Lecture 22: Data Reliability and Modern Storage Systems

(RAID, Crash, FSCK, Journaling, Distributed Storage)

Xin Liu

Florida State University
xliu15@fsu.edu

COP 4610 Operating Systems
https://xinliulab.github.io/FSU-COP4610-Operating-Systems/

Review

- File system implementation: Data structures on `bread`/`bwrite`
  - `balloc`/`bfree`
  - Files: FAT (linked list) / UNIX File System (indexing)
  - Directory files

Overview

Questions:

- To achieve high speed, we can tolerate data loss in volatile memory (volatile) due to power outages.
- However, data stored on storage devices (persistent data) must not be lost!
    - How can we ensure the reliability of persistent data?

Main topics of this lecture

- RAID (Redundant Array of Inexpensive Disks)
- Crash consistency
    - File System Checking
    - Journaling
- Modern Storage Systems
    - Capabilities and Limitations of File Systems
    - Relational Database
    - Distributed Storage

Redundant Array of Inexpensive Disks (RAID)

Growing Demand for Persistent Storage

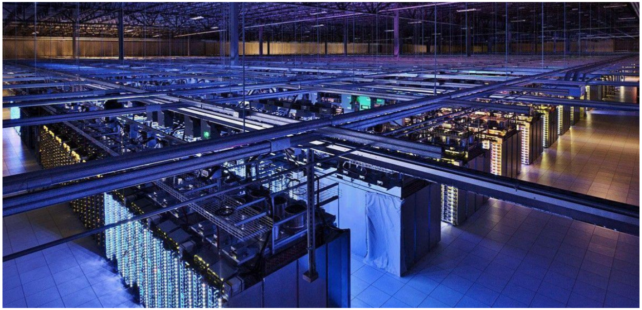
The storage device industry: As long as the CPU (DMA) can handle it, we can provide sufficient bandwidth!

- The tradition of the computer system "industry" — creating practical, efficient, reliable, and cost-effective systems.

EMC VNX5300 Unified Storage (Disk) Array

Performance VS. Reliability

- Any physical storage medium has the potential to fail.
  - Extremely low-probability events:
    - Earthquake, war, alien invasion
  - Low-probability events: Hard drive failure
    - Large-scale redundancy = inevitable occurrence
- But we still hope the system continues running (data integrity even when storage devices fail)


Google Data Center

RAID: Virtualization of Storage Devices

So, can we achieve both performance and reliability?

Redundant Array of Inexpensive (Independent) Disks (RAID)

- Virtualize multiple (unreliable and cheap) disks into a highly reliable and high-performance virtual disk.
  - *A case for redundant arrays of inexpensive disks (RAID)* (SIGMOD '88)

RAID is a ”reverse” form of virtualization

- Process: Virtualize one CPU into multiple virtual CPUs
- Virtual Memory: Virtualize one memory unit into multiple address spaces
- File System: Virtualize one drive into multiple virtual drives

RAID Fault Model: Fail-Stop

Disks may suddenly become completely inaccessible at any time.

- Mechanical failure, chip malfunction...
  - The disk seems to ”disappear suddenly” (data completely lost)
  - Assume the disk can report this issue.

The Golden Age in that Era

- 1988: Combine a few disks and disrupt the entire industry!
  - ”Single Large Expensive Disks” (IBM 3380) vs.
  - ”Redundant Array of Inexpensive Disks”

RAID 0 - Disk Striping

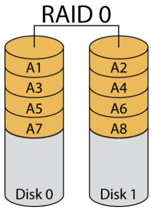
- Breaks data into **striped units** and spreads it across multiple drives.

Advantages:

- **High Performance:** High throughput (parallel I/O)
- **Cost-effective:** Better performance compared to a single large disk with similar capacity.

Disadvantages:

- **No Redundancy:** No backups; data loss occurs if any drive fails.
- **Increased Risk:** More drives mean a higher likelihood of failure.



RAID 0 – Striping

RAID 0: Design Space

Mapping virtual disk blocks to physical disk blocks.

