Bringing Cloud-Native Storage to SAP IQ

Mohammed Abouzour, Güneş Aluç, Ivan T. Bowman, Xi Deng, Nandan Marathe, Sagar Ranadive, Muhammed Sharique, John C. Smirnios
SAP

ABSTRACT

In this paper, we describe our journey of transforming SAP IQ into a relational database management system (RDBMS) that utilizes cheap, elastically scalable object stores on the cloud. SAP IQ is a three-decade old, disk-based, columnar RDBMS that is optimized for complex online analytical processing (OLAP) workloads. Traditionally, SAP IQ has been designed to operate on shared storage devices with *strong consistency guarantees* (e.g., high-caliber storage area network devices). Therefore, deploying SAP IQ on the cloud, as is, would have meant utilizing storage solutions such as NetApp or AWS EFS that provide a POSIX compliant file interface and strong consistency guarantees, but at a much higher monetary cost. These costs can accumulate easily to diminish the economies of scale that one would expect on the cloud, which can be undesirable. Instead, we have enhanced the design of SAP IQ to operate on cloud object stores such as AWS S3 and Azure Blob Storage. Object stores rely on a weaker consistency model, and potentially have higher latency; however, because of these design trade-offs, they are able to offer (i) better pricing, (ii) enhanced durability, (iii) improved elasticity, and (iv) higher throughput. By enhancing SAP IQ to operate under these design trade-offs, we have unlocked many of the opportunities offered by object stores. More specifically, we have extended SAP IQ’s buffer manager and transaction manager, and have introduced a new caching layer that utilizes instance storage on AWS EC2. Experiments using the TPC-H benchmark demonstrate that we can gain an order of magnitude reduction in data-at-rest storage costs while improving query and load performance.

CCS CONCEPTS

• **Information systems → DBMS engine architectures**; **Cloud based storage**; **Storage management**.

KEYWORDS

cloud-native storage, caching, garbage collection, snapshots

ACM Reference Format:
Mohammed Abouzour, Güneş Aluç, Ivan T. Bowman, Xi Deng, Nandan Marathe, Sagar Ranadive, Muhammed Sharique, John C. Smirnios. 2021. Bringing Cloud-Native Storage to SAP IQ. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD ’21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457563

1 INTRODUCTION

Database management system vendors are moving their systems to the cloud [52]. The cloud offers elasticity of storage and compute, built-in fault-tolerance and disaster recovery, and a better overall user experience. On the cloud, users of database management systems benefit from a pay-as-you-go pricing model while the vendors of these systems benefit from economies of scale.

Recently, SAP has released the SAP HANA Cloud [17] where users can deploy SAP HANA [34] and/or SAP IQ [18] as software-as-a-service (SaaS) instances on the cloud. SAP HANA and SAP IQ are two complementary products: SAP HANA is an in-memory, columnar RDBMS that is capable of fast query execution over terabytes of data [34] while SAP IQ is a disk-based, columnar RDBMS that is optimized for petabytes of data [18]. In this paper, we describe our journey of transforming SAP IQ into a cloud RDBMS.

SAP IQ is a mature product that benefits from three decades of research and development. In particular, the following techniques enable on-premise deployments of the product to efficiently load, