WORKSHOP 3 - GOOGLE DRIVE CLONE

Cristian David Casas Diaz - 20192020091

Dylan Alejandro Solarte Rincón - 20201020088

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

PROFESSOR:

CARLOS ANDRÉS SIERRA VIRGUEZ

FRANCISCO JOSÉ DE CALDAS DISTRICT UNIVERSITY

COMPUTER ENGINEERING

DATABASES II

July 2025, Bogotá D.C.

Concurrency Analysis

1. Scenarios in the domain of FileService (File Management)

Simultaneous uploading of files by different users to the same folder

Context: Two or more users attempt to upload different files (or even the same file under different names) into the same folder in parallel.

Affected data:

- Files collection in a NoSQL technology (file metadata).
- Physical storage (Storage DB) where the blobs reside.
  Folder table in a SQL technology (querying the "number of items" or available space).

Possible concurrency:

- Simultaneous insertion of documents into the metadata collection ("FileMetadata").
- Simultaneous increment of used_storage in the User entity (if both uploads belong to the same user or to different users sharing quota under the same plan).
- Concurrent update of the parent folder (for example, if the file count is tracked in a field).

Downloading (read) of a file while it is being deleted or moved

Context: A user requests to download a file at the exact moment another process (for example, a trash purge) is performing the physical deletion (marking it as is_deleted = true) or moving it to another folder.

Affected data:

- is_deleted and deleted_at fields in the files collection.
- Location metadata (parent_folder_id) if it is being moved concurrently.

Possible concurrency:

- **Race condition:** The FileService logic might read the file that was just marked as deleted, returning an inconsistent or "in-between" resource.
- The physical storage repository (Storage DB) might attempt to delete the blob while the download stream has already been opened.

Simultaneous update of the same version of a file (versioning)

Context: Two processes/users (for example, the web client and a synchronization job) want to update the same "File" entity to create a new version at the same time.

Affected data:

- Version_History table (inserting a new version).
- version_path and checksum fields.
- Updating last_modified in the main "File" entity.

Possible concurrency:

- Inserting duplicate or inconsistent entries in Version_History if no locking mechanism (optimistic locking or serializable transaction) is in place.
- The size and checksum fields might be overwritten with different values depending on which process finishes first, potentially resulting in an incorrect version record.

2. Scenarios in the domain of Folder (Folder Organization)

Simultaneous creation/migration of subfolders under the same "parent_folder_id"

Context: Multiple users (or the same user from different browser tabs) create child folders under a root folder in parallel.

Affected data:

- Folder table (new row with parent_folder_id).
- Possible update of the parent folder's element count (if such a field exists).

Possible concurrency:

- Duplicate folder names if uniqueness is not enforced on (parent_folder_id, folder_name).
- If there is a children_count field in the parent folder, two simultaneous insertions could produce an incorrect value (if transactions or locks are not used).

Concurrent movement (rename/move) of nested folders

Context: A user moves folder A from /Folder1/ to /Folder2/ while another user (or a batch job) simultaneously attempts to move a subfolder of A to a different location.

Affected data:

- parent_folder_id field in the Folder table for folder "A."
- parent_folder_id fields of all subfolders of "A" (if a cascading operation is performed).

Possible concurrency:

- A cycle or inconsistency in the hierarchy could arise if changes are not serialized (for example, if "A" changes its parent before the subfolder processes its own change).
- If the system does not strictly validate the hierarchy, an orphaned folder could be created or even a parent–child loop.

3. Scenarios in the domain of SharingService and PermissionService (Sharing and Permissions)

Concurrency in assigning permissions to the same resource

Context: Two users (or the same user from different sessions) attempt to modify the same resource (file or folder) to assign different permissions to the same shared_with_id or to different users.

Affected data:

- Sharing collection in a NoSQL technology or an equivalent relational table (depending on the database choice).
- Resource_Tag table if permission is based on tags.
- Possible update of the record in Sharing_Link (if sharing via links).

Possible concurrency:

- Inserting two distinct documents with conflicting entries for the same combination (resource_id, shared_with_id, resource_type).
- Consistency could break if there is no unique index or an atomic upsert mechanism.
- If both processes change the permission_level simultaneously (for example, one sets "view" and the other sets "edit"), the last write will prevail unpredictably.

Concurrent generation of shareable links for the same resource

Context: A user creates a shareable link while another user (or an automated process) simultaneously claims it, modifies it, or deletes it (for example, due to automatic expiration).

Affected data:

- Sharing_Link table (fields: access_token, expires_at, password_hash, permission_level).
- Related data in the Resource entity that stores an index of "active links."

Possible concurrency:

- Two simultaneous accesses could overwrite the same row in the Sharing_Link table if a fixed identifier (link_id) is used without concurrency control.
- If a nightly job deactivates a URL at the same time the user modifies it (for example, changing the expiration date), the final outcome may not reflect the intention of either process.

Simultaneous permission change and permission read for auditing

Context: While an administrator modifies permissions (e.g., "remove access" or "grant edit") on a file, the auditing module (LogService) reads the current state of the permissions to record an activity log.

Affected data:

- Record in the Permission store (or in the permissions collection in a NoSQL technology).
- Insertion into Activity_Log (operation log entry).

Possible concurrency:

- The auditor might read an intermediate state that has not yet been committed (for example, reading that a user "has permission" when another process has revoked it but not yet completed the transaction).
- A "false positive" could be recorded (log of "grant" before it is actually applied) if transactions or read/write locks are not used.

4. Scenarios in the domain of UserService (User Management)

4.1 Concurrent update of storage quota (storage_quota and used_storage)

Context:

- A) Two upload processes (FileService) that simultaneously add file size to used_storage for the same user.
- B) A space-release process ("remove old versions" or "purge trash") that decrements used_storage at the same time another process is adding to it.

Affected data:

- User table (fields: used_storage, storage_quota, possibly last_login, is_active).

Possible concurrency:

- Two simultaneous increments to used_storage may not be mutually accounted (race condition), so one could overwrite the other, resulting in an incorrect value (for example, if used_storage was 100 MB and both processes upload 10 MB each, the result could end

up as 110 MB instead of 120 MB if the clause SET used_storage = used_storage + X is not used atomically).
- If one process “releases” 20 MB while another “consumes” 5 MB, the final value could fall below 0 or fail to accurately reflect the actual 75 MB usage.

Concurrent credential verification (Authentication)

Context:

- Multiple login attempts from different users or from the same user (on different devices) at the same time.
- While a user is logging in, another process (for example, a lock job after X failed attempts) could be changing the is_active column or the failed_attempts count.

Affected data:

- Authentication_Log table (inserting a new record).
- is_locked field or a failed_login_attempts counter in the User table.
- Updating last_login if authentication is successful.