- Assume two disks *A* and *B*
  - Same configuration, with *n* blocks in total
- A **"striped"** virtual disk *V* is created
  - $\{V_{2i-1}, V_{2i}\} \rightarrow \{A_i, B_i\} \quad (1 \leq i \leq n)$

Alternative Ways to Combine Two Disk Blocks

- Sequential Concatenation
  - $V_1 \rightarrow A_1, V_2 \rightarrow A_2, \ldots, V_n \rightarrow A_n$
  - $V_{n+1} \rightarrow B_1, V_{n+2} \rightarrow B_2, \ldots, V_{2n} \rightarrow B_n$
- Interleaving
  - $V_1 \rightarrow A_1, V_2 \rightarrow B_1$
  - $V_3 \rightarrow A_2, V_4 \rightarrow B_2$
  - $V_{2i-1} \rightarrow A_i, V_{2i} \rightarrow B_i$
- Although it lacks fault tolerance, it effectively utilizes disk bandwidth.

What do we *also* want?

Reliability: Data fetched is what you stored.
Availability: Data is there when you want it.

- More disks means higher probability of some disk failing.
- Striping reduces reliability
    - **N disks:** $1/n$th mean time between failures (MTBF) of 1 disk.

What can we do to improve Disk Reliability?

RAID 1 – Disk Mirroring

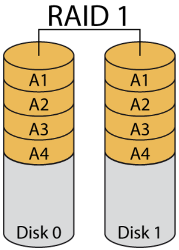
- Duplicates data and stores a copy on each drive (redundancy).
  - Requires at least two drives.
  - If one drive fails, data is still available on the other drive.
  - Supports **hot-swapping**: replace failed drives while the system is running.

Advantages:

- **Data Reliability:** Ensures no data loss even if a drive fails.
- **Redundancy:** Creates a mirror image of your data.

Disadvantages:

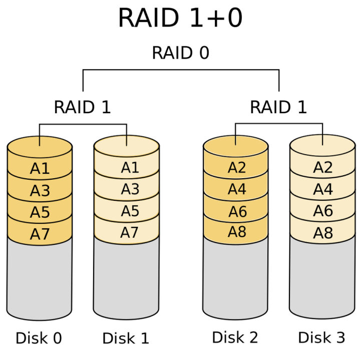
- **Storage Overhead:** Only 50% of total capacity is usable.



RAID 1 – Mirroring

RAID 10 = RAID 0 + RAID 1

- A hybrid RAID configuration: combines the speed of striping with the redundancy of mirroring.



RAID 10 – Combines RAID 1 (Mirroring) and RAID 0 (Striping)

RAID 10 = RAID 0 + RAID 1

Advantages:

- **High Performance:** Speed boost from striping.
- **High Reliability:** Data redundancy ensures no data loss.
- Can tolerate up to 2 drive failures (if not in the same mirrored pair).

Disadvantages: High Cost

- Minimum of **4 drives**.
- Only **50% of total capacity** is usable due to mirroring.
- Requires twice as many drives to achieve the desired storage capacity.

Recovering from Failures with Parity

RAID-10: Sometimes can tolerate two disk failures, sometimes not.
If we have many disks, can we reduce waste?

Reframe the Question

- Given n bits $b_1, b_2, \ldots, b_n$
- How many bits of information do we need to store at minimum so that we can recover any missing b_i (given that we know i)?
  - Parity (or error-correcting code)!
  - $x \oplus x = 0$
- ***n*-input XOR** gives bit-level parity:
  - $1 =$ odd parity, $0 =$ even parity
- Example:

$$1101 \oplus 1100 \oplus 0110 = 0111 \quad \text{(parity block)}$$

- Can reconstruct any missing block using XOR with others.

Parity Disk

Reserve a dedicated disk as a parity disk.

- $\{V_1, V_2, V_3\} \rightarrow \{A_1, B_1, C_1, D_1\}$
    - $V_1 \rightarrow A_1, V_2 \rightarrow B_1, V_3 \rightarrow C_1$ (No Fault Tolerance)
    - $D_1 = V_1 \oplus V_2 \oplus V_3$ (Parity Check)

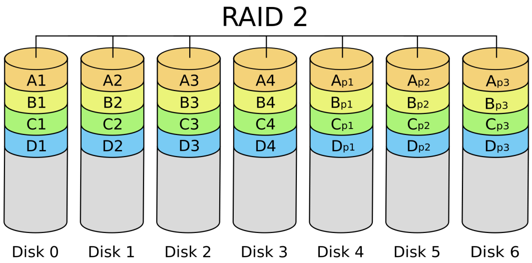
Performance Analysis

- Sequential/random read: 3x (75% of total bandwidth)
- Sequential write: 3x (75% of total bandwidth)
- Random write (tricky)
    - $D_1 = V_1 \oplus V_2 \oplus V_3$
    - Writing to any of V_1, V_2, V_3 requires updating D_1
        - Updating V_1 requires `readb(`$\{A_1, D_1\}$`)`, `writeb(`$\{A_1, D_1\}$`)`

