(An Overview of) Cloud Storage

Renaud Lachaize

Univ. Grenoble Alpes

M2 MoSIG

December 2025

Outline

- Block storage
- File storage
- Object storage
- Database storage

Block storage

- **Examples:**
  - Cloud services: AWS Elastic Block Store, Google Persistent Disk
  - OpenStack Cinder Block Service
- Provides a block device abstraction (**akin to a hard disk in a single computer**).
- This is the **lowest abstraction level**. A block device is typically formatted and managed by a (local) file system.
- Such a block device:
  - Is accessed remotely (over the network), in the same datacenter
  - Can only be attached to a single virtual machine at a time
  - Typically supports snapshots
- **Use cases:** system volumes, data volumes (e.g., for databases)

File storage

- **Examples:**
  - Cloud services: AWS Elastic File Service, Google Filestore
  - Network File System (NFSv3, NFSv4)
  - HDFS, Lustre
- Unlike a local file system stored on a block device, this abstraction **allows several client machines to simultaneously have access to the same file system**
- Typically supports a locking service to manage concurrent file accesses
- Various kinds of file systems are available
  - Various consistency and fault tolerance semantics
  - General purpose vs. optimized for specific types of workloads (e.g., big data, HPC)

- **Warning:** Completely different requirements than personal cloud storage systems (e.g., Dropbox, Google Drive)

Object storage

- (Sometimes also called **“blob storage”**)
- **Examples:**
  - Cloud services: AWS S3 (“Simple Storage Service), Google Cloud Storage
  - Open-source frameworks: OpenStack Swift, Garage, MinIO
- **Main goals:**
  - Scalability (number of data items, size of data items, number of clients)
  - Fault tolerance, durability and high availability (through redundancy)
  - Simplicity (simple to use & to scale)
  - Genericity (very diverse use cases: web sites/applications, multimedia, system images, backups, data lakes, analytics …)
  - Cost effectiveness
  - **Typically supported (in public clouds) as several storage “classes”**, corresponding to various requirements in terms of performance, durability, availability, cost.

Object storage – Main characteristics (1/2)

- **Relies on the notion of ”object”**
  - Somewhat different from other traditional abstractions (block, file, database tuple).
  - Some similarities with the concept of key-value database (yet with some differences in features).
  - **An object is a generic container of bytes, with a variable size (potentially very big).**
  - **Versatile support for very different object sizes (from bytes to several TBs)**
  - Each object is also associated with a few kBs of metadata stored in a key-value format. Some of these metadata are system defined (e.g., last modification date). Others are flexible (user defined).

Object storage – Main characteristics (2/2)

- **Loosely structured namespace**
  - Each object is identified by a globally unique, user-defined key (text string of variable length)
  - Each object belongs to a given storage container named “bucket”
  - **The bucket namespace is flat** (i.e., there is no hierarchy, unlike in a file system).

- **Optional features:**
  - Versioning (per object)
  - Access logging (per-object)
  - Access control lists (typically per-bucket)
  - Public access from the Internet, via a URL

Object storage – Interface (1/4)

- Main interface: accessible via a ”Web service” interface (**HTTP-based, REST API**)
- (A client application can also mount a bucket as a file system … but not fully compliant with POSIX file system semantics, and also some features of object storage will not be available)
- Main operations (for buckets, objects and metadata) : **“CRUD”**
  - Create, Read, Update, Delete
  - In the case of objects, “Update” = full overwrite

- Each object is uniquely identified with the combination of bucket name + key (+ optional version ID)
- Although a bucket is flat namespace, it is somehow possible to “simulate” the notion of “folder” by using of a common prefix in the name of various objects (e.g., “logs/”)
- Data at rest can be encrypted on the client side or on the server side

Object storage – Interface (2/4)

- **Other operations:**
  - Partial read of an object
  - Conditional read of an object (if-modified-since, if-unmodified-since …)
  - Copy of an object
  - Delete multiple objects at once
  - Set object/bucket in “lock mode”, preventing future deletion or update (e.g., to enforce a given retention threshold)

Object storage – Interface (3/4)

- The AWS S3 API (although not standardized) has become a de-facto standard, implemented by most systems
  - Note however that, although this facilitates porting applications from one system to another, object storage systems are often not fully compatible and thus not always easily interchangeable, because advanced details/features have significant differences.

Object storage – Interface (4/4)

- **Additional details** - For most object storage systems:
  - Updates to a single object are guaranteed to be atomic (no partial writes)
  - ... However, they may not become immediately visible (see description of consistency semantics on the next slides)
  - There are no atomicity/transactional guarantees across several objects
  - The system does not provide any object locking facility to manage concurrent requests.
  - Note that, given the limited consistency model provided by many object storage systems, it is generally not possible/practical to build a locking service on top of them.
  - Instead, if a locking service is needed, it is more appropriate to use an external locking service.

