IBM Cloud Object Storage System™
Version 3.14.2

Methods to Manage Simple Object Metadata

IBM
**Intended purpose and audience**

Client applications can store Binary Large Objects (BLOB or Objects) in a IBM Cloud Object Storage System™ using a variety of API methods, including Amazon Web Services Simple Storage Service (AWS S3), OpenStack Swift, Web ObjectScaler (WOS) and the IBM Cloud Object Storage System™ Simple Object over HTTP (SOH) API. Each of these methods are defined as the means of storing all of the data in a particular logical storage construct called a Vault. Vaults that use SOH are called Simple Object Vaults (SOV) and the Objects written to them are called Simple Objects (SO). SOH is unique in that it does not track the metadata, such as the filename, so the client developing applications that use SOH need to determine how they will store and track this metadata. There are three methods to store metadata using a IBM Cloud Object Storage System™ which are explored in this guide.
Chapter 1. How simple objects are stored

An IBM Cloud Object Storage System™ links systems to store data that is divided into multiple slices by using an Information Dispersal Algorithm (IDA) before sending each slice to a storage node. The storage node is called an IBM Cloud Object Storage Slicestor® node. A client application sends object data to a device that slices the data, called an IBM Cloud Object Storage Accesser® node, over an HTTP-based protocol.

When the system stores an object by using the SOH, it assigns the object a unique 18-byte object ID (OID). This OID contains enough information to allow the system to perform a quick read without consulting a centralized routing system.

From the application’s perspective, objects are stored in a large capacity storage system without a hierarchical structure.

If the object changes, it must be written again as a brand new object.
IBM Cloud Object Storage System™: Methods to Manage Simple Object Metadata
Chapter 2. Define content data and metadata

Conceptually, a client application can view an object as both its content data and any associated metadata.

Objects that are written to an SOV cannot be read unless a threshold number of slices are recovered. It means that objects cannot be searched. The SOV type is suitable for storing the content data portion of an object.

Table 1. Characteristics of metadata versus content data

<table>
<thead>
<tr>
<th></th>
<th>Metadata</th>
<th>Content data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scaling</td>
<td>Small relative size, scale up scale or partitioning</td>
<td>Massive relative size, out slice servers</td>
</tr>
<tr>
<td>Search</td>
<td>Indexed, fast search</td>
<td>Binary blob, no search</td>
</tr>
<tr>
<td>Reliability</td>
<td>Replication</td>
<td>Replication</td>
</tr>
</tbody>
</table>

Typically, metadata is smaller in size and must be indexed and searchable. A database is better suited to store metadata because it can be indexed and searched. A database can make data more reliable by replicating it. Any hierarchy, file names, or other such metadata should be stored with the object metadata in an application database.

Content data is often of limited use without its associated metadata. A developer needs to ensure that metadata is stored as reliably as the content data stored on a SOV. Two approaches to accomplish it are available.

- Use built-in replication features of the application database.
- Back up a copy of the metadata to the IBM Cloud Object Storage System™.

A metadata backup cannot be searched so an indexed copy of the metadata must be in the application database.
Chapter 3. Determine how to store metadata

Application developers should use an external database to store object metadata. Client-side metadata operations can involve complex and fast queries that are demanding on subsystems common to many database technologies.

Most databases index fields that are commonly searched. IBM Cloud Object Storage System™ storage is paired successfully with numerous database technologies, including many open- and closed-source RDBMS, key-value stores, and NoSQL cloud document systems.

All objects that are written to the system are immutable by design.

To replace or update the content of an object, a new object must be written again and the old object deleted. Typically, client applications store the OID returned from the system as a column in a database table.

When object content is updated, this OID is updated in that database table. Applications store object metadata as other columns within the application database. It allows the application to indexing and query the metadata.

Metadata is as valuable as the object data and must be stored reliably. Three methods for achieving reliable metadata storage are available.

**Database replication**
- Replicate the database if the size of metadata is insignificant compared to content data size.

**Metadata dispersal**
- Store bundled and non-searchable metadata in an SOV either:
  - **Covariant metadata dispersal**
    - Updates metadata only when the object updates.
  - **Independent metadata dispersal**
    - Updates metadata whenever an update is needed.

**Replicate the database**

Database replication is used on many database management systems (DBMS), usually with a master/slave relationship between the original and the copies. The master logs the updates, which then propagate to the slaves. The slave outputs a message that states that it received the update successfully, thus allowing the sending (and potentially resending until successfully applied) of subsequent updates.

Multi-master replication, where updates can be submitted to any database node and then propagate to other servers, often is wanted. It introduces substantially increased costs and complexity, which might make it impractical in some situations. The most common challenge in multi-master replication is transactional conflict prevention or resolution. Most synchronous or eager replication solutions prevent conflicts, while asynchronous solutions resolve conflicts.

For instance, if a record is changed on two nodes simultaneously, an eager replication system would detect the conflict before it confirms the commit and cancels one of the transactions. A lazy replication system would allow both transactions to commit and run a conflict resolution during resynchronization. The resolution of such a conflict can be based on a timestamp of the transaction, on the hierarchy of the origin nodes, or on much more complex logic, which decides consistently on all nodes.
Database replication increases in difficulty as it scales. Usually, the DBMS is scaled in two dimensions, horizontal and vertical:

- Horizontal scaling has more data replicas. A multi-layer multi-view access protocol can alleviate horizontal scaling issues.
- Vertical scaling has data replicas that are located further away in distance. The steady and continuous improvements in internet reliability and performance make vertical scaling less challenging.

The steady and continuous improvements in internet reliability and performance make vertical scaling less challenging.

![Ingest Client](image1)

**Figure 1. Depiction of database metadata storage**

### Disperse metadata with object data

Immutable objects are unchanging discrete object data and metadata stored to a storage system.

On such systems, object metadata is determined one time at object creation time. These systems can update metadata, but only when new object data is written. When metadata changes only when the object data changes, it is known as **covariant metadata**.

Covariant metadata can be stored within the same Simple object as the object data itself. With covariant metadata, the metadata is either added as a prefix or appended to object data.

Ranged reads can be used without penalty for selective reads.

**Metadata prefix**

For fixed-size metadata, adding a prefix is advisable as it allows no-penalty ranged reads for the first byte of content data. Metadata read would be from zero-offset (normal read).

For variable sized metadata, read-penalty on reads might exist, since the first byte must always be read to determine data offset.

```c
struct {
    long metadata_length,
    byte[] metadata,
    byte[] content_data
};
```

**Appending metadata**

For variable sized metadata, storing metadata at end of content data allows no-penalty object ranged reads.
**Disperse metadata independently of object data**

Mutable data objects are discrete sources of changeable object data and metadata stored to a storage system.

On such systems, object metadata needs to be changed independently from, and often more frequently than, content data.

These systems need a way to reliably store metadata updates without rewriting the entire object’s data. When metadata changes on a separate schedule than when the object data changes, it is known as *independent metadata*.

With independent metadata, the metadata is either added as a prefix or appended to object data. Metadata is stored reliably as independent metadata bundle objects to an SOV. These metadata bundles need to indicate which object ID they relate to and also need revision information to help the disaster recovery process. Some slight changes in content data storage also are needed to allow content data objects to be ignored during recovery.

It shows storage formats as C structures:

```c
struct metadata_Object {
    byte METADATA_MAGIC_NUMBER
    long database_primary_key, Long revision,
    byte[] metadata content
};
struct content_Object {
    byte CONTENT_MAGIC_NUMBER
    byte[] content_data
};
```

Example of application database schema, which would allow for object tracking:

```sql
-- ---
-- Table 'DispersedStore'
-- ---
DROP TABLE IF EXISTS `DispersedStore`;
CREATE TABLE `DispersedStore`(
  `permanent_id` VARCHAR NOT NULL,
  `Object_id` VARCHAR NOT NULL,
  `metadata_id` VARCHAR NOT NULL,
  `metadata_revision` INTEGER NOT NULL DEFAULT 0, PRIMARY KEY (`permanent_id`))
```

Every time the metadata is updated for an object, the entire set of metadata is bundled and stored as a new object. It results in a new metadata OID to insert into the application database.

**Bundling metadata as a new object**

**About this task**

This procedure has a number of desirable properties, not least of which is the resiliency against client failures mid-transaction. The use of a monotonically incrementing revision number allows disaster recovery to insert the newest revision during database restoration.
**Procedure**

1. Start a database transaction.
2. Update database with new metadata.
3. Generate a new revision strictly greater than the old metadata revision. The old revision can be obtained from the database table.
4. Generate a new metadata bundle. A bundle consists of metadata in an application-parseable representation, with the new revision number and the permanent OID generated by the application.
5. Store this bundle as a new object.
6. Update metadata OID and metadata revision in the database.
7. Commit database transaction.
8. Remove old metadata object from SOV.

**Recover metadata as part of disaster recovery**

The revision number that was stored earlier is used to avoid reverting to stale metadata if an old metadata object bundle was orphaned during metadata update failure.

1. Listing an SOV returns a list of OIDs, which the recovery agent should iterate over.
2. For each OID, start reading the object and interpret the first byte.
   a. If the first byte is content magic number, continue listing.
   b. If the first byte is metadata magic number, interpret in permanent ID and metadata revision.
3. Query the database for the row with given permanent ID.
   a. If row does not exist, or if metadata revision in table is less than the revision found in the current metadata bundle, restore metadata.
   b. Otherwise, skip the object as the newest metadata is already stored.

With this technique, every metadata update must bundle of the entire metadata set and write a brand new object.

An alternative is to bundle metadata at a more granular level. It requires extra metadata tracking columns in the database. For small and frequent metadata updates, this technique can be modified to batch metadata updates to the SOV. There exists a tradeoff between performance and the freshness of metadata after disaster recovery.

![Diagram](image.png)

*Figure 2. Depiction of metadata dispersal*
Trademarks

IBM®, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at Copyright and trademark information at [www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml).

Accesser®, Cleversafe®, ClevOS™, Dispersed Storage®, dsNet®, IBM Cloud Object Storage Accesser®, IBM Cloud Object Storage Dedicated™, IBM Cloud Object Storage Insight™, IBM Cloud Object Storage Manager™, IBM Cloud Object Storage Slicestor®, IBM Cloud Object Storage Standard™, IBM Cloud Object Storage System™, IBM Cloud Object Storage Vault™, SecureSlice™, and Slicestor® are trademarks or registered trademarks of Cleversafe, an IBM Company and/or International Business Machines Corp.

Other product and service names might be trademarks of IBM or other companies.
Index

T
   trademarks  9