Lesser Loved RAIDs - RAID 2

RAID 2: bit-level striping with ECC codes

- 7 disk arms synchronized and move in unison.
  - Uses **Hamming Code** for error correction.
  - For $n = 4$ data disks, minimum $r = 3$ redundant bits (ECC disks) are required.
- Complicated controller (and hence very unpopular).
- Tolerates 1 error with no performance degradation.

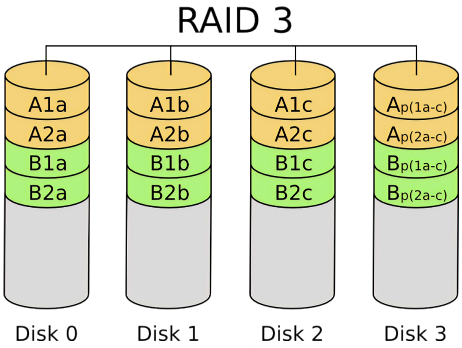


RAID 2 – Bit-level striping with ECC codes

Lesser Loved RAIDs - RAID 3

RAID 3: byte-level striping + parity disk

- Read accesses all data disks.
- Write accesses all data disks + parity disk.
- On disk failure: read parity disk, compute missing data.

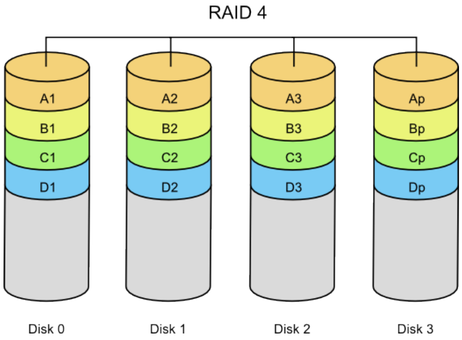


RAID 3

Lesser Loved RAIDs - RAID 4

RAID 4: block-level striping + parity disk

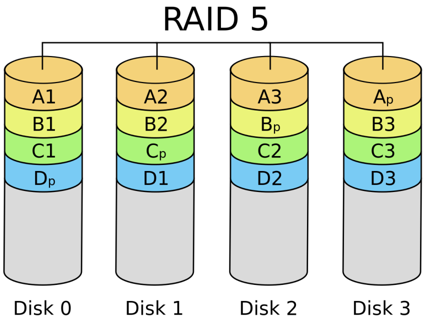
- **Advantage:** Better spatial locality for disk access.
- **Disadvantage:** Parity disk is a write bottleneck and wears out faster.



RAID 4

RAID 5: Rotating Parity

"Interleaved" parity block!



- In RAID 5, parity blocks are distributed across all disks to avoid a single parity disk bottleneck.
- This setup provides fault tolerance while maximizing disk performance.

RAID 5: Performance Analysis

Each disk has an equal opportunity to store parity.

- Sequential read/write: 3x (75% of total bandwidth)
- Random read (tricky)
    - If the read is large enough, all disks can provide data: 4x (100% of total bandwidth)
- Random write (tricky)
    - $D_1 = V_1 \oplus V_2 \oplus V_3$; writing to any of V_1, V_2, V_3 requires updating D_1
    - Parity check still heavily slows down random writes
        - At least *n* disks can achieve $n/4$ of random write performance
        - Offers some scalability

RAID: Discussion

Faster, more reliable, and nearly free high-capacity disks.

- Revolutionized the concept of ”high-reliability disks”
  - Became the standard configuration for today’s servers
- Similar milestones
  - *The Google File System* (SOSP ’03)
  - *MapReduce: Simplified Data Processing on Large Clusters* (OSDI ’04)
    - Transformed a collection of unreliable, commodity computers into a reliable, high-performance server.
    - Launched the ”Big Data” era!
- What is next?

