A Detailed enquiry and assessment into Building Domain Specific ODS using AWS S3 like Object Store

Baiju Thakkar, Shak Kathirvel

Home and Auto Lending Architecture Group [HALT]
JP Morgan Chase Consumer and Community Banking [CCB]

Abstract

The objective of this research report is to examine and do a detailed enquiry into the complexities and opportunities associated with implementing key-value data store atop standard cloud object store platform. It examines a architectural pattern, challenges, design compromises for specific use cases. The analysis highlights the inherent trade-offs in consistency, performance, and scalability across these approaches, underscoring that while object storage offers unparalleled foundational benefits, transforming it into a fully functional key-value database for niche application needs often requires significant architectural augmentation or reliance on higher-level managed services.

Motivation & business use cases to build a simple KV datastore

For general-purpose transactional or high-performance OLTP application workloads, developing, building and operating a custom key-value data store using an object store (like AWS S3, Google Cloud Storage, or Azure Blob Storage) is generally quite complex endeavor. A dedicated key-value databases like DynamoDB, Redis, or Apache Cassandra are purpose-built for those scenarios and offer significantly better performance, consistency models, and features.

Rationale

However, there are still niche, narrow, specialized business-specific use cases, say an **Domain specific Operational Data Store DS-ODS**, where the standard KV databases may be an **overweight** and may not fit a business specific use cases. In those cases, **a tailor-made light-weight DS-ODS** object store backed simple key-value Database middleware implementing the basics of a standard DB primitives, perhaps with additional layers supporting minimum transactional needs, is rightly justifiable.

ODS Data Access Patterns

1. **Single Record Lookups (by Primary Key / Unique Identifier):**
   Access Pattern: Applications or users need to retrieve a complete record for a specific entity (e.g., a customer, an order, a product) using its unique identifier.
   Access Purpose: To get the most up-to-date, holistic view of a single entity.
   Access Velocity: High frequency, low latency, direct access.
2. **Small Range Scans / Filtered Lookups (by Secondary/Indexed Attributes):**
   Access Pattern: Retrieving a small set of records based on non-primary key attributes that are often indexed. These are not full table scans but targeted queries.
   Access Purpose: To find specific sets of related operational data.
   Access Velocity: Moderate frequency, low to moderate latency, requires efficient indexing.
3. **Small Range Scans / Filtered Lookups (by Secondary/Indexed Attributes):**
   Access Pattern: Retrieving a small set of records based on non-primary key attributes that are often indexed. These are not full table scans but targeted queries.
   Access Purpose: To find specific sets of related operational data.
   Access Velocity: Moderate frequency, low to moderate latency, requires efficient indexing.
4. **Real-Time (or Near Real-Time) Data Integration/Synchronization (Writes):**
   Access Pattern: The ODS itself is primarily a target for data integration. It receives data from various source systems, often in real-time or near real-time, to maintain its current state.
   Access Purpose: To consolidate and cleanse data from disparate sources into a single, unified view.
   Access Velocity: High volume of in-

serts/updates, low latency for writes, requires robust ETL/ELT or streaming capabilities. The ODS must be optimized for efficient data ingestion.

5. **Ad-hoc Operational Queries:**
   Access Pattern: Business users or analysts performing spontaneous, often one-off queries to investigate specific operational issues or answer immediate business questions.
   Access Purpose: Quick diagnostics and problem-solving.
   Access Velocity: Unpredictable, can vary in complexity, requires flexible query tools (e.g., SQL client, self-service BI tools).
6. **Data Feeds for Downstream Systems:**
   Access Pattern: The ODS often serves as a source for other downstream systems, such as data warehouses, data marts, or other operational applications that need a consistent view of consolidated data.
   Access Purpose: To provide clean, integrated data for further analytical processing or to synchronize other operational systems.
   Access Velocity: Often batch-oriented (though can be streaming), involves extracting and potentially further transforming data, high volume of reads from the ODS.
7. **Data Quality and Data Governance Checks:**
   Access Pattern: Automated processes or manual reviews that access data in the ODS to monitor data quality, identify inconsistencies, and ensure compliance with governance rules.
   Access Purpose: To maintain the integrity and reliability of the operational data.
   Access Velocity: Regular, often scheduled queries; can involve complex logic for validation; may trigger alerts or data correction workflows.

Specific Use Cases for Custom DS-ODS using KV DB with object backend

1. **Usecase:** You have millions of small-to-medium sized files (completed legal documents, billing or usage records, aged transaction records) that are written once and rarely modified. You need to retrieve specific items quickly by a unique identifier, but not necessarily query across attributes within the items.

