

263-3855-00: Cloud Computing Architecture

Section 06: Availability and Reliability

Swiss Federal Institute of Technology Zurich (ETH Zürich)

Last Edit Date: 05/31/2025

Disclaimer and Term of Use:

1. We do not guarantee the accuracy and completeness of the summary content. Some of the course material may not be included, and some of the content in the summary may not be correct. You should use this file properly and legally. We are not responsible for any results from using this file.
2. This personal note is adapted (derived) from the slides and codes from *Professor Ana Klimovic and people from ETH Zurich Systems Group*. Please [contact us](#) to delete this file if you think your rights have been violated.
3. This work is licensed under a [Creative Commons Attribution 4.0 International License](#).

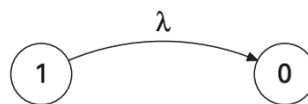
I. Basics of Availability and Reliability

1. Reliability

- Reliability is the ability of a system to remain operational over time, i.e., to offer the service it was designed for.
- Reliability is influenced by many factors
 - Errors: the system is operational but produces wrong results, for example, a memory glitch flipping bits
 - Faults: the system is operational but some parts of it might not be, for example, one of the disk fails
 - Failures: the system is no longer operational, for example, system crashes
 - Performance issues: the system is working but it is too slow to be useful
- Reliability is typically measured in terms of system “availability”
- Availability: time the system is working / total time

2. Mean time to failure

- During an interval of time T , we observe F failures
- Frequency of failures (also called failure rate) = F / T
- Meant time to failure (MTTF) = T / F



A single component with failure rate λ

- The definition above does not include repairs
 - Assume failures are independent (an earlier failure is uncorrelated with the next failure)
 - Consider the component in isolation (ignores failures caused by other components)
 - Thus, the MTTF is an upper bound, real failures are likely to happen more often

Example:

A system can have two types of failures A and B

- Failures of type A happen at a rate of 30 every 1000 years
- Failures of type B happen at a rate of 20 every 1000 years
- Both are independent of each other

The system fails if either A or B happen

- So the rate at which either A or B happen is $30 + 20$ every 1000 years
- $MTTF = 1000 / 50 = 20$ years, failure rate is 0.05 per year

3. Mean time between failures

- If we also model the time it takes to repair a component, what is relevant is the Mean Time Between Failures (MTBF), which is $MTBF = MTTF + MTTR$
- MTTR is the mean time to repair or the time between a failure occurring and the moment the system is operational again



- Availability = Time system is working / total time = $MTTF / (MTTF + MTTR)$
- Failure rate = $1 / (MTTF + MTTR) = 1 / MTBF$

4. Availability

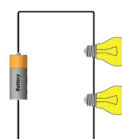
Example:

- A 2400-server cluster has the following events per year:
 - Cluster upgrades: 4 (need to fix)
 - Hard drive failures: 250 (need to fix)
 - Bad memories: 250 (need to fix)
 - Misconfigured machines: 250 (need to fix)
 - Flaky machines: 250 (need to reboot)
 - Server crashes: 5000 (need to reboot)
- Assume time to reboot software is 5 minutes, time to fix hardware is 1 hour
- Hours of outage = $(4 + 250 + 250 + 250) * 1 \text{ h} + (250 + 5000) * 5 \text{ min} = 754 + 438 = 1192 \text{ h}$
- Hours in a year = $365 * 24 = 8760$
- Availability = $1 - 1192 / 8760 = 86\%$
- This for the cluster as a whole: application that need everything to work, will see an availability of 86% (very low)
- If applications only use part of the cluster, the availability they will see will be higher depending on how much “real estate” they need

Uptime	Downtime in one year
99% (two 9's)	87.6 hours
99.9% (three 9's)	8.76 hours
99.99% (four 9's)	53 min
99.999% (five 9's)	5 min
99.9999% (six 9's)	32 sec
99.99999% (seven 9's)	3 sec

Availability of multi-component cloud services

Consider a system with N components. There are different cases:



- Case 1: All N components are necessary for the system to be functional
- Case 2: Redundant N components, only 1 of N is necessary for the system to be functional
 - We cannot act on the failures of individual components. When the entire system stops working, we can fix all its components, or
 - We can detect individual component failures and we can fix a failed component while the system is still running

Case 1: system fails if any of the N components fail

Assume a system has N components. Each component has a failure rate of r failures a year. For now assume no repairs.