More on Reliablity of RAID

Reliability of RAID

- What happens if a RAID system loses power?
  - Example: RAID-1 mirror disks with inconsistent data
- Detecting disk failures?
  - Automatic rebuild

Crash Consistency and Crash Recovery

Another Fault Model

The disk itself has no failure.

- However, the operating system kernel may crash, and the system might lose power.

File System: Tree Structure on the Device

- Even appending a single byte may involve multiple modifications to the disk
  - FAT, directory file (file size), and data
  - The disk itself does not provide ”all or nothing” support

Crash Consistency

Move the file system from one consistent state (e.g., before the file got appended to) to another atomically (e.g., after the inode, bitmap, and new data block have been written to disk).

Explanation: Ensures that if a crash occurs during an update, the file system is either fully updated or remains in its original state, avoiding partial or inconsistent changes.

However, the disk does not provide ”all or nothing” support for multi-block writes.

- Even for performance, there is no ordering guarantee
  - `bwrite` may be reordered: A sector that’s close but placed later, or a sector that’s far away but placed earlier — as a manufacturer, in pursuit of performance, which one would you choose?
  - Here we see reordering again—think about where else we’ve encountered it: compilers, CPUs, and race conditions ...

Consequences of Disk Reordering

Adding a cluster (4KB) to a FAT file requires updates to:

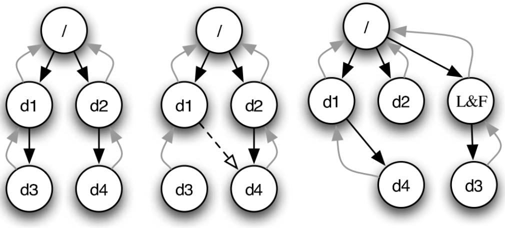
- File size in the directory entry (e.g., 100 $\rightarrow$ 4196)
- Chain in the FAT table (EOF $\rightarrow$ cluster-id, FREE $\rightarrow$ EOF)
- Actual data

This is troublesome...

- Any subset of writes might be lost
- The file system can enter an "inconsistent" state
  - May violate basic assumptions of FAT
    - Chain has no loops, and length matches file size
    - FREE clusters should not be linked
    - ...

File System Checking (FSCK)

Recover the "most probable" data structure based on existing information on the disk.



- **SQCK: A declarative file system checker** (OSDI '08)
- **Towards robust file system checkers** (FAST '18)
  - "Widely used file systems (EXT4, XFS, BtrFS, and F2FS) may leave the file system in an uncorrectable state if the repair procedure is interrupted unexpectedly"

Journaling

File System Checking is not the fundamental solution to crash consistency; **journaling** is.

Rethinking Data Structure Storage

Two ”Perspectives” of our data structure

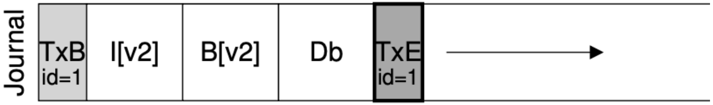
1. Actual Data Structure Storage
   - The ”direct view” of the file system
   - Crash unsafe
2. Append-only Log for Historical Operations
   - ”Redo” all operations to achieve the current state of the data structure
   - Easily achieves crash consistency

Combining Both Approaches

- When data structure operations occur, use (2) append-only log to record logs
- After logging, use (1) to update the data structure
- After a crash, replay the log and clear it (known as redo log; can also be undo log)

Implementing Atomic Append

Using `bread`, `bwrite`, and `bflush` to implement `append()`



1. Position at the end of the journal (`bread`)
2. `bwrite TXBegin` and all data structure operations
3. `bflush` to ensure data is written to disk
4. `bwrite TXEnd`
5. `bflush` to ensure data is written to disk
6. Write data structure operations to the actual data structure area
7. After data is written, delete (mark) the journal

Journaling: Optimization

Now, disk writes require duplicating data.

- **Batch Processing** (xv6; jbd)
  - Combine multiple system calls into a single transaction, reducing log size
  - jbd: periodic write back
- **Checksum** (ext4)
  - No longer marks `TXBegin/TXEnd`
  - Directly marks the length and checksum of the transaction
- **Metadata Journaling** (ext4 default)
  - Data writes occupy the majority of disk writes
  - Only journaling inode and bitmap can improve performance
  - Ensures the file system's directory structure is consistent, though data might be lost

Capabilities and Limitations of File Systems

Example: If you want to buld a Canvas

Using file system to store data:

- Each course is represented by a **directory**.
  - Student list is saved in `students.csv`
  - Stored in the format: `course/student_id`.
- Submissions are stored in the format: `course/student_id`.
  - `xxxxxxxx-2024-11-18-14-20-52.tar.bz2`
  - `xxxxxxxx-2024-11-18-14-20-52.tar.bz2.results`
    - Results are evaluated by Online Judge and then transferred using `scp`.
- Example: Displaying evaluation results.