Object storage – Consistency (1/4)

- **Some Cloud-based object storage systems only provide limited/weak consistency guarantees.** This means, for example, that, in some situations, a read request may return an outdated view of the data.

- This is due to designed choices related to the management of replication, and the emphasis given to scalability.

- However, consistency guarantees of object storage are being improved (i.e., becoming stronger) due to the growing popularity and diversity of use cases for object storage, and the productivity/correctness benefits for application developers.

Object storage – Consistency (2/4)

- Terminology

  - **“Eventual consistency”** corresponds to a weak level of consistency. It only provides the following guarantees:

    - An attempt to read an existing object may return the current value of the object, or an older version, or NULL.

    - If we stop updating the object, all its copies (“replicas”) will eventually converge to the same state, after a finite amount of time (but we do not know the precise upper bound for this time interval).

  - In contrast, **“read-after-write”** semantics provide stronger consistency guarantees.

Object storage – Consistency (3/4)

- **Case study #1: AWS S3 – Before December 2020**
  - **S3 generally provides read-after-write consistency**
  - … **except in the following situations, which only guarantee eventual consistency**:
    - Modification of an existing object (overwrite or delete) followed by a read
    - Attempting to read an object that does not exist, followed by a creation of the object, followed by a read of the object

Illustration (quotes from the old version of the S3 official documentation):

- *"S3 achieves high availability by replicating data across multiple servers within Amazon's data centers. If a PUT request is successful, your data is safely stored.* ***However, information about the changes must replicate across S3, which can take some time, and so you might observe the following behaviors****:*
  - *A process writes a new object to S3 and immediately lists keys within its bucket. Until the change is fully propagated, the object might not appear in the list.*
  - *A process replaces an existing object and immediately attempts to read it. Until the change is fully propagated, S3 might return the prior data.*
  - *A process deletes an existing object and immediately attempts to read it. Until the deletion is fully propagated, S3 might return the deleted data.*
  - *A process deletes an existing object and immediately lists keys within its bucket. Until the deletion is fully propagated, S3 might list the deleted object."*

Object storage – Consistency (4/4)

- **Case study #2: AWS S3 – Since December 2020**
  - "**all S3 GET, PUT, and LIST operations, as well as operations that change object tags, ACLs, or metadata, are now strongly consistent**. What you write is what you will read, and the results of a LIST will be an accurate reflection of what's in the bucket."

Illustration (quotes from the current version of the S3 official documentation):

- *"S3 achieves high availability by replicating data across multiple servers within Amazon's data centers. If a PUT request is successful, your data is safely stored.* ***Any read (GET or LIST request) that is initiated following the receipt of a successful PUT response will return the data written by the PUT request.*** *Here are examples of this behavior:*
  - *A process writes a new object to Amazon S3 and immediately lists keys within its bucket. The new object appears in the list.*
  - *A process replaces an existing object and immediately tries to read it. Amazon S3 returns the new data.*
  - *A process deletes an existing object and immediately tries to read it. Amazon S3 does not return any data because the object has been deleted.*
  - *A process deletes an existing object and immediately lists keys within its bucket. The object does not appear in the listing."*

- For more details:
  - https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html#ConsistencyModel
  - https://www.allthingsdistributed.com/2021/04/s3-strong-consistency.html
  - https://blog.kylegalbraith.com/2021/01/12/the-s3-consistency-model-got-an-upgrade/

Object storage – Wrap-up

- **Main strengths**
  - Simplicity
  - Versatility (use cases, scale, guarantees, access model …)
  - Cost
- **Main weaknesses**
  - Some systems **only provide limited guarantees regarding data consistency and the management of concurrent operations**. This may not be appropriate for certain kinds of applications, in particular with the following characteristics:
    - Frequently modified objects and freshness requirements
    - Concurrent read-write accesses to the same objects
  - Poorer performance (w.r.t. latency and bandwidth) compared to some other types of storage services (especially lower-level and/or more closely-coupled interfaces like block storage)

Databases (1/3)

- There are many types of databases for cloud-based systems

- **Main aspects to be considered:**
  - Managed (by the Cloud provider) or not
  - Data model (see details on the next slides)
  - Transactional features or not
  - Scalability (volume of data, number of requests) & sharding
  - Replication & fault-tolerance
  - Geographic scale (single datacenter vs. geo-replication)
  - Design trade-offs in terms of consistency, performance and availability

Databases (2/3)

There is a **growing diversity of database features**, due to :

- A growing number of data-intensive applications, with diverse requirements.