   Example: A customer's billing and payment history in JSON file format

   Why S3 or equivalent object store: The primary need is cheap, highly durable storage for vast quantities of immutable data. The "key" (S3 object key) serves as the unique identifier for direct retrieval. You don't need complex transactional integrity or real-time analytical queries.

2. **Usecase:** You have a large number of user profiles or long form application submission materials that are accessed frequently during initial business stages and infrequently in the later business stages (e.g., for interactive processing, or when a user logs in after a long period of inactivity). When accessed, you need the full profile/application state data.

   Example: Hundreds, Thousands of active user application states during long business process journey that are accessed frequently. Millions of the finished application states that are accessed infrequently or rarely or never.

   Why S3 or equivalent object store: If the data volume is very large and the access pattern is sparse (e.g., 90% of rarely accessed data are cold), storing them in S3 can be significantly cheaper than a hot database. You would use a dedicated cache layer (like memcached) for frequently accessed, "hot" accessed data and move cold data to S3 via lifecycle policies or custom logic.)

3. **Usecase:** "Data Lake" Micro-files with Known Access Patterns: Part of a larger data lake architecture where specific application jobs or functions need to retrieve individual data points or small data chunks by a predefined key/path.

   Example: A segment of a data lake storing pre-computed application state for individual customer segments.

   Why S3 or equivalent object store:: S3 is the backbone of data lakes. If your application workload primarily involves "get by ID" on small, pre-processed files rather than complex SQL-like queries on raw data, S3 can efficiently serve those lookups.

4. **Usecase:** Ephemeral Data or Scratch Space for Batch Processing: As a temporary store for intermediate results in long-running batch jobs, where the data needs to be retrieved by a specific job ID or part ID. The data is eventually discarded or moved.

   Why S3 or equivalent object store:: Its scalability and durability make it robust for handling

large intermediate data sets, and its object key serves as a natural identifier.

What core Database primitives are needed?

The special purpose S3-like Object-store based DB should implement primitives operations and special protocols **store, read, delete and update primitives and indexes using S3-like Object-store**. The ultimate goal is to preserve the scalability and availability of a distributed system like object store and achieve the same level or near-same level of consistency as a database system (i.e., ACID transactions).

Unfortunately, it is not possible to have it all because of Brewer's famous CAP theorem [1]. Given the choice, this investigation follows the distributed systems' approach, thereby preserving scalability and availability and **maximizing the level of consistency** that can be achieved under Brewer's CAP constraint.

As a result, we will not even try to support **full ACID transactions** because we feel that it is not needed for most Web-based applications [2], whereas scalability and availability are a must. We will enquire/analyze how **basic and necessary transactional properties** (e.g., atomicity and durability) can be implemented;

When NOT to use S3-like as a Key-Value Store

1. Frequent Updates/Writes: S3 requires overwriting the entire object for any change, which is inefficient.
2. Strong Consistency Requirements for Updates: S3 is eventually consistent for overwrites, meaning a read after an update might return stale data.
3. Complex Querying/Indexing: If you need to search or filter data based on attributes within the "value" (object content) or join data across multiple "keys," S3 alone is insufficient.
4. Low Latency (Sub-10ms): While S3 is fast, dedicated key-value stores are designed for much lower single-digit millisecond latencies.
5. Transactional Integrity (Multi-item writes): S3 doesn't offer ACID transactions across multiple objects.
6. Small Objects with High Read/Write Throughput: While S3 can store small objects, the per-request cost can become higher than a dedicated database for extremely high throughput on very small items.
7. Complex Data Models (e.g., nested documents, lists, sets): While you can store these as JSON in S3, you lose the native data type support and atomic operations offered by NoSQL databases.

Why chose S3-like Object Stores backed KV database for Niche Use Cases:

1. Simplicity of Core Operations: For simple Put/Get/Delete by key, the API is straightforward.
2. Direct Accessibility: Objects can often be directly accessed via HTTP/HTTPS, which can simplify some distribution scenarios.
3. High Durability and Availability: They are designed for 11 nines of durability (extremely low chance of data loss) and high availability, making them suitable for critical, long-term storage.
4. Massive Scalability (Storage): Object stores can handle virtually unlimited amounts of data. You never have to worry about provisioning storage capacity.
5. Extremely Low Storage Cost: For vast amounts of data that are accessed infrequently, object storage is orders of magnitude cheaper per GB than managed databases.

