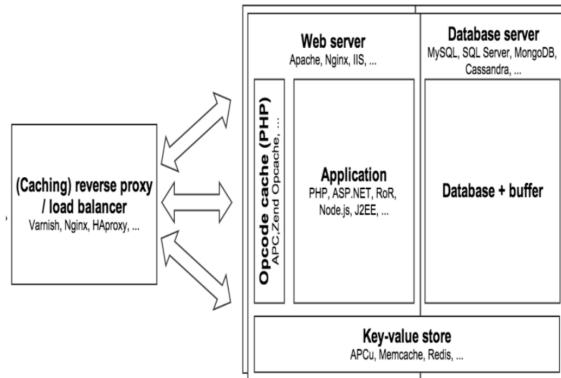
1.4.2 ElasticSearch Scaling: distributed system



Figuur 4: Response time per request

- Scaling out: add more servers to your data system

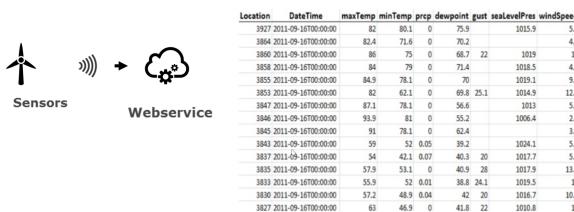
1.4.3 Professional architecture (Dev oriented)



Figuur 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)



Figuur 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

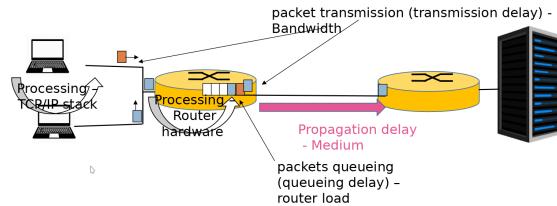
1.5.2 Response times for websites

- **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

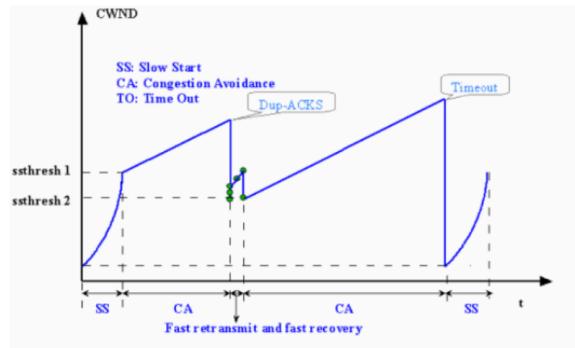


Figuur 7: Network latency diagram

- 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)
- Queuing delay
 - Time spent in router & NIC buffers

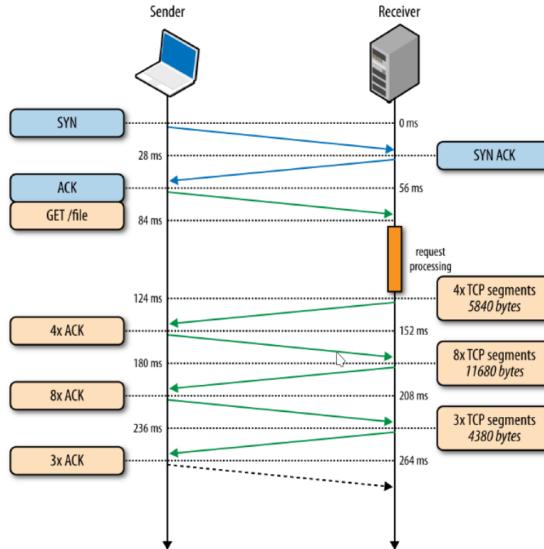
1.5.4 TCP Congestion Window - slow start