- A growing resort to storing derived data (i.e., multiple views/formats for the same initial piece of information), in order to facilitate/accelerate some types of data analysis.

- The rise of software architectures based on “micro-services” (a software design pattern that we will study in another lecture), which fosters the use of many small components and encourages the decentralization of the application data into several specialized datastores throughout a software system.

Databases (3/3)

We will give an overview of **several popular data models**:

- Key-Value
- Document
- Relational
- NewSQL
- Graph
- Column family
- Time series
- Streams and queues

Databases – Key/value

- Roughly speaking, akin to a very large and persistent hash table.
- The value is an opaque type.
- There is no schema.
- In some databases, the key can comprise multiple elements and support lookups based on the key prefix.
- Very basic but also usually very fast and scalable.
- Various goals:
  - (1) Many K/V databases are optimized for efficiently storing large volumes of data.
  - (2) Some K/V databases are specialized for storing important configuration data (metadata) and provide strong consistency and fault tolerance properties.
- Examples:
  - (1) Redis, Riak, AWS DynamoDB
  - (2) Etcd, Consul

Databases – Document

- Like a key/value DB, a document DB associates a primary key with a value. But the value must conform to a predefined format.
- This provides the ability to index the data items and to issue queries based on their values.
- However, a document DB does not rely on a schema. Each document can possibly have a different structure.
- A popular format for the document is JSON. Many documents are a composition of hashmaps (JSON objects) and lists (JSON arrays).
- Other popular formats are XML and BSON.
- Examples :
  - MongoDB, CouchBase
  - AWS DocumentDB, Google Cloud DataStore/FireStore

Databases – Relational

- Well-known, traditional model for SQL-based DBs.
- Data organized into two dimensional structures called tables (consisting of columns and rows).
- The inserted data items (“tuples”) must conform to the predefined schema of each table (number of columns and corresponding types).
- Most relational DBs support ACID transactions.
- Examples :
  - PostgreSQL, MySQL, MariaDB
  - AWS RDS, Google Cloud SQL (managed versions of the above ones)

Databases – NewSQL

- A new, emerging kind of relational databases.
- Support the main traditional features of relational DBs, like SQL queries and ACID transactions (sometimes with restrictions/variants).
- Are **specifically designed to provide high performance and/or high availability despite massive data scale and/or large geographic scale** (distribution/replication over multiple data centers).
- Examples :
  - CockroachDB
  - Google Cloud Spanner, Microsoft Azure CosmosDB

Databases – Graph

- The data model is based on two types of information: nodes (~ entities) and edges (relationships between nodes). Specific properties can be attached to each edge or node.
- Useful data model for analyses based on the relationships between data items.
- The storage format of such DBs is optimized for fast graph traversal (potentially much more efficient that joins between tables in a relational DB).
- Examples :
  - Neo4j, Dgraph
  - AWS Neptune

Databases – Column family

- Organizes data into rows and columns, like a relational DB. But :
  - Instead of tables, the DB uses structures named “column families”.
  - Each row defines its own format.
  - Rows can be sparse.
- Can also be described as a special key/value DB in which each key (row identifier) is associated with a dictionary of arbitrary attributes and their values.
- Good/useful for applications requiring great performance for row-based operations. (In contrast to joins or aggregations.)
- Examples :
  - Cassandra, HBase
  - AWS Redshift, Google Cloud BigTable

Databases – Time series

- A time series DB is optimized to store records based on time (e.g., telemetry data).
- Records are often small but plentiful.
- Such a DB often must support a very high number of write operations in real time (insertion of new records).
- Updates are rare, and deletions are performed in bulk.
- Such a DB often provides flexible support for data retention and down-sampling.
- Examples :
  - InfluxDB, Graphite
  - AWS Timestream

Databases – Streams & queues

- Store events and messages.
- There are significant differences between them.
- **Streams**
  - Data is stored as an immutable stream of events.
  - Typically used for data integration across heterogeneous services. Multiple services can consume data from the same stream without interfering on each other.
  - Example: Kafka
- **Queues**
  - Store messages/events that can be changed or removed.
  - Typically used for communication between services (e.g., dispatching requests and replies).
  - Example: RabbitMQ

Databases – For further details

- Prisma blog. Comparing Database Types: How Database Types Evolved to Meet Different Needs. https://www.prisma.io/blog/comparison-of-database-models-1iz9u29nwn37

- Martin Kleppmann. Designing Data-Intensive Applications. O'Reilly, 2017. (Chapters 2, 3, 4, 6)

- B. Scholl, T. Swanson, P. Jausovec. Cloud native: Using containers, functions, and data to build next generation applications. O'Reilly, 2019. Chapter 4: Working with data.