- $MTTF_{component} = \frac{1}{r}$
- $MTTF_{system} = \frac{1}{N \cdot r} = \frac{MTTF_{component}}{N}$
- $Failure\ rate_{system} = N \cdot r$

Example: a system has N components and fails if any component fails. Assume every component has a $MTTF_c$ of 25 years. Then for the whole system:

- $MTTF_{system} = MTTF_c / N = 25 / N$
- For $N = 5$, $MTTF_{system} = 5$ years, much smaller than the individual component

Takeaway: to make a system with many components work, either ensure each component is highly reliable or compensate with redundancy. In the cloud, everything is built out of sets of components and the cloud often uses commodity, unreliable hardware. Need redundancy to minimize system failures.

What if: Assume a system has N components. Each component has a failure rate of r failures a year. Each failure takes x hours to be repaired and we can repair 1 component at a time.

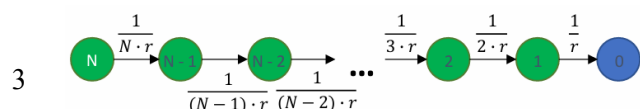
- $MTBF_{component} = \frac{1}{r}$
- $MTBF_{system} = \frac{1}{N \cdot r} = \frac{MTBF_{component}}{N}$
- $Failure\ rate_{system} = N \cdot r$
- $Availability_{system} = 1 - \frac{N \cdot r \cdot x}{365 \cdot 24}$

Note that the lower bound of availability is 0.

Case 2a: only 1 of N components is necessary

We do not fix one component failure at a time. We repair the system in x hours when all N components fail. Assume the per-component failure rate in this system is r failures a year.

- $Failure\ rate_{system} = \frac{1}{MTBF_{system}}$



- $MTBF_{system} = \sum_{i=0}^{N-1} \frac{1}{(N-i) \cdot r}$
- $Availability_{system} = 1 - \frac{failure\ rate_{system} \cdot x}{365 \cdot 24}$

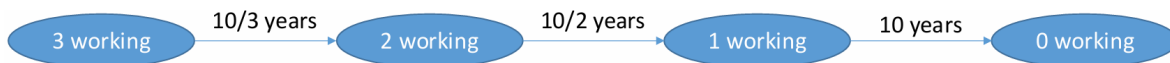
Example: We have 2 components with a $MTTF_C$ of 10 years. Do not consider repairs. The system is built using both components in such a way that if one fails, the other one takes over (e.g., a backup system).

The $MTTF_{system}$ is as follows:

- We start with the two components working
- The time until any of the two fails is $10 / 2 = 5$ years
- After one fails the other one continues working with a MTTF of 10 years (Markov model, assuming memoryless distributions)
- Thus, $MTTF_{system} = 5 + 10$ years



What if: Instead of 2, consider 3 components, each with an MTTF of 10 years. Triple module redundancy is when the system works as long as 2 or more work.



$$MTTF_{system} = 10 / 3 + 10 / 2 = 8.3 \text{ years}$$

- Less than MTTF of one component because we need 2 to operate
- More than MTTF of 2-component system because the third one provides an alternative in case of 1 failure

It allows to upgrade the system without turning it off

Case 2b: only 1 of N components is necessary, but we fix components as they fail

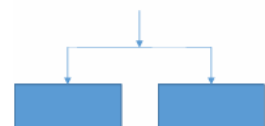
Each component has a failure rate of r failures a year. A failure takes x hours to be repaired. Assume we fix components as they fail and can fix components in parallel.