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



Figuur 8: TCP Congestion window

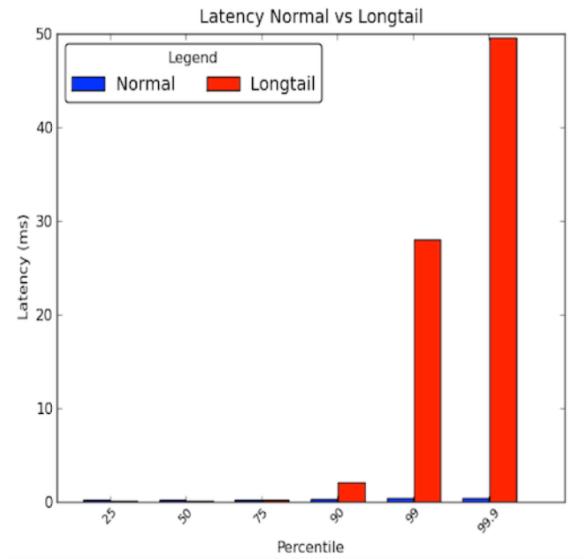
- 4-8-16-32 TCP segments (Win 2008, Win7)
- 10-20-40 (Linux 2.6+, Windows Server 2016 / Windows 10)



Figuur 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



Figuur 10: Long tail latency vs Normal latency

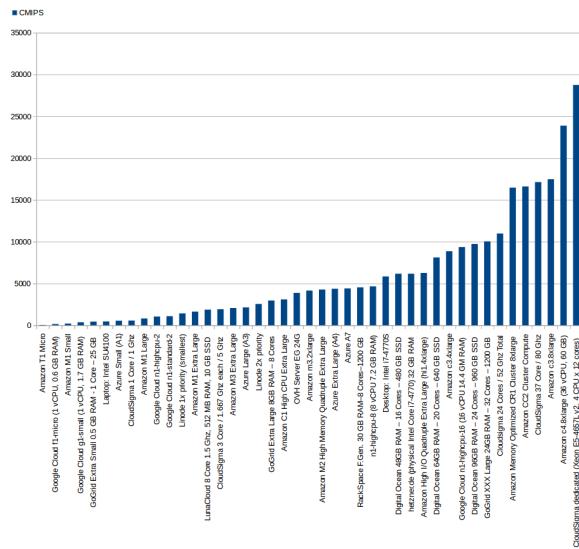
- Average = useless
- Long tail latency = 99th percentile
 - To be experienced by a lot more than 1% of users!
- Best customers encounter highest percentiles
- URL consists of many requests