Architecture

Table/Collection, Pages, Records, Objects, Keys and Indices

Figure 1 shows the high level architecture of an S3-backed KV database. This architecture has a great deal of commonalities with a distributed shared-disk database system [3]. Clients retrieve pages from S3 based on the pages' URIs, buffer/cache the pages locally, update them, and write them back.

Each record is associated to a Table/collection. A record is composed of a key and payload data. The key uniquely identifies the record within its collection. Both key and payload are bytestreams of arbitrary length; the only constraint is that the size of the whole record must be smaller than the page size. Physically, each record is stored in exactly one page which in turn is stored as a single object in S3.

Logically, each record is part of a collection (e.g., a table). A collection is implemented as a bucketname+collection-name suffix (eg. s3://<bucketname>/collection-name in S3 and all the pages that store records of that collection are stored as S3 objects in that bucket. A collection is identified by a URI.

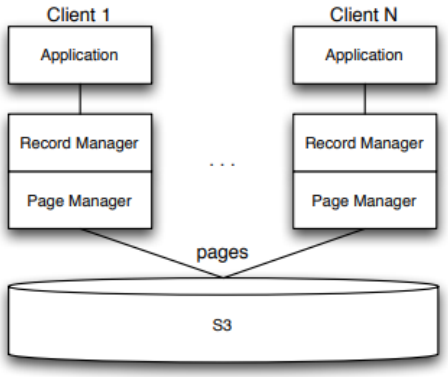

Figure 1: Shared-disk Architecture

The unit of transfer is a **page** which contains typically several records or index entries. Following the general DB terminology, we refer to records as a bytestream of variable size whose size is constrained by the page size. **Records** can be standard JSON or equivalent structured document.

Record and Page Managers

From the client's perspective, there is a stack of components that support the application. It contains two lowest layers; i.e., the **record and page managers**. All other layers (e.g., the query processor) are not affected by the use of S3 and are, thus, considered to be part of the application.

It is assumed that application, record, and page manager run on a single machine and in a single process.

The **record manager** provides a record-oriented interface, organizes records on pages, and carries out free-space management for the creation of new records. Applications interact with the record manager only, using the interface defined below.

The page manager coordinates read and write requests to S3 and buffers pages from S3.

Internals of Record Manager

The record manager provides functions to create new objects, read objects, update objects, and scan collections.

- **Create(key, payload, uri)**: Creates a new record into the collection identified by uri. In our implementation, free-space management is carried out using a **B-tree**; this approach is sometimes also referred to as index-organized table. That is, the new record is inserted into a leaf of a B-tree. The key must be defined by the application and it must be unique.

- **Read(key, uri)**: Reads the payload information of a record given the key of the record and the URI of the collection.

- **Update(key, payload, uri)**: Update the payload information of a record. In this study, all keys are immutable. The only way to change a key of a record is to delete and re-create the record.

- **Delete(key, uri)**: Delete a record.

- **Scan(uri)**: Scan through all records of a collection. To support scans, the record manager returns an iterator to the application.

  In addition to the create, read, update, and scan methods, the API of the record manager supports commit and abort methods.

  These two methods are implemented by the page manager, described in the next section. Furthermore, the record manager exposes an interface to probe B-tree indexes (e.g., range queries); the implementation of B-trees is described in section below.

Internals of Page Manager

Page Manager supports reading pages from S3, pinning the pages in the buffer pool, updating the pages in the buffer pool, and marking the pages as updated. The page manager also provides a way to create new pages on S3.

Additionally, the page manager implements the **commit and abort methods**. We refer to the term **transaction** for a sequence of read, update, and create requests between two commit or abort calls. It is assumed that the write set of a transaction (i.e., the set of updated and newly created pages) fits into the client's main memory.

If an application commits, all the updates are propagated/flushed to S3 and all the affected pages are marked as unmodified in the client's buffer pool. If the application aborts a transaction, all pages marked modified or new are simply discarded from the client's buffer pool, without any interaction with S3. The commit protocols for this design do not give ACID guarantees in the DB sense.

The page manager keeps copies of S3 pages in the buffer pool across transactions. That is, no pages are evicted from the buffer pool as part of a commit. (An abort only evicts modified and new pages.) Pages are refreshed in the buffer pool using a time to live

(TTL) protocol: If an unmodified page is requested from the buffer pool after its time to live has expired, the page manager issues a get-if-modified request to S3 in order to get an up-to-date version, if necessary.

B-tree Indexing

The root and intermediate nodes of the B-tree are stored as pages on S3 (via the page manager) and contain (key, uri) pairs: uri refers to the appropriate page at the next lower level. The leaf pages of a primary index contain entries of the form (key, payload); that is, these pages store the records of the collection in the index-organized table.