Example Python Code:

```
for f in wiki.UPLOAD_PATH.glob(
    f'{course}/{module}/{stuid}/{file_pattern}'):
    if not f.name.endswith('.result'):
        # f is a submission
```

Did you realize Canvas is essentially a file system?

File System: Advantages

- A large number of UNIX tools/standard libraries can process files.
    - Easy to view, debug, and hack.
- **Examples:**
    - `find COP4610/L1 -name "*.result" | xargs rm`
    - `spam template.md COP4610/students.csv`

File System: Limitations

- **Low scalability**
    - Any page contamination involves traversing the entire directory.
- **Low reliability**
    - Almost no crash resistance (e.g., possibly invalid `.result` files).
        - Server Error

Relational Database

Structured Query Language (SQL)

"Everything is a table"

- SQL describes "what you want to do," and the database engine helps you "figure out how to execute it."
  - Indexing and query optimization

SQL Example:

```
    SELECT *
    FROM students, submissions
    WHERE submission.sid = students.sid AND
          submissions.course = 'COP4610' AND
          submissions.module = 'HW10';
```

Python Equivalent:

```
    for stu in students:
        for sub in submissions:
            if (stu.sid == sub.sid and
                (sub.course, sub.module) ==
                ('COP4610', 'HW10')):
               yield stu, sub
```

Database Transactions

ACID - Atomicity, Consistency, Isolation, Durability

- This is not something a ”big lock” can solve.
- Allows thousands of concurrent transactions
  - Supports backend applications in the Web 2.0 era
  - Example: Large-scale production deployment of a Canvas

```
BEGIN WORK;
-- All or nothing
INSERT INTO students VALUES (...);
INSERT INTO students VALUES (...);
INSERT INTO students VALUES (...);
COMMIT;
```

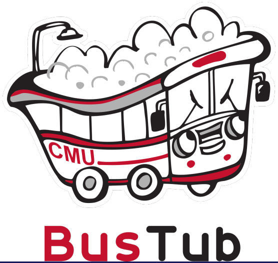
Database Implementation

Data structures on a virtual disk (file)

- Translate SQL queries into calls to `read`, `write`, `lseek`, `fsync`
- Concurrency control (transaction processing)

Want to learn how it's implemented?

- **Bustub from CMU 15-445**
  - L0 - C++ Primer
  - L1 - Buffer Pool Manager
  - L2 - Hash Index
  - L3 - Query Execution
  - L4 - Concurrency Control
- **SQLite**

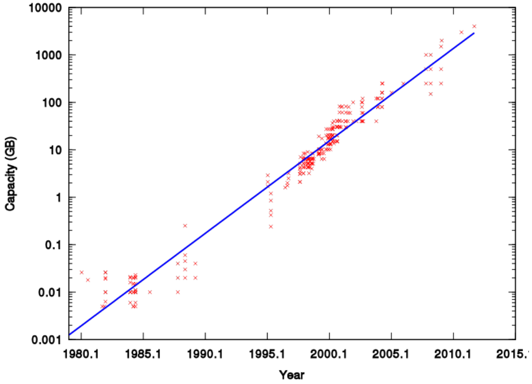


Towards a New Era

Why is the database no longer as popular as it used to be?

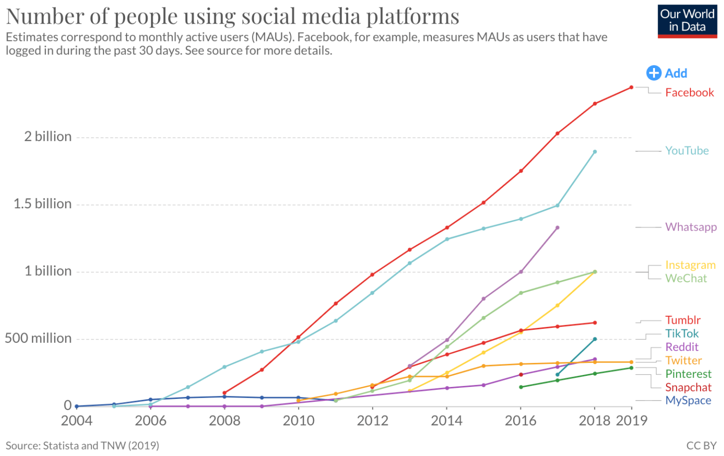
Demand Higher Performance

How Can Storage Systems Handle Massive, Real-Time Data?