Possible concurrency:

- A successful login could be recorded at the exact moment a batch job is locking the account, causing a legitimate user to be locked out even though their credentials were correct.
- If two failed login attempts occur simultaneously and exceed the limit (for example, both reach 4 where the limit is 5), two processes could “count” separately, causing an undesired lockout or a failure to unlock.

5. Scenarios in the domain of LogService (Audit Logs)

Concurrent reading of logs for report generation

Context:

- An Analytics process (or an administrator) executes complex queries on the Activity_Log table to display the “last 10 actions,” usage statistics, or to feed the data warehouse.
- At the same time, the system continues inserting records into that table at a high frequency.

Affected data:

- Reading entries from the Activity_Log table.

Possible concurrency:

- The read query may suffer from inconsistent reads (non-repeatable reads, phantom reads) if appropriate isolation levels (for example, READ COMMITTED) are not set.
- The report could include “half-written” records (write skew) or miss rows if a snapshot is not up to date.

6. Scenarios in the domain of Search & Tagging (Search and Tagging)

6.2 Concurrent assignment of the same tag to multiple resources

Context:

Two users (or automated scripts) attempt to assign the tag “proyecto2025” at the same time to a file and to a different folder.

Affected data:

- resource_tags collection (fields: resource_id, tag_id, resource_type) in a NoSQL technology.
- tag collection (which might insert a new document for tag_name if it does not exist) in a NoSQL technology.

Possible concurrency:

- Race condition when creating a new document in the tag collection: both processes check whether “proyecto2025” exists; since it does not yet exist, both insert it, resulting in a duplicate tag or a unique index error.
- If there is no uniqueness validation on tag_name, redundancy is created, and subsequent searches may return duplicate or confusing results.

7. Scenarios in the domain of AnalyticsService (Statistics and Big Data)

7.1 Large-scale (batch) extraction of usage data while new logs are being written

Context:

- An ETL job reads in bulk from the Activity_Log, files, sharing tables, etc., to feed the Data Warehouse (post-ETL) and compute global metrics (storage usage, access frequency).
- At the same time, the system continues inserting rows into Activity_Log and updating metadata in files.

Affected data:

- Tables in a SQL technology (e.g., User, Activity_Log).
- Collections in a NoSQL technology (e.g., files, resource_tags).
- Partial results in the Data Lake (NoSQL technology) and the Data Warehouse (SQL technology).

Possible concurrency:

- Dirty reads or non-repeatable reads: the batch job could include records that are being written in parallel, which may be acceptable in many analyses but could skew metrics if a consistent snapshot at the transaction level is not selected or if “read-only routing” to replicas is not used.
- The index in the NoSQL technology may not reflect the information “at the exact moment,” leading to discrepancies between the transactional database and the Data Warehouse.

8. Scenarios in the domain of Permissions DB (Access Control)

8.1 Concurrent permission evaluation at access time

Context:

- User A attempts to download a file shared by user B; the system must check in the Sharing store (or in the permissions graph in a NoSQL technology) whether A has "view" or "download" permission.
- At the same time, B revokes or modifies that permission from another session (for example, changing from "view-only" to "no-access").

Affected data:

- Shares collection or table (fields: permission_level, is_active) in the appropriate data store.

Possible concurrency:

- Race condition where A obtains permission because the query is read just before B executes the change, but by the time A confirms "download," the permission no longer exists (or has changed to "no-access").
- A temporary download might be allowed for A if a "lease" mechanism or a time-limited token is not implemented.

8.2 Concurrent update of the list of authorized users (role/ACL)

Context:

- While the system is updating (batch) the list of authorized users for a resource (for example, to synchronize a mass permission change), another process (a regular user) adds or removes a user from that list manually.

Affected data:

- Permission or Sharing table/collection (depending on the data model).

Possible concurrency:

- The entire list could be overwritten, undoing either the manual change or the batch change if the update is not performed “in cascade” or does not use an incremental model (e.g., SET allowed_users = array_union(allowed_users, new_users) vs. SET allowed_users = new_list).
- If the batch process removes users simultaneously with the manual edit, a permission that the user intended to retain could be revoked.

9. Scenarios in the domain of Subscription Plan (Plans and Quotas)

9.1 Concurrent subscription plan change and usage count

Context:

- The user requests an upgrade of their plan (for example, from “standard” to “premium”) at the same time as they are uploading multiple files (FileService).

Affected data:

- Subscription_Plan table (possible fields: is_active, plan_name) in a SQL technology.
- User entity (the storage_quota field, which depends on the plan).

Possible concurrency:

- If the “change plan” transaction occurs after the upload transaction, the quota‑assignment logic (which relies on the current plan) may assign the new quota incorrectly because the plan update and the used_storage increment are not in the same transaction.
- While uploading a large file at the same time the plan change is applied, the calculation of the “new remaining space” could be incorrect.

10. Scenarios crossing multiple services and databases (Distributed Transactions)

10.1 Distributed transaction between FileService and LogService

Context:

- When the user uploads a file, FileService must:
    - Save metadata in a NoSQL technology (FileMetadata).
    - Save the blob in the Storage DB (S3 or on‑premise).
    - Insert a record into Activity_Log (in a SQL technology).
- If performed without orchestration, there is a risk that the metadata is saved but the log insert fails, or vice versa.

Affected data:

- files collection (NoSQL).
- Object in Storage (S3 or similar).
- Activity_Log table in a SQL technology.

Possible concurrency/consistency:

- Lack of atomicity: If the insertion into Activity_Log fails, the file will already be stored without an audit trace.
- A mechanism such as Two‑Phase Commit or Sagas is required to coordinate the distributed operation between heterogeneous databases (NoSQL + SQL).

10.2 Consistency between NoSQL (Metrics) and SQL (Users)

Context:

- When generating a "storage usage" report for a user, the system reads from a SQL technology (used_storage) while, in parallel, AnalyticsService is writing usage statistics to a NoSQL technology or another NoSQL technology (time‑series).

Affected data:

- Reading User.used_storage (in a SQL technology).
- Inserting metrics into a NoSQL technology (time‑series).

Possible concurrency:

- The report could display outdated or inconsistent data if the sources are not synchronized; for example, used_storage indicates 500 MB, but in the analytics data store the recent upload peak has not yet been recorded, causing discrepancies in the dashboards.

Problem Identification And Proposed Solutions

1. FileService (File Management)

1.1. Potential Problems

Race conditions when uploading files concurrently

- When two processes insert metadata in parallel into the same file location, each may read the same previous value (for example, used space) and then write its own calculation, causing one of the increments to be lost.

- If there is a counter of the number of files or total used space within the parent folder, both processes may read the same current count and then write an identical value, leaving the counter in an incorrect state.