- $Availability_{component} = 1 - \frac{r \cdot x}{365 \cdot 24}$
- $Availability_{system} = 1 - (1 - Availability_{component})^N = 1 - (\frac{r \cdot x}{365 \cdot 24})^N$

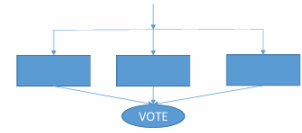
5. System configurations

There are many different configurations depending on what we want to achieve:

- Parallel system with redundant components tolerates failure of 1 component
 - If both active, capacity doubles, one failure allows the system to continue at lower capacity



- If one inactive (passive backup), one failure allows the system to continue but the capacity is that of a single system
- Redundant components used to decide based on a majority. Works as long as 2 out of 3 work. Tolerates 1 failure but can detect errors by comparing results.



Parallelization vs. reliability trade off

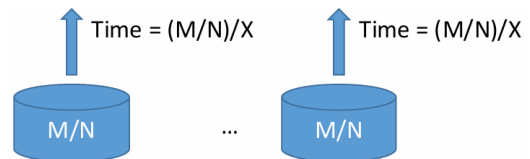
Processing rate in one component = X . Amount of data = M .

- $\text{Time} = M / X$
- $MTTF = MTTF_{\text{component}}$



If using parallelization

- $\text{Time} = (M / N) / X \rightarrow$ system is faster as N increases
- $MTTF_{\text{system}} = MTTF_{\text{component}} / N \rightarrow$ probability that 1 node is down increases with N



If no other measures were taken, it would be impossible to scale out systems.

From MTTF to “Best Effort”

- Many systems work on what is called a “Best Effort” guarantee:
 - They will try their best to work correctly but there is no guarantee
- Example: search engines
 - Data from the search domain is split across many machines to increase read bandwidth and processing capacity
 - If all machines are working, results are complete and reflect all the available data
 - If one or more machines fail or are not working at the time of the query, there are two options
 - Do not provide an answer $\rightarrow$ not good, would happen too often
 - Provide an answer based on what works $\rightarrow$ best effort

II. Errors, faults, and failures

Error: we get a result but it is incorrect (system is “working,” but not as specified)

- Should be detected
- Typically corrected by the system

Fault: some part of the system is not working and, as a result, some of its functionality might be compromised

- Should be detected

- Typically compensated through redundancy

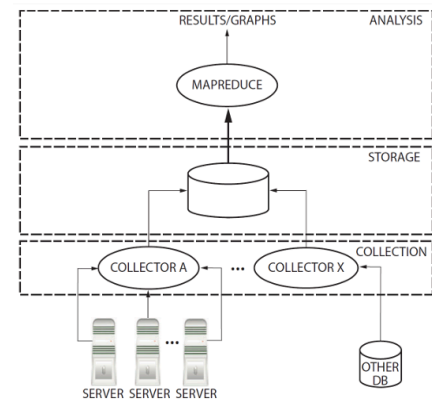
Failure: system is not working

- A range of situations that require different solutions, and each situation might be different and requires a different approach. The recovery procedure or remedial action will also be different (restart, reconnect, add more copies, recovery from backup, etc.)
 - Degraded performance (e.g., system is too slow)
 - Unstable behavior (e.g., sometimes work, sometimes it does not)
 - Service is not reachable (may be working but we cannot see it)
 - Service is down but failed cleanly
 - Service is down and data has been corrupted or lost

1. Detecting and predicting failures

A typical infrastructure in many cloud systems involves

- Data collection at every machines (configuration, activity, environment, errors)
- Data gathering and storage (typically as a time series), combined with additional information
- Analysis tools to identify patterns, problematic machines, and ML tools to recommend repair actions and even to predict future problems



2. Tolerating and hiding faults

- Fault tolerance increases availability by hiding problems from the user
- Must be implemented at all levels and for all components: the system is as strong as its weakest component
- It includes networks, power supplies, data centers, storage, etc.