The leaf pages of a secondary index contain entries of the form (search key, record key). That is, probing a secondary index involves navigating through the secondary index in order to retrieve the keys of the matching records and then navigating through the primary index in order to retrieve the records with the payload data.

Holding locks must be avoided as much as possible in a scalable distributed architecture. The use of B-link trees allow concurrent reads and writes. That is, each node of the B-tree contains a pointer (i.e., URI) to its right sibling at the same level. At the leaf level, this chaining can naturally be exploited in order to implement scans through the whole collection or through large key ranges.

A B-tree is identified by the URI of its root page. A collection is identified by the URI of the root of its primary index. Both URIs are stored persistently as metadata in the system's catalogue on S3.

Since the URI of an index is a reference to the root page of the B-tree, it is important that the root page is always referenced by the same URI. Implementing this requirement involves a slightly modified, yet straightforward, way to split the root node. Another deviation to the standard B-tree protocol is that the root node of a B-tree in S3 can be empty; it is not deleted even if the B-tree contains no entries.

Challenges with 'Page' commit

This section addresses one particular issue which arises when concurrent clients commit updates to records stored on the same page. If no care is taken, then the updates of one client are overwritten by the other client, even if the two clients update different records. The reason is that the unit of transfer between clients and S3 in the architecture of Figure 1 is a page, rather than an individual record.

This issue does not arise in a (shareddisk) database system because the database system coordinates updates to the disk(s); however, this coordination limits the scalability (number of nodes/clients) of a shared-disk database.

This issue does not arise in the way that S3 is used conventionally because today S3 is mostly used to store large objects so that the unit of transfer to S3 can be the object; for small objects, clustering several objects into pages is mandatory in order to get acceptable performance. Obviously, if two concurrent clients update the same record, then the last updater wins

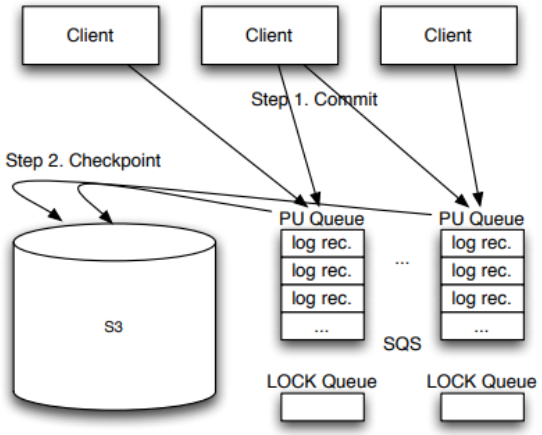

Figure 2: Basic Commit Protocol

Page Commit Protocol - A Custom Protocol to gain eventual consistency

Figure 2 demonstrates the basic idea of how clients commit updates. The protocol is carried out in two steps:

- In the first step, the client generates log records for all the updates that are committed as part of the transaction and sends them to SQS
- In the second step, the log records are applied to the pages stored on S3. We call this step checkpointing

This protocol is extremely simple, but it serves the purpose. Assuming that SQS is virtually always available and that sending messages to SQS never blocks, the first step can be carried out in constant time (assuming a constant or bounded number of messages which must be sent per commit). The second step, checkpointing, involves synchronization but this step can be carried out asynchronously and outside of the execution of a client application. That is, end users are never blocked by the checkpointing process. As a result, virtually 100 percent read, write, and commit availability is achieved, independent of the activity of

concurrent clients and failures of other clients. Furthermore, no additional infrastructure is needed to execute this protocol; SQS and S3 are both utility services provided by AWS, and the hope is that similar utility services will be supported by other providers soon.

The simple protocol of Figure 2 is also resilient to failures. If a client crashes during commit, then the client resends all log records when it restarts. In this case, it is possible that the client sends some log records twice and as a result these log records may be applied twice. However, applying log records twice is not a problem because the log records are idempotent. If a client crashes during commit, it is also possible that the client never comes back or loses uncommitted log records. In this case, some log records of the commit have been applied (before the failure) and some log records of the commit will never be applied, thereby violating atomicity. Indeed, the basic commit protocol of Figure 2 does not guarantee atomicity.