Inconsistent reads during download versus deletion or move

- A download process may begin reading a file at the exact moment another process marks that file as deleted or moves it to another folder, delivering a resource in an “in-between” state (neither fully present nor fully deleted).

- If the physical deletion of the blob occurs at the same moment a download stream is opened, the reading process may encounter a partially deleted file or I/O errors.

Conflicts in simultaneous versioning

- Two processes attempt to generate a new version of the same file from the same base (for example, the last recorded version). Both read the same "previous version" reference and then insert an entry into the version history, creating duplicate or inconsistent records.

- The fields that store last‑modified timestamp, size, or checksum may end up holding the value from whichever process finishes last, ignoring the other process's modification and causing misalignment in the version history.

Deadlocks between metadata writes and blob storage

- Process A first locks the metadata entry (to update or insert) and then tries to lock the physical blob. Meanwhile, process B does the reverse (first locks the blob, then the metadata entry). Acquiring locks in the opposite order can create a mutual waiting cycle (deadlock).

1.2 Proposed Solutions

For concurrent uploads

- **Use of transactions at the user‑record level:** Before increasing a user's used space, wrap the operation in a transaction that locks the corresponding user record until the calculation and update complete. This guarantees that no other process can read or write to that record simultaneously.

- **Atomic increment operations:** Employ the repository's built-in mechanism to add or subtract used space in a single atomic operation, so that the database itself handles the increment without risk of two processes interfering.

- **Uniqueness constraint on filenames within each folder:** Configure a unique index on (parent folder, file name) so that if two processes try to upload a file with the same name at once, one of them will receive a duplication error that can be handled in the service layer.

For concurrent downloads versus deletion or move

- **Pre‑download metadata check:** Before actually opening the download stream, start a read‑only transaction on the metadata and confirm that the “is_deleted” flag is false and that the file path remains as expected. If the file is already marked deleted or has moved, abort the download.

- **Logical delete lock:** Introduce a temporary field in the metadata (e.g., download_in_progress) to signal that someone is downloading. When attempting deletion, the flow first checks that field and either waits or fails with “file in use.” This ensures that the blob is not deleted while a download is in progress.

For simultaneous versioning

- **Optimistic concurrency using a version field:** Add a version number or timestamp field to the file’s main entry. When creating a new version, read the current version, compute the next version number, and try to insert the new record. If another process has already inserted that version in the meantime, detect that the base version changed and retry.

- **Locking the primary file record during version creation:** Instead of allowing two processes to read and write simultaneously, first lock the file’s main record until the new version entry is inserted. This prevents any other process from creating a version until that lock is released.

Deadlock prevention between metadata and blob storage

- **Consistent lock‑acquisition order:** Establish that any operation involving metadata and blobs must first acquire the lock on the metadata entry, and only then acquire the lock on the blob. Never allow the reverse ordering.

- **Lock‑timeout and retry with back-off:** Configure a timeout for each lock attempt so that if it cannot be acquired within a set interval, the transaction rolls back and retries after a short delay.

2. Folder (Folder Organization)

2.1 Potential Problems

Race conditions when creating subfolders simultaneously

- When two processes try to create child folders under the same parent at the same time, each may read the same count of existing children or the same list of names. Without atomic uniqueness checks, duplicate folder names can be created, or the parent's child count may end up incorrect.

Conflicts in moving or renaming nested folders

- If process A locks folder "A" to change its parent, then cascades locks on its subfolders to update their parent references, while simultaneously process B tries to lock a subfolder of "A" first, the opposite lock ordering can cause a deadlock.

- If operations are not fully serialized, a cycle can form in the hierarchy: e.g., folder "A" has already changed its parent, but its subfolders have not yet been updated, leading to inconsistent paths.

Intermediate reads of the hierarchy

- A process browsing the full folder tree might read a state where "A" has already changed its parent, but its subfolders have not, resulting in a snapshot containing invalid routes or logical cycles.

2.2 Proposed Solutions

Creating subfolders concurrently

- **Unique index on folder name within the same parent:** Enforce a constraint so that if two processes attempt to create folders with the same name under one parent, one of them fails with a duplication error. That exception is caught and the user is informed that a folder with that name already exists.

- **Transaction to lock the parent folder entry:** Wrap the insertion in a transaction that locks the parent's record until the new child is inserted and any child‑count field is updated. Whoever obtains the lock first updates the count and inserts; the other must wait until the lock is released.

Concurrent move/rename of nested folders

- **Hierarchical lock protocol in descending order:** Any move operation must first request a lock on the highest‑level folder being moved, then, in descending order, lock each subfolder that will be updated. This ensures that if another process attempts to move a subfolder of that same branch, it must respect the same order and a deadlock cannot occur.

- **Strict isolation for updating parent references:** Perform the parent_folder_id update inside a transaction at the highest isolation level (serializable). If another process is also modifying any of the same folders, only one transaction will succeed and the other will be retried, preventing inconsistent moves.

Pre‑move hierarchy validation in the application layer

- Before confirming the move of folder “A” to a new parent, traverse the hierarchy (for example, with recursive queries) to ensure that the destination is not one of “A”’s own subfolders. If it is, reject the operation to avoid creating a cycle.

3. SharingService and PermissionService (Sharing and Permissions)

3.1 Potential Problems

Race conditions when assigning permissions simultaneously

- Two processes attempt at the same time to insert or update the same permission entry that grants access to a resource for a given user. Both read the previous state (e.g., no permission), each calculates its own update, then writes, creating duplicates or unpredictably overriding the permission level.

Concurrent generation of shareable links for the same resource

- When a user creates a link to share a resource, a unique identifier is generated and stored along with expiration and access level. If two processes generate links simultaneously (due to lack of token‑generation control), they may both attempt to use the same token ID, leading to overwrites or duplication errors.

- An automated job that marks expired links as inactive could run at the same moment a user manually extends the expiration date of that link, resulting in a final state that matches neither intention fully.

Intermediate read of permissions during auditing

- While an administrator revokes or modifies permissions in the permission store, the LogService may read the record before the revoke transaction commits, capturing a state that incorrectly shows “still permitted” and producing a misleading audit log entry.

Deadlock between updating permissions and associated tag updates

- If there is logic that assigns tags to resources based on permission level, process A might lock the tags repository to add a tag and then try to update the sharing entry. Simultaneously, process B locks the sharing entry first (to change permissions) and then attempts to modify tags. Because they lock in opposite orders, a deadlock arises.

3.2 Proposed Solutions

Unique composite key for permissions

- Define a uniqueness constraint on the combination (resource, user, resource type) so that if two processes try to insert the same permission simultaneously, one fails with a duplication error. That error can be caught and handled by updating the existing record's level rather than inserting anew.

Record‑level or document‑level locking in a fixed order

- Establish a protocol requiring any modification of permissions to lock the specific permission record first, then only after confirming that lock proceed to modify any related entities (e.g., shareable links or tags). As long as every process adheres to this lock order, no deadlocks occur from inverted lock requests.

Permission read for auditing under repeatable read isolation

- When the LogService needs to read permissions and generate a log entry, perform the read inside a transaction that fixes the state at the start (equivalent to "repeatable read" isolation). Even if another process is modifying the permission concurrently, the audit read sees only the state that existed when the transaction began, avoiding false audit entries.

Automatic expiration of links via an internal mechanism

- Instead of relying on an external job to periodically mark expired links as inactive, configure the link store so that once the expiration timestamp is reached, the link is removed or disabled automatically. This removes any contention between the automatic revocation process and manual expiration–date adjustments.

4. UserService (User Management)

4.1 Potential Problems

Race conditions when updating a user's used storage

- Multiple file‑upload processes for the same user read the current "used storage" in parallel, each adds its own file size, and then both write the result, so only one increment actually takes effect.

- At the same time another process purges old versions (freeing space) and another uploads a file, if they read the value in an overlapping manner, the final used space can be incorrect (even dropping below zero).

Race conditions on credential verification

- Two failed login attempts for the same account happen simultaneously. Each reads the count of failed attempts, both add one, and both write the same new count, when actually one needed to push the count beyond the threshold to trigger a lock.

- A successful login check and a background job that blocks the account after too many failures may occur at the same moment, resulting in the user authenticating successfully but also being blocked immediately.

Deadlocks between updating user record and writing authentication logs

- Process A locks the user row to update "last login" and then attempts to insert a new authentication log entry. In parallel, process B locks the log table (for example, archival jobs) and then attempts to mark the user account as inactive. Since they lock resources in opposite orders, a deadlock arises.

4.2 Proposed Solutions

Use transactions for updating used storage

- Before adding or subtracting space used, start a transaction that locks only the user's record. Within that transaction, read the current used space, compute the new value (adding or subtracting), and write it. Only when this transaction commits is the new used‑space value visible. This prevents any other process from modifying it concurrently.

- Alternatively, use the storage‑repository's built-in atomic add/subtract capability so that the operation never has a window where two processes interfere.

Managing the failed‑attempts counter with a locked record

- To increment the counter of failed login attempts, open a short transaction that locks the user's record. Inside that transaction, read the current count, add one, and then write the updated count. When the transaction completes, the lock is released, guaranteeing no two processes read the same old value.

- If blocking behavior is triggered at a threshold, separate the “verify credentials” step from “increment attempt count” into two very short transactions so that the lock on the user record holds only while updating the counter.

Grouping login check and log entry in a single transaction

- Bundle credential validation, possible updates to “last login” or “failed attempts,” and insertion of the login‑event record into one transaction at a sufficiently strict isolation level. If the database detects a deadlock, it will abort one transaction so it can be retried, ensuring data consistency.

Consistent lock ordering to prevent deadlocks

- Define that any process updating both the user record and the log table must first acquire the lock on the user record, then insert or update the log. Any archival job or background task touching the log table should first check that no user record locks are in progress, or else back off and retry, ensuring the same acquisition order is respected.

5. LogService (Audit Logs)

5.1 Potential Problems

Inconsistent reads when generating reports

- An Analytics process or administrator queries the log table for complex statistics (e.g., “last ten actions”) while the system continues inserting new log entries. Without an appropriate isolation level, the query might return ghost rows, incomplete records, or duplicates.

Contention during high‑volume inserts

- A large number of concurrent insert operations into the log table can cause index or storage‑engine locks, slowing down writes and leading to transaction timeouts.

Deadlocks between log inserts and aggregation triggers

- If the log table has a trigger that updates another table of aggregated counts (for example, actions per user), process A might insert a log entry first and then the trigger tries to update the counts table. Meanwhile, process B attempts to update the counts table directly. If they don’t respect the same lock order (counts first, then insert log), a deadlock can occur.

5.2 Proposed Solutions

Read from replicas or materialized views for analytics

- Instead of querying the primary log table when generating reports, direct those reads to a read‑only replica or a materialized view that receives near‑real-time data. This way, primary inserts are not contending with read queries, avoiding inconsistent reads and improving performance.

Optimize for high-throughput inserts

- Partition the log table by date (for example, month). When inserting new entries, only the current partition is locked, leaving older partitions free from contention.

- Temporarily disable nonessential triggers during peak insertion periods, and then update aggregates in bulk afterward.

Consistent lock order in aggregation triggers

- If a trigger must update an aggregation table each time a log entry is inserted, enforce that the trigger first locks the aggregation table and then inserts into the log. If all processes follow that same order, deadlocks due to inverted lock requests cannot occur.

Short transactions and retry on timeouts

- Keep transactions that insert log entries as short as possible, avoiding prolonged reads or heavy computations. If the log‑insert transaction times out, abort and retry after a randomized back-off to free up resources quickly.

6. Search & Tagging (Search and Tagging)

6.1 Potential Problems

1. **Desynchronization between the metadata store and the search index**

    - When tags or metadata for a file or folder are updated, the change is first saved in the primary metadata store and then pushed to the search index. If two processes modify the same metadata almost simultaneously and each notifies the indexer in parallel, the search index may end up reflecting a state that does not match the primary store.

2. **Race condition when creating the same tag**

    ○ Two processes simultaneously query whether the tag named “project2025” exists. Both see that it does not exist and then both try to insert it. This results in duplicate tags in the tags collection, leading to confusion in later searches.

3. **Deadlock between writing metadata and notifying the indexer**

    ○ If the application groups the metadata write and the search index update in the same logical operation, a failure in the indexer could leave metadata written but not indexed. Attempting to roll back could cause a deadlock if the rollback tries to delete the metadata while another process already has a partial lock on that record.

6.2 Proposed Solutions

1. **Messaging queue pattern for indexing**

    ○ Instead of immediately sending the update to the indexer after saving metadata, publish an event to a messaging queue (e.g., “MetadataUpdated”). A dedicated consumer reads from that queue and updates the search index in sequence. In this way, if two events arrive nearly simultaneously, the indexer processes them one after the other, preserving eventual consistency between the primary store and the index.

2. **Unique index on tag name with duplicate-error handling**

    ○ Enforce a unique constraint on the “tag name” field. If two processes try to insert “project2025” at the same time, one will fail with a duplication error. By catching that error, you can simply look up the existing tag ID and proceed without creating a duplicate.

3. **Version or timestamp control in metadata and index**

    ○ Add a “last modified” timestamp or version field to each metadata document. When sending the update to the index, include that timestamp. If a later event arrives with an older timestamp than what the index already has, the indexer discards it, preventing outdated updates from overwriting newer information.

4. **Transactional grouping in the metadata store with guaranteed queue publication**

    ○ Wrap metadata writes and queue notifications in a single atomic operation at the metadata‑store level. If the messaging system cannot accept the event, the entire write is rolled back. This ensures there are never metadata changes without a corresponding indexer notification.

7. AnalyticsService (Statistics and Big Data)

7.1 Potential Problems

1. **Dirty or non-repeatable reads in batch data extraction**
   - An ETL job reads historical usage logs and file metadata while new events continue to be inserted. Without proper isolation, the extraction can include records that are in the middle of being written or may read the same row twice if records change during the read.
2. **Contention in high-volume data systems**
   - Reading large tables or collections of logs and metadata can lock indexes or data regions where new records are being inserted concurrently, slowing down writes and causing timeouts in user operations.
3. **Lag between transactional data and analytical metrics**
   - When generating a report showing a user's used space, the query reads the updated value in the user's record (for example, 600 MB). But the analytics system's time-series data might not yet reflect the latest upload, showing only 550 MB and causing a discrepancy.

7.2 Proposed Solutions

1. **Read from replicas or snapshots in read-only mode**
   - For large‑scale data extraction, direct queries to a read-only replica or snapshot that receives changes with minimal delay. In this way, primary inserts and updates do not compete with the read job, ensuring consistent data and reducing contention.
   - If available, use snapshot isolation so that each extraction sees a stable view of the data at the transaction's start, unaffected by ongoing writes.
2. **Partitioning or sharding to reduce locking**
   - Partition the log table by date (e.g., monthly partitions). When the ETL job reads from a specific month, it locks only that partition, leaving other partitions unaffected by the read.
   - In the metadata collections for files, distribute data across multiple shards or nodes in a distributed deployment, so that intensive read operations do not saturate a single node or

block the entire collection.

3. **Incremental extraction using a timestamp watermark**
    - Maintain a "last processed timestamp" field in a control table. Each ETL run reads only records with timestamps later than that watermark. After successfully processing, update the watermark within the same transaction. This avoids re-reading the entire history and significantly reduces contention.
4. **Event-driven architecture for real-time metrics**
    - Whenever an important event occurs (file upload, deletion, sharing), publish a message to a message bus. A dedicated consumer processes each event and updates the analytics store (time-series or data lake) incrementally. This eliminates the need for large batch jobs and achieves eventual consistency with minimal locking.

---

8. Permissions DB (Access Control)

8.1 Potential Problems

1. **Race condition in permission evaluation at access time**
    - User A attempts to download a resource, and the system checks the permission store just before User B revokes that permission. If the read occurs just before the revoke, the download is erroneously authorized with an already outdated permission.
2. **Race conditions when updating the list of authorized users (ACL)**
    - A batch process attempts to overwrite the entire list of users authorized for a resource (e.g., as part of a mass synchronization) at the same time a normal user is manually adding or removing their own permission. When both processes finish, one of the changes is lost, depending on who wrote last.
3. **Deadlocks in graph-model permission reads and writes**
    - If permissions are modeled as a graph (to allow inheritance), process A might read node and relationship data from the graph while process B tries to modify those same nodes/relationships. If they lock in opposite order, the system can deadlock.

8.2 Proposed Solutions

1. **Validate and generate access tokens in a single transaction**
   - When checking whether a user has permission and, if so, generating an access token for download, combine both actions into one transaction. This way, no one can revoke the permission until the transaction completes, preventing a stale permission from being used.
2. **Incremental updates instead of overwriting entire ACL lists**
   - Instead of setting “allowed_users = new_list,” use operations that add or remove individual elements from the list (e.g., “add user X” or “remove user Y”). That approach prevents a batch process from wiping out manual changes, because each change is applied as an incremental update.
   - If a complete ACL replacement is truly required, first compare the current list with the new list in a transaction, then apply only the necessary additions or removals rather than overwriting everything.
3. **Optimistic concurrency using a version field in the permission store**
   - Include a version number or `updated_at` timestamp in each permission record. When updating, require that the version or timestamp matches the previously read value. If it does not (meaning somebody else updated in the meantime), the update fails and must be retried with fresh data.
4. **Automatic link expiration via an internal mechanism**
   - Instead of having an external job periodically scan and deactivate expired links, configure the link store so that when the expiration timestamp is reached, the link is automatically removed or flagged “inactive.” This eliminates any contention between an automatic revocation process and manual expiration‑date adjustments.

---

9. Subscription Plan (Plans and Quotas)

9.1 Potential Problems

1. **Race condition between plan change and file uploads**
   - The user requests a plan upgrade that changes their storage limit at the same time other processes are uploading files (which increase used space). If the upload completes just before the plan change, the calculation of the new available space may be incorrect or

outdated.

2. **Inconsistency in available quota during an update**
   - If the plan change is applied without first reading how much space the user is already using, the system may assign a new quota that ignores previously stored data, allowing the user to exceed their limit without warning.

3. **Deadlock between user record and subscription‑plan record modifications**
   - Process A locks the subscription plan record (for example, to mark it inactive or update its attributes) and then tries to read/update the user record. In parallel, process B locks the user record (for example, to update used space) and then reads the plan record to check limits. Since they lock resources in opposite orders, a deadlock can occur.

9.2 Proposed Solutions

1. **Single transaction to verify used space and change plan**
   - When processing a plan change, first read the current used‑space value for the user and, in that same transaction, calculate whether the new plan's quota can accommodate it. If used space exceeds the new plan, abort. If it fits, update both the user's plan assignment and the quota in one step, ensuring no other process can modify those values in between.

2. **Consistent lock order to prevent deadlocks**
   - Define a rule that any flow updating both the user record and the plan record must first lock the user record and then the plan record. For example, lock the user's row to check/update data, then lock the plan's row. If all processes follow that same ordering, no deadlocks will occur from reversed locks.

3. **Pre-check in the service layer to anticipate errors**
   - Before starting a transaction, perform a quick read of the user's used space and the new plan's quota. If used space already exceeds the new quota, return an error without starting a transaction. This reduces the window in which locks are held and lowers the chance of deadlocks.

4. **Optimistic concurrency with a version field in the user record**
   - Add a version or last-updated timestamp field to the user's record. When requesting a plan change, read that version and include it as a condition in the update. If the version

changed before writing, the update fails and must be retried with fresh data.

---

10. Distributed Transactions (Cross-Service and Multiple Data Stores)

10.1 Potential Problems

1. **Lack of atomicity between FileService (metadata and blobs) and LogService (activity logs)**
    - First the file metadata is written and the blob is stored. Then the system attempts to insert an activity log entry. If that last insertion fails, the file remains in the system without any audit history.
    - Attempting a traditional two-phase commit across heterogeneous stores (metadata store and activity log store) often becomes extremely slow and may lead to prolonged locks in both systems.
2. **Inconsistencies between transactional data (users, used space) and analytical metrics**
    - The analytics service writes usage metrics to a time-series store while billing is calculated based on the used space stored in the user's record. If the analytics store hasn't yet received the latest usage event, reports or dashboards show outdated values.
3. **Deadlocks in poorly coordinated distributed flows**
    - One distributed flow A writes first to the metadata store (or user record) and then to the activity log. Another flow B does the reverse (first to the log, then to the metadata). If they lock related records at each step, a cycle of waiting and a deadlock can result.

10.2 Proposed Solutions

1. **Saga pattern (orchestrated or choreographed) to coordinate remote steps**
    - Instead of attempting one global distributed transaction, implement a saga:
        - **FileService** writes file metadata and publishes an event to a message queue (e.g., "MetadataCreated").

        - A central orchestrator listens for “MetadataCreated” and invokes the blob-storage service.
        - When the blob is successfully stored, the orchestrator publishes “BlobStored.”
        - **LogService** listens for “BlobStored” and inserts an entry into the activity log.
        - Finally, the orchestrator marks the saga as completed.
    - **Compensation steps:**
        - If the blob-storage step fails, an event is published to remove the metadata record.
        - If the log insertion fails, an event is published to remove the blob and metadata or mark the metadata as “pending audit” for manual intervention.
    - This way, each step runs within its own local transaction, and coordination happens through events rather than holding locks across multiple systems.
2. **Eventual consistency using message publication and periodic reconciliation**
    - Each update to a user’s used space publishes a message to a metrics channel. The AnalyticsService consumes those messages and updates its time-series store incrementally.
    - Periodically, a reconciliation job compares the used-space value in the user record with the analytics store value and corrects any discrepancies. This approach accepts a small window of inconsistency but ensures that the data eventually converges.
3. **Version and timestamp control for synchronization**
    - Add a `last_updated_ts` field to the user record. Whenever used space changes, update that timestamp and publish an event containing both the new value and the timestamp.
    - The AnalyticsService, when processing each event, checks if the event’s timestamp is newer than what is already stored in its database. If it is older, it discards the event, avoiding overwriting with stale data.
4. **Partial rollback and compensation on failure**
    - If a distributed flow fails at any step, trigger compensation actions to undo previous changes in each system:

        - If the log insertion fails, notify FileService to delete the blob and mark or delete the metadata.
        - If blob storage fails, notify FileService to delete the metadata record.
    - Although there is no single global transaction, each repository is returned to a consistent state via compensating actions.

Parallel and Distributed Database Design

High-Level Design Overview

To meet the scalability, performance, and availability requirements of the system, our architecture adopts a distributed and parallel database design. This approach aligns naturally with the modular and hexagonal structure defined in the current architecture, enabling each service to operate independently, scale horizontally, and handle user demand efficiently.

Instead of relying on monolithic storage solutions, data is partitioned and replicated across multiple nodes or regions based on access patterns, data type, and usage characteristics. Services such as file management, analytics, user handling, logging, sharing, and permission control interact with dedicated storage layers that support parallelism at both the query and infrastructure levels.

Distributed Data and Query Model

Each domain service accesses its own data in a way that optimizes performance and fault tolerance. Here is an overview of how data is distributed and accessed in parallel:

- **UserService** connects to replicated user data distributed by region to reduce latency and ensure availability.

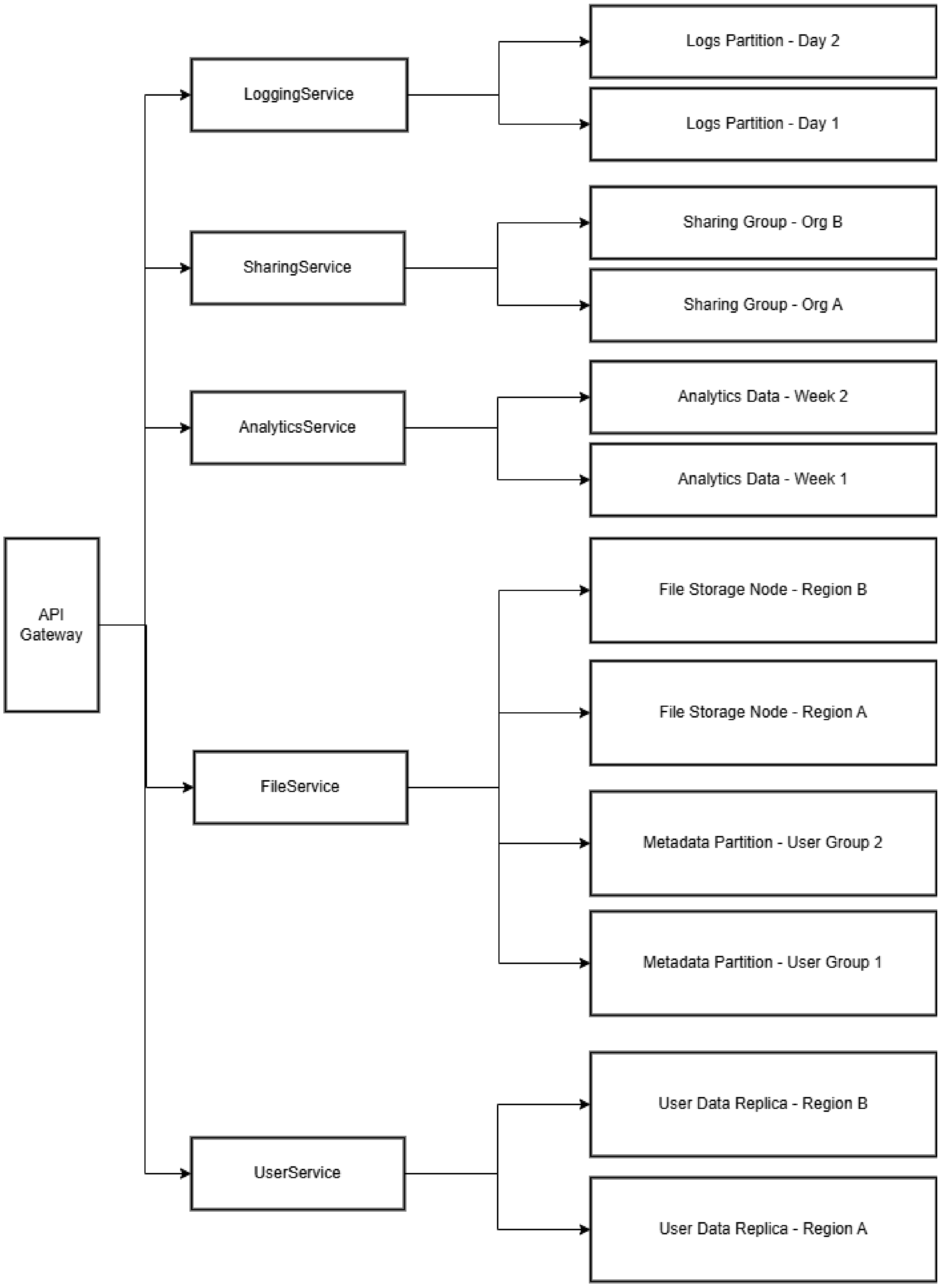
- **FileService** handles file metadata and content that is partitioned by user groups and stored regionally based on access location.

- **AnalyticsService** operates over time-based partitions of logs and usage metrics, allowing parallel processing of historical data.

- **SharingService** accesses grouped access control data optimized for frequent collaboration.

- **PermissionService** (described in detail below) uses graph-based community partitioning.

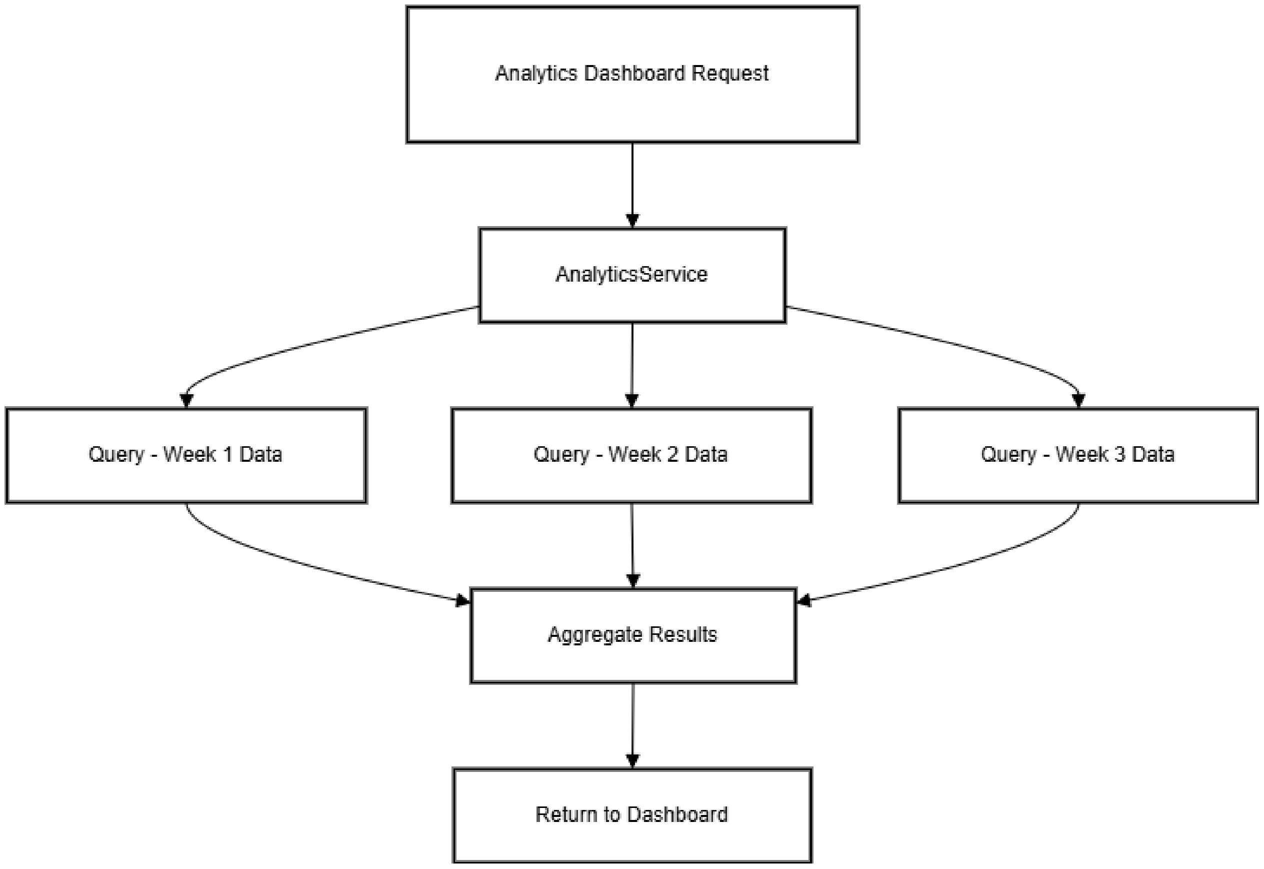
- **LoggingService** stores and queries event logs distributed by time range, optimizing analytical and debugging queries.


API Gateway
LoggingService
SharingService
AnalyticsService
FileService
UserService
Logs Partition - Day 2
Logs Partition - Day 1
Sharing Group - Org B
Sharing Group - Org A
Analytics Data - Week 2
Analytics Data - Week 1
File Storage Node - Region B
File Storage Node - Region A
Metadata Partition - User Group 2
Metadata Partition - User Group 1
User Data Replica - Region B
User Data Replica - Region A

Parallel Query Execution for Analytics

The system's analytics module benefits significantly from parallelism. Instead of scanning large datasets sequentially, queries are divided into independent subtasks that target specific time intervals (e.g., per week or per day). Each subquery is executed in parallel, and the results are then merged and aggregated by the analytics service before being returned to the user interface.

This allows faster insight generation even when the data volume is large or the query is complex.