III. Mechanisms for fault tolerance

1. Replication

Replication is widely used for

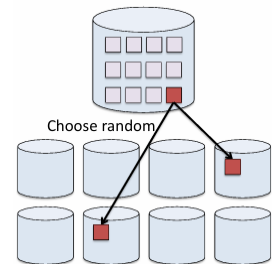
- Availability
- Performance

Several forms of replications

- Data replication
- Error correcting / detecting codes
- Sharding (not exactly replication but often combined with replication)

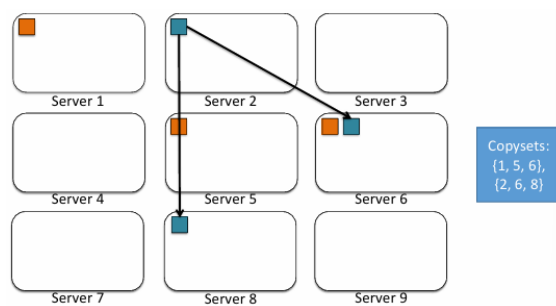
How to place data copies across nodes?

- Idea: random replication, for example, spread replicas across 3 randomly chosen servers
- Is random a good idea? Yes, if independent failures. But data centers often have correlated failures
 - Rack power failure
 - Network failure



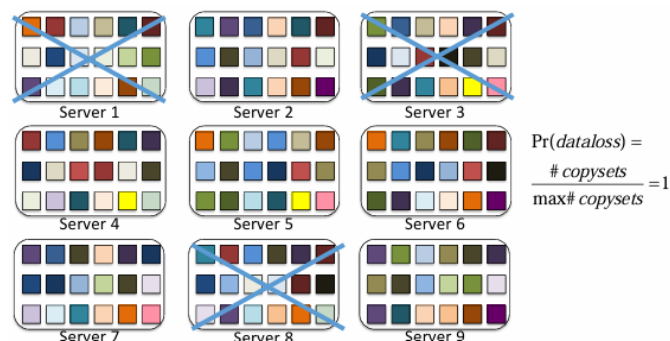
Copyset replication

Copyset: set of nodes containing all copies of a data chunk. Single unit of failure: if loose copyset, loose data.



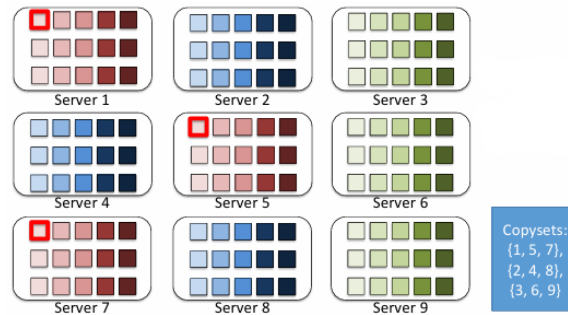
Random replication

Random replication eventually creates a maximum number of copysets. Any combination of 3 node failures leads to data loss.



MiniCopysets

Minimize the number of copysets and make data failure rare. Randomly place first chunk, place replicas deterministically in the copyset.



Data loss trade-off

We minimize the probability of data loss, but every time we lose data, we lose more data. For example, we have a 5000-node cluster, and power outages occur every year. Then:

	MinCopysets	Random Replication
Mean Time to Failure	625 years	1 year
Amount of Data Lost	1 TB	5.5 GB

Recovery time trade-off

Copysets are much slower because the work of reading copies is not equally distributed across all nodes but across the copysets -> there is less bandwidth available to read the data to recover a failed node.