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

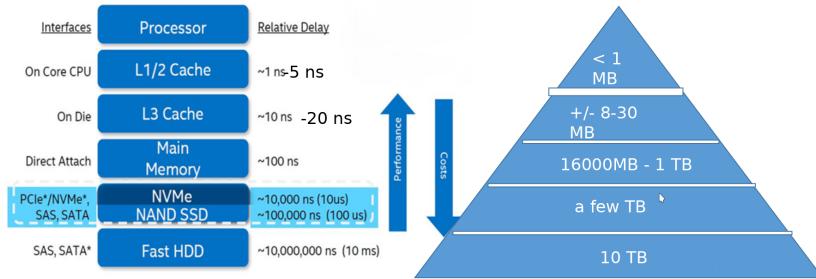
2 Professional storage

2.1 Cloud MIPS



Figuur 11: MIPS = Million Instructions Per Second

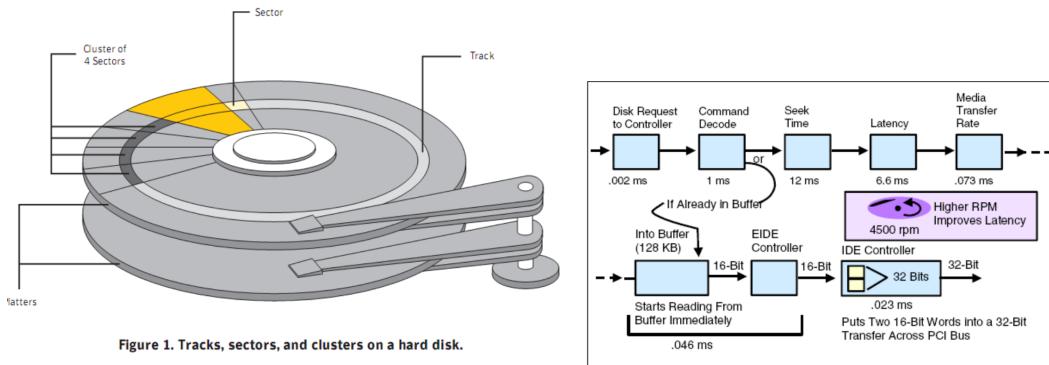
2.2 Latency vs storage space pyramid



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

2.3 Storage media

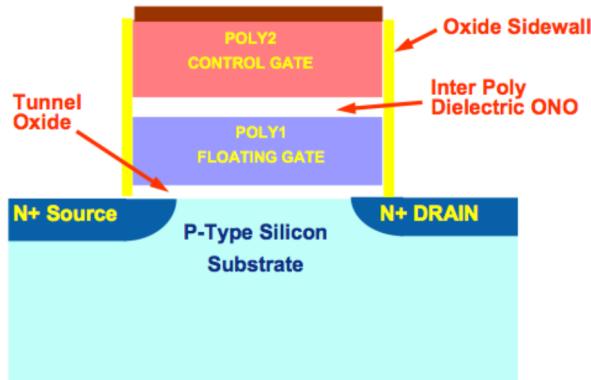
2.3.1 Magnetic disks



Figuur 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



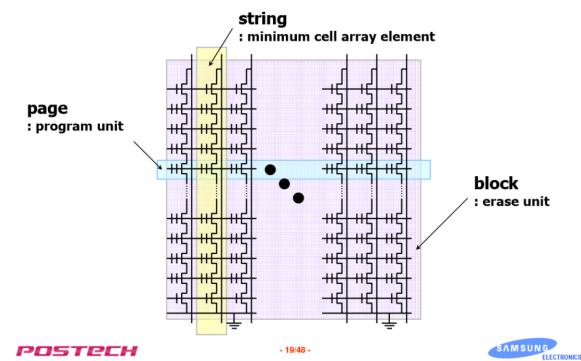
Figuur 14: Flash storage

- 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 ($4\text{KB} * 128 = 512\text{ KB}$)
- You can read or write page per page

- Erasing has to erase the entire block



Figuur 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

Figuur 16

- 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

Figuur 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

Figuur 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

Figuur 19: Performance Conditions

2.4 RAID

2.4.1 Definition

Redundant Array of Inexpensive Disks is a storage technology that combines multiple physical drives into one logical unit.

Purpose:

- Data redundancy
- Performance improvement
- Both

2.4.2 Hardware <> chip

2.4.3 Raid levels

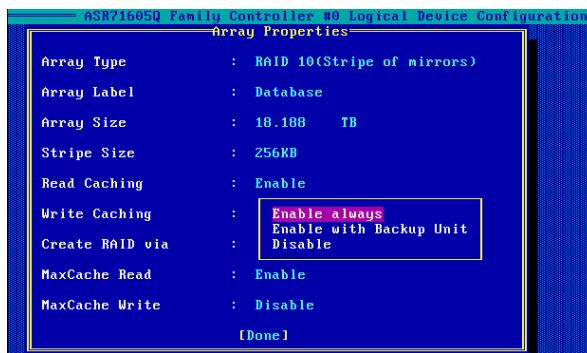
- RAID 0
- RAID 1
- RAID 5
- Combinations are possible (RAID 10, 01, 51, 15)

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

Figuur 20: RAID level choices

2.4.4 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 bandwidth, +- same latency
 - Latency does not increase as fast when load increases (vs single disk)
 - More bandwidth & capacity available



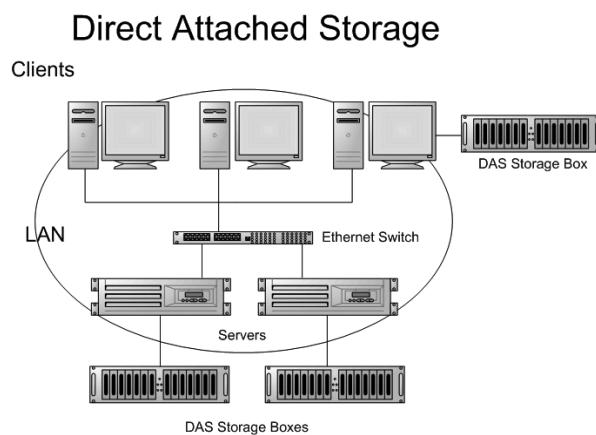
Figuur 21: RAID configuration

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)

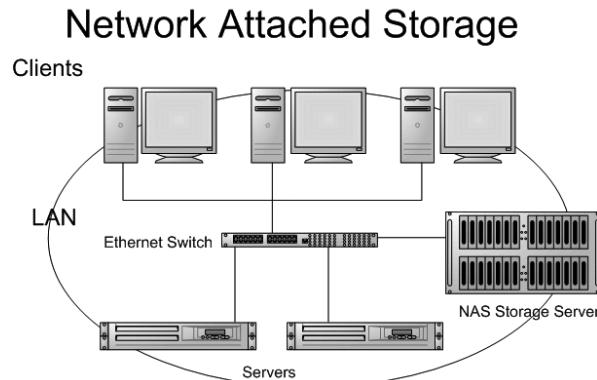
2.5.2 DAS - Block storage



Figuur 22

- Up to 122 disks per SAS controller
- Similar to disks inside the server
- No centralized back-up

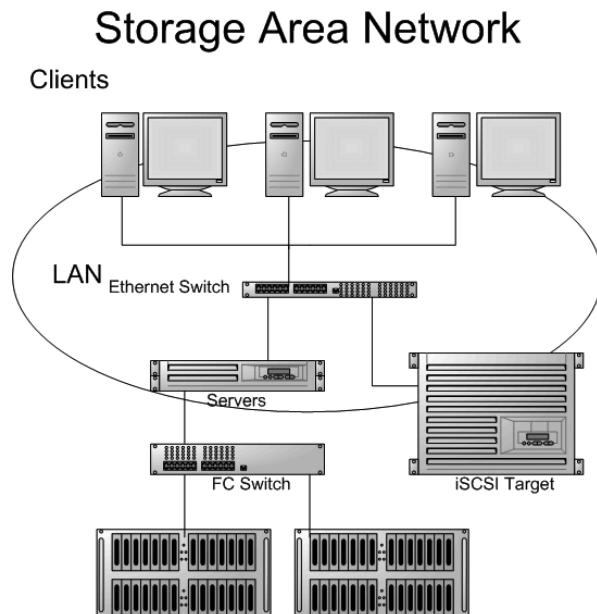
2.5.3 NAS - File storage



Figuur 23

- 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



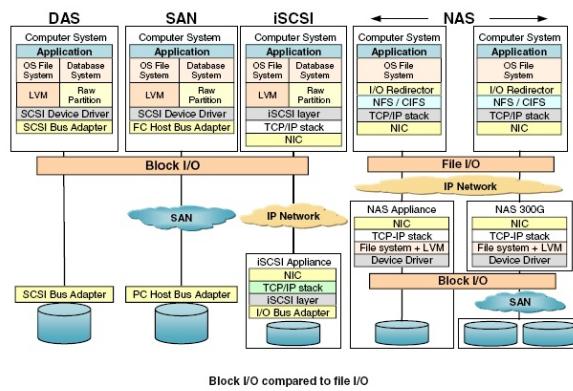
Figuur 24

- Separate 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

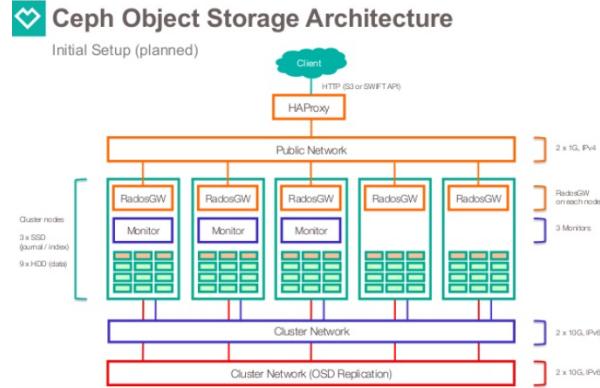
Storage characteristic	iSCSI SAN	Fibre Channel SAN	NAS
Protocol	Serial SCSI	Fibre Channel Protocol	NFS, CIFS
Network	Ethernet, TCP/IP	Fibre Channel	Ethernet, TCP/IP
Source / target	Server / Device	Server / Device	Client / Server or Server / Server
Transfer	Blocks	Blocks	Files
Storage device connection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

Figuur 25

2.5.6 Object storage

- NAS hardware
 - Distributed over multiple datacenters
- Object Data
 - Metadata
- Globally Unique Identifier

- URL
- RESTful API
- Examples:
 - AWS S3
 - Ceph - Lustre
 - Google Cloud storage



Figuur 26

2.5.7 Link with Databases & other data storage

- Transactional database: needs block storage
 - Performance
 - Durability
 - Consistency
- Block storage best for ‘raw data’ (no meta data involved)
- NAS = ‘file based’ services like sharepoint
- 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 Animax 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

Figuur 27: 1 relation 'student': 20 tuples, 8 attributes

3.2 Reliability problems

- Applications crash
- Client (website) - network - database
 - ⇒ network is very unreliable
- Multi-threaded code: race conditions ⇒ 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
- **Consistent**
 - 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

3.5.1 Dirty Reads

Definitie 3.1 (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.*

Solutions:

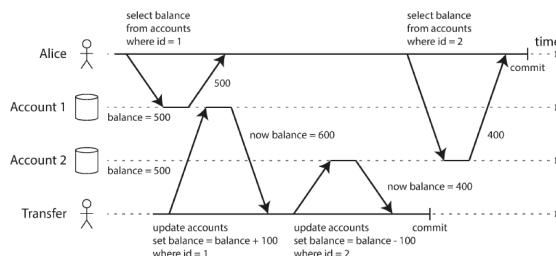
- Read locks (=very bad performance)
- Remember the old value until commit

3.5.2 Dirty Writes

TODO

3.5.3 Read skew

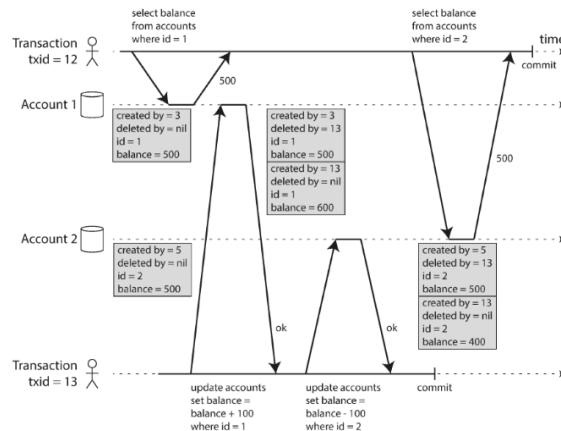
- TODO



Figuur 28: Read skew

3.5.4 Repeatable Read or Snapshot Isolation

- Which transaction changes what?
 - TODO
- "Multi Version Concurrency Control (MVCC)"
- Readers do not block writes, writers do not block reads



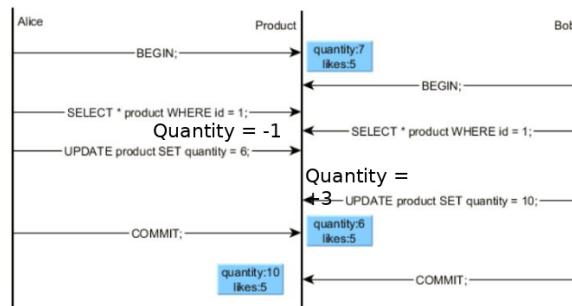
Figuur 29

3.5.5 Multi Version Concurrency Control (MVCC)

TODO

3.5.6 Lost updates & Atomic updates

- Two read - modify - write transactions
 - Problem: Bob reads 'quantity' while transaction is committing
- Solutions: 'atomic updates' or manual lock
 - No reads or update object until commit
 - Update 'X' SET value = "X2"
 - (Read - modify - write in one operation)