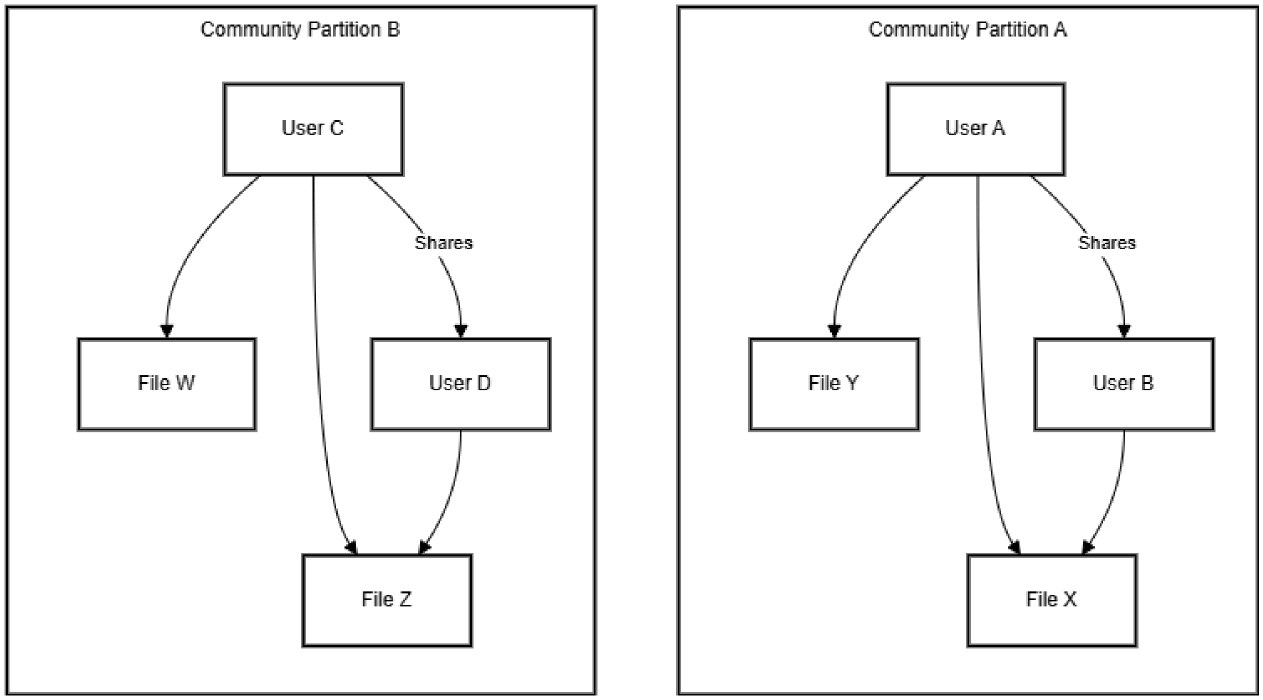


PermissionService: Graph-Based Community Partitioning

Unlike other services that work with structured or semi-structured data, the PermissionService operates over a dynamic, relationship-rich graph. In this context, traditional sharding strategies (e.g., by workspace or user ID) are inefficient due to the cost of traversing across partition boundaries.

To address this, we use **graph community-based partitioning**. This method groups highly connected nodes into the same partition. Community detection algorithms (e.g., Louvain, Label Propagation) are used to periodically rebalance the graph as relationships evolve. This reduces cross-partition queries and significantly improves performance on permission checks and access traversals.

Inactive or low-activity nodes can be grouped into a separate "cold" partition, while active collaborative clusters remain in optimized "hot" partitions.



Performance Improvement Strategies through Parallelism and Distribution

Below are four strategies to improve system performance using parallelism and distributed architectures, aligned with the requirements and system architecture defined in the Google Drive Clone project:

1. Horizontal Scaling with Load Balancing

This strategy involves adding multiple server instances (e.g., API Gateway nodes and domain services) to distribute the workload. It aligns with the hexagonal architecture proposed in the Workshops, allowing backend requests to be handled by multiple nodes without degrading user experience.

Project Application:

- The system supports up to 5,000 concurrent users (see RNF2.1), so horizontal scaling is essential to maintain performance.
- A load balancer can distribute traffic across multiple instances of services such as FileService, UserService, and AnalyticsService.

Trade-offs and Challenges:

- Increased complexity in session and cache synchronization.
- Requires consistent session handling across nodes (e.g., session stickiness or centralized session storage).
- Higher infrastructure costs.

2. Data Partitioning (Sharding)

Data is divided horizontally across multiple databases or collections based on criteria such as user ID, geographical region, or file type. For example, file metadata (stored in a NoSQL document-oriented database) can be partitioned by user or by resource type.

Project Application:

- Since a flexible, scalable NoSQL database is used to store metadata and logs (Workshop 1 and 2), sharding is a viable solution to distribute read/write loads over millions of documents.
- The domain-based architecture and use of microservices facilitate its implementation.

Trade-offs and Challenges:

- Requires robust mechanisms for aggregating and querying across shards.
- Increases complexity for transactions and maintenance.
- Risk of data imbalance if the sharding key is not well chosen.

3. Database Replication

Database replication involves maintaining synchronized read-only copies of databases to distribute query loads and ensure availability. The project architecture already considers this setup for recovery and analysis purposes.

Project Application:

- Relational databases managing structured data like users and profiles can maintain active replicas for read-heavy operations (e.g., dashboards or auditing).
- Non-relational databases used for metadata and event storage can also be replicated to support frequent queries.
- In the analytics module, a time-series oriented database can be replicated for historical data review.

Trade-offs and Challenges:

- Synchronization between the primary and replicas may introduce replication lag.
- Not all operations can be performed on replicas (read-only).
- Higher storage consumption.

4. Parallel Execution of Queries and Analytical Processes

This technique divides complex analytical or query operations into subtasks that run in parallel across data partitions. It significantly improves response time in large-scale data environments.

Project Application:

- In the big data and analytics module (Workshop 2), ETL processes and metric visualization benefit from parallel execution, reducing processing time for logs, usage patterns, and storage consumption.
- Queries to the datalake or data warehouse can be parallelized using time ranges, user segments, or regions.

Trade-offs and Challenges:

- Requires efficient coordination to merge partial results.
- Higher CPU and memory usage.
- Poorly distributed tasks may become bottlenecks if not balanced properly.

General Considerations

These strategies should be implemented considering challenges already identified in previous workshops:

- **Eventual vs. strong consistency:** especially important for replication and sharding scenarios.
- **Operational complexity:** maintaining integrity across distributed services (e.g., FileService, SharingService) in parallelized architectures.
- **Distributed transactions:** needed when operations involve multiple nodes or services; coordination mechanisms such as *Two-Phase Commit (2PC)* or the *SAGA pattern* should be considered in critical flows.

References

Amazon Web Services. (n.d.). *What is data architecture?* AWS. Retrieved May 13, 2025, from https://aws.amazon.com/es/what-is/data-architecture/

Business model canvas examples. (n.d.). *Corporate Finance Institute*. Retrieved March 17, 2025, from https://corporatefinanceinstitute.com/resources/management/business-model-canvas-examples/

IBM. (n.d.). *What is data architecture?* IBM. Retrieved May 13, 2025, from https://www.ibm.com/es-es/topics/data-architecture

UNIR. (n.d.). *Entity-relationship model*. UNIR Revista. Retrieved May 13, 2025, from https://www.unir.net/revista/ingenieria/modelo-entidad-relacion/

Workspace, G. (n.d.). *Google Drive: Share files online with secure cloud storage*. Google Workspace. Retrieved March 17, 2025, from

https://workspace.google.com/intl/es-419/products/drive/