Eventual consistency

- Eventual consistency is guarantee that a data update in a distributed system will eventually be reflected in all nodes that store the data
- Eventually consistency boosts both performance and availability:
 - Better performance because there is no need to update all replicas
 - Better availability because failures of one or more replicas can be ignored and continue working with the replicas that are working
- Downsides
 - The application must be able to operate with inconsistent data
 - The system must be brought to a consistent state at a certain point (inconsistency is tolerated only for short periods of time)

Load balancing

- Load balancing goes well with sharding
- Ability to balance load depends on the nature of the jobs and data distributions
 - Micro-service and micro-shard tendencies go together
 - The smaller the components, the more we can balance and distributed the load

Redundant execution and tail latency

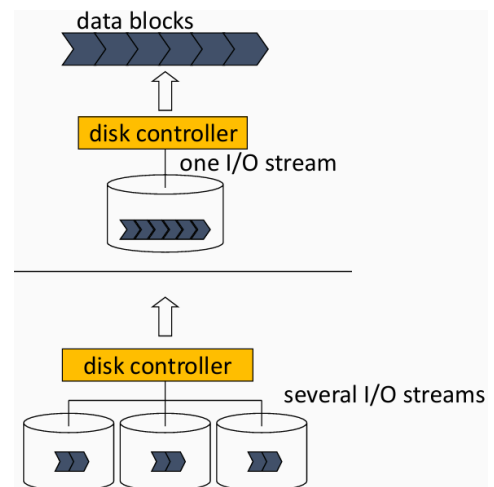
- Many computations in the cloud are highly parallel. They suffer from
 - Stragglers: machines that are slow and slow down the entire computation
 - Failures: machines that fail and do not complete the assigned task
- A common strategy is to use redundant execution
 - Send each job to two machines
 - As soon as one returns the result, kill the second job
- Helps with
 - Failures: the probability that both jobs fails is much smaller
 - Tail latency: the probability that the two jobs hit a straggler is smaller

Monitoring

- Many other techniques involved in maintaining availability
 - Containers: a large scale parallel job that has a bug can trigger a massive failure; run the job in a “canary” and only run at scale if it has no problems
 - Watchdog timers: regularly check the responsiveness of a machine to identify the ones that may cause trouble
 - Integrity checks: regular integrity checks that complement standard techniques in hardware to make sure that data is not lost or corrupted; run by background jobs
- Need good logging and analysis tools
 - ML is often applied today to better understand problems and help predict them

2. Data striping

- Distribute data among several disks so that it can be accessed in parallel
- Data striping implemented at a low system level (not user driven; different than sharding or replication)
- Reasons for striping
 - Increase disk bandwidth by concurrently retrieving data from several disks
 - Decreases seek time (all disk do the seek in parallel)
 - Handle several disk requests in parallel
- Implemented in RAID disk arrays



Fine grain vs. coarse grain

Fine grain data striping:

- Use small data blocks so that all requests serviced using all the disks at the same time

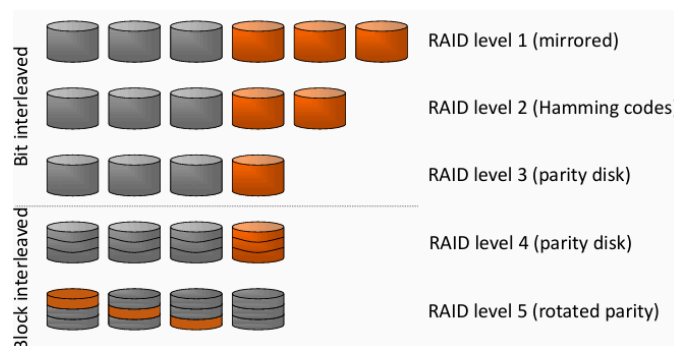
- Maximize the disk bandwidth (data transfer)
- The penalty for positioning the disk heads for each request must be paid for every request since requests are dealt with sequentially and all disk are used for every request
- Only one logical I/O request can be serviced at a time

Coarse grain data striping:

- Use large data blocks so that
 - Small request can be serviced in parallel since they will access only a few disks
 - Large request can still benefit from high transfer rates by using many disks
- For small requests, the seek penalty is not sequential since several disks are used at the same time

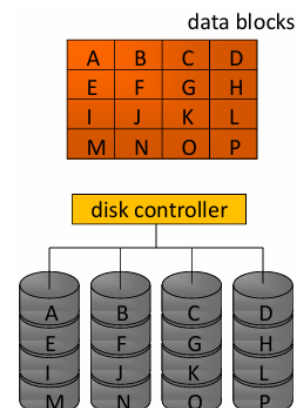
Fault tolerance in disk arrays

- Disk array uses many independent disks. Deal with failures in disk array by using redundancy and / or mirroring
 - Parity information is used to both detect and correct disk errors
 - Mirroring is based on replication of data blocks
- The striping and parity depends on the block size
 - Bit-interleaved: each block is one bit, for example, a byte can be stored in 8 disks, parity is then on a per-byte basis
 - Block interleaved: each block contain several bytes (up tp KBs), parity is then on a per block basis
- Different combinations of parity and striping units lead to different “RAID levels”



3. RAID level 0

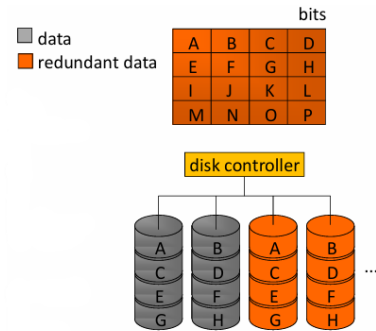
- A RAID level 0 strips the data across the disks but without redundancy
- The data is divided into blocks (of arbitrary size) and the blocks are uniformly distributed across the disks
- I/O bandwidth increases (N times that of a single disk) by using multiple disk channels in parallel for reading / writing



- A failure in one of the drives makes the entire RAID unavailable (no fault tolerance)
- Easily implemented in software
- No constraints on the number of disks

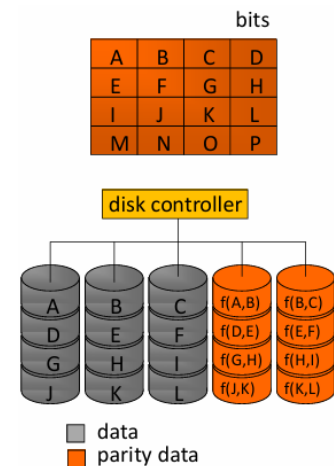
4. RAID level 1

- Bit interleaved
- Fault tolerance by mirroring (no parity)
- Read operations are performed on the copy that offers the shortest seek time
- Two read operations can be performed in parallel. Writes are sequential
- 50% of the disk capacity is used for redundancy purposes. I/O bandwidth is half of RAID 0 ($N/2$)
- Recovery from failures is trivial
- Requires at least two disks (and twice as many as RAID level 0)
- It can handle multiple disk failures (as long as they are not on a mirrored pair)



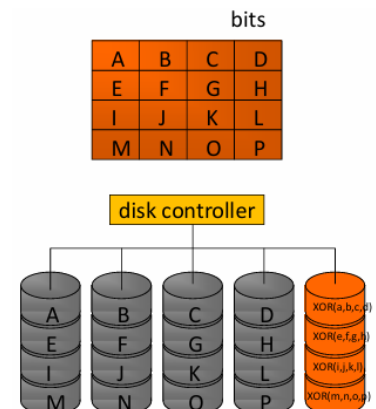
5. RAID level 2

- Bit interleaved
- Fault tolerance by mirroring based on Hamming codes
- Hamming codes implement parity for overlapping segments of data. They require less space than full mirroring but must be implemented in hardware
- Recovery is more complex (depends on the parity of several segments)
- I/O bandwidth is $(N - \log N)$ with $\log N$ being the number of disks needed for storing the parity
- It can handle multiple disk failures (depending on the failures)



6. RAID level 3

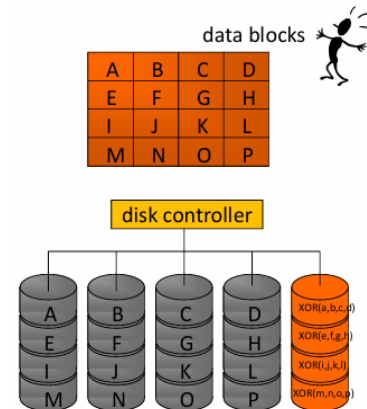
- Bit interleaved
- There is a disk devoted to store the bit-wise parity of the other disks
- I/O bandwidth much better than levels 1 and 2 ($N - 1$)
- It can only handle one request at a time (no parallelism)
- Recovery is relatively simple (use parity to restore the data)



- Tolerates one disk failure
- This is fine grain striping (adequate for applications that use large files, e.g., multimedia)

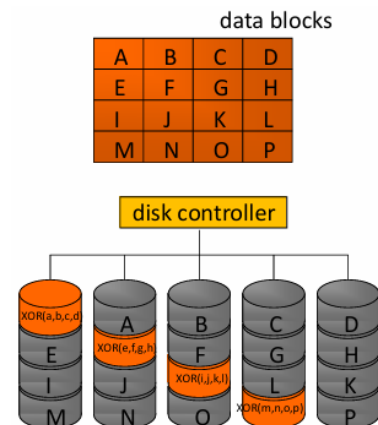
7. RAID level 4

- Block interleaved (blocks of arbitrary size, the size is called the striping unit)
- There is a disk devoted to store the block-wise parity of the other disks
- Write operations are sequential (all of them need to update the parity disk)
- Read operations can be done in parallel when on different blocks
- Parity disk is not used in read operations (limiting bandwidth)
- Tolerates one disk failure
- This is coarse grain striping (adequate for standard databases with few update operations)



8. RAID level 5

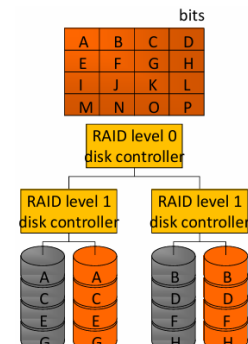
- Block interleaved (blocks of arbitrary size, the size is called the striping unit)
- The block-wise parity is uniformly distributed across all disks
- Write operations can be done in parallel
- Read operations can be done in parallel when on different blocks
- Tolerates one disk failure, recovery is somewhat complex
- Overall good performance
- Small writes can be quite inefficient (because they require to read other blocks to complete the parity)
- RAID 6 is almost the same as RAID 5 but with an additional parity block calculated on different blocks



9. Nested RAID: RAID 1 + 0 and RAID 0 + 1

RAID level 1 + 0:

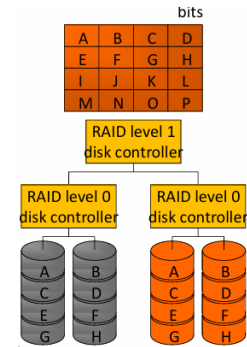
- RAID level 1 + 0 uses a RAID level 0 controller to stripe the data. Each striping unit is then mirrored by a RAID level 1 controller
- Same fault tolerance as RAID 1
- Requires at least 4 disks



- I/O bandwidth can be slightly better than level 1 because the level 1 controllers have less disks to manage

RAID level 0 + 1:

RAID level 0 + 1 uses a level 1 controller for mirroring the data and level 0 controllers for striping the mirrored disks.



10. Sharding or partitioning

- Sharding involves splitting dataset into smaller pieces (the shards) and placing them across different machines:
 - Operations can be parallelized by submitting a request to the shards involved and then merging the results -> improves response time
 - Different requests can be parallelized if they can be served from different shards -> improves throughput
- If shards are replicated, they can be used for recovery (like RAID)
- Micro-shards (very small shards) can be very useful in many applications for load balancing across many machines.

Big shards vs. small shards