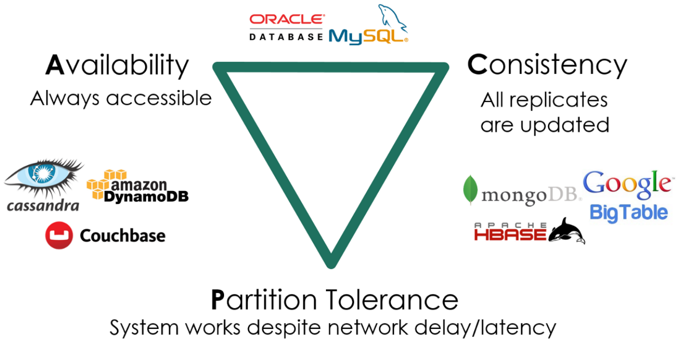
(1990s: High-speed networks + data centers could potentially store the entire Internet $\rightarrow$ Google Search)

Demand Higher Performance (Cont')



(2010s: Relational databases can no longer support social networks $\rightarrow$ Key-Value Storage)

Encountering the CAP Theorem Again



Building planet-scale databases faces unprecedented challenges:

- Requires an innovation similar to **“SLED vs. RAID”**.

Distributed Consensus Protocol

- In search of an understandable consensus algorithm (USENIX ATC’14, Best Paper Award)
  - ”Replicated State Machines”
    - Another state machine!
  - **RaftScope** Visualization Tool
    - Experience the horror of concurrent programming again
    - Not only concurrency, but threads may disappear at any time!

A More Distributed-Friendly Data Model: Key-Value

LevelDB Key-Value Storage

- Single-threaded (multi-threaded), ordered by key
- Supports transactions
- Supports snapshots (instantaneous read snapshots of a particular state)

One Implementation Method: Logging

- `snapshot()` returns the current length of the file
- `put(k, v)` directly appends `(k, v)` to the end of the file
  - Highly efficient
- `get(k, v)` traverses the entire file
  - Very low efficiency ("read amplification") $\rightarrow$ What can we do?

Log-structured Merge (LSM) Trees

A simulated "Memory Hierarchy"

- Writes directly append to a log file
- Crash safe; can't get any faster

Addressing Read Amplification

- Regardless of the order, maintain a real-time data structure (memtable) in memory for the log
    - `get` can first check in memory
- **Level 0**: Directly dump the memtable to disk
    - If the search fails, continue to the next level
- **Level 1**: When Level 0 is full (4MB), sort all keys, and merge with Level 1
    - Each subsequent level is 10 times the size of the previous level
- **Level 2**: When Level 1 is full, move the operations to Level 2, and so on

The New Era We Are In

The fundamental assumptions of storage systems are constantly being challenged:

- SSD: Low latency, high-speed sequential read/write
    - *WiscKey: Separating keys from values in SSD-conscious storage* (FAST'16)
- NVM (Intel Optane); byte-addressable
- RDMA high-speed network interconnect
- ...

The Future?

- Transactional flash?
- The return of SQL (F1, TiDB, CockroachDB, ...)
- Even a return to ”File Systems”?

In-Class Quiz

Takeaways

Questions addressed in this lecture

- **Q1:** How can we ensure the reliability of persistent data?
- **Q2:** How do modern applications use file systems?
  - Hardware redundancy: RAID
  - Software resilience: `fsck` and journaling

Takeaway Messages

- Multiple `bwrite` calls do not guarantee order and atomicity
- Journaling: Two perspectives on data structures
  - Actual data structure
  - Executed historical operations
- The file system provides a relatively simple file indexing mechanism.
  - Cannot guarantee the atomicity between multiple operations.
- Directly using the file system faces consistency and performance issues.
  - Relational Database
  - Key-Value Store (NoSQL DB)