Figuur 30

3.5.7 Write Skew and others...

- Multi object updates & lost updates
- TODO

```

BEGIN TRANSACTION
SELECT COUNT(*) FROM bookings WHERE seat = 'front row'
INSERT INTO bookings ...          -- count > 0
COMMIT;

```

3.5.8 Single Threaded - Serial Execution

- Serial instances for reads
 - One fast write thread in RAM
 - Select on separate snapshot
- Examples:
 - Redis
 - VoltDB
- Complete transaction in one serial stored procedure
 - = piece of code in the DB, already compiled, that is ready to be executed
 - Upsides: TODO
 - Downsides: TODO
- Partition data
 - Multiple threads, one thread per partition
 - Cross partition transaction? Hard!

3.5.9 2-phase lock - Serial execution

- Weak isolation levels:
 - Readers never block writers, and you can read the old value while it is written
- 2PL = exclusive lock on read data, then exclusive lock for writing
 - Deadlocks are possible! (= transactions keep waiting for each other's locks)
- Examples:
 - * MySQL InnoDB
 - * SQL server
 - * DB2 ("Repeatable read")

3.6 Isolation levels

- = Choose between strong isolation or strong performance
 - Modern processing 8 - 100+ threads
 - Choose an isolation level:
 - 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!	

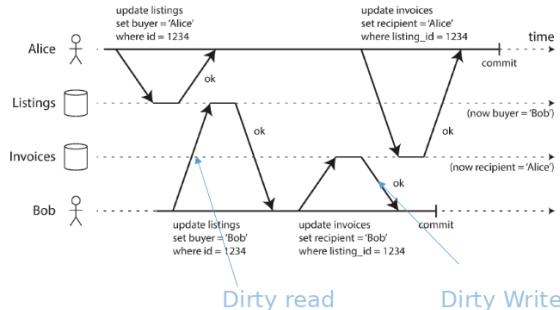
Figuur 31: (default) isolation levels in current databases. (*) Wrong, Oracle does not comply with ANSI

3.6.1 Conclusion

- TODO

3.6.2 Isolation level 1: Read Uncommitted

- Read Uncommitted offers no protection against dirty writes & dirty reads
- Fastest performance, lowest isolation



Figuur 32: Dirty read & dirty write example

3.6.3 Isolation level 2: Read Committed

- No dirty reads
- No dirty writes

TODO

Solutions

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

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.

3.7.1 Caching & BBU

- Write through / back TODO
- Battery Backup Unit
- RAM caches in SSDs have to be Non-Volatile (NV)!

3.7.2 The transaction chain: innodb_flush_log_at_try_commit

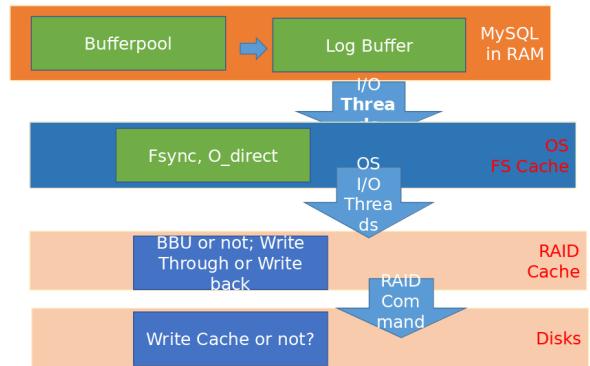
- 0: Write the log buffer to the log file and flush the log file **every second**, but do nothing at transaction commits (fastest)
 - Fastest
- 1: Write the log buffer to the log file and flush it to durable storage **at transaction commits**
 - This is the only option that is fully ACID compliant
-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

Flushing = writing to disk

3.7.3 innodb_flush_method

- 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
 - The exception: it uses buffered I/O to the log files when innodb_flush_log_at_try_commit = 2

3.7.4 The Transaction chain



Figuur 33: The transaction chain when writing to disk

TODO