The simple protocol of Figure 2 preserves all the features of utility computing. Unfortunately, it does not help with regard to consistency. That is, the time before an update of one client becomes visible to other clients is unbounded in theory. The only guarantee that can be given is that eventually all updates will become visible to everybody and that all updates are durable. **This property is known as eventual consistency.**

In practice, the freshness of data seen by clients can be controlled by setting the checkpoint interval and the TTL value at each client's cache. Setting the checkpoint interval and TTL values to lower values will increase the freshness of data, but it will also increase the $ cost per transaction. Another way to increase the freshness of data (at increased cost) is to allow clients to receive log records directly from SQS, before they have been applied to S3 as part of a checkpoint.

Pending Update Queues

Figure 2 shows that clients propagate their log records to so called PU queues (i.e., Pending Update queues). In theory, it would be sufficient to have a single PU queue for the whole system. However, it is better to have several PU queues because that allows multiple clients to carry out checkpoints concurrently:

1. a PU queue can only be checkpointed by a single client at the same time. Specifically, we propose to establish PU queues for the following structures:
2. Each B-tree (primary and secondary) has one PU queue associated to it. The PU queue of a B-tree is created when the B-tree is created and its URI is derived from the URI of the B-tree (i.e., the URI of the root node of the B-tree). All insert and delete log records are submitted to the PU queues of B-trees.
3. One PU queue is associated to each leaf node of a primary B-tree of a collection. We refer to these leaf nodes as data pages because they contain all the records of a collection. Only update log records are submitted to the PU queues of data pages. The URIs of these PU queues are derived from the corresponding URIs of the data pages.

Checkpoint Protocol for Data Pages

The input of a checkpoint is a PU queue. The most important challenge when carrying out a checkpoint is to make sure that nobody else is concurrently carrying out a checkpoint on the same PU queue.

For instance, if two clients carry out a checkpoint concurrently using the same PU queue, some updates (i.e., log records) might be lost because it is unlikely that both clients will read the exactly same set of log records from the PU queue

One solution to synchronize checkpoints is to designate machines to carry out checkpoints on particular PU queues. This approach is referred to as watchdog or owner, but it is not used in this work because it does not degrade gracefully in the event that one of these designated machines fail. The alternative is to allow any client which has write permission to carry out checkpoints and to implement a lock in order to guarantee that two clients do not checkpoint the same PU queue concurrently.

As shown in Figure 2, such a lock can be implemented using SQS. The idea is to associate a LOCK queue to each PU queue. Both, the PU queue and the LOCK queue are created at the time a new page is created in S3. At every point in time, a LOCK queue contains exactly one message which can be regarded as a token. This token message is created when the LOCK queue is created and nobody is allowed to send messages to a LOCK queue or delete the token message.

When a client (or any other authority) attempts to do a checkpoint on a PU queue, it tries first to receive and lock the token message from the LOCK queue of that PU queue. If the client receives the token message, then the client knows that nobody else is concurrently applying a checkpoint on that PU queue and proceeds to carry out the checkpoint.

As part of the receive request to the LOCK queue, the client sets a timeout to lock the token message. During this timeout period the client must have com-

pleted the checkpoint; if the client is not finished in that timeout period, the client aborts checkpoint processing and propagates no changes. If the client does not receive the token message at the beginning, the client assumes that somebody else is carrying out a checkpoint concurrently. As a result, the client terminates the routine.

In summary, update log records are applied in the following way:

1. receive(URIofLOCKQueue, 1, timeout). If the token message is returned, continue with Step 2; otherwise, terminate. The timeout on the token message should be set such that Steps 2 to 4 can be executed within the timeout period.
2. If the leaf page is cached at the client, refresh the cached copy with a get-if-modified call to S3. If it is not cached, get a copy of the page from S3.
3. receive(URIofPUQueue, 256, 0). Receive as many log records from the PU queue as possible. (256 is the upper bound for the number of messages that can be received within one SQS call.) The timeout can be set to 0 because the log records in the PU queues need not be locked.
4. Apply the log records to the local copy of the page.
5. If Steps 2 to 4 were carried out within the timeout specified in Step 1 (plus some padding for the put), put the new version of the page to S3. If not, terminate.
6. If Step 5 was successful and completed within the timeout, delete all the log records which were received in Step 3 from the PU queue using the delete method of SQS.

Transaction Management

In Progress

Atomicity

Monocity

References

[1] S. Gilbert and N. Lynch. “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant Web services”. In: *SIGACT News* (2002), pp. 51–59.

[2] W. Vogels. “Data access patterns in the Amazon.com technology platform”. In: *In Proc. of VLDB* 2007.1 ().

[3] M. Stonebraker. “The case for shared nothing”. In: . *IEEE Data Engg. Bulletin* 9(1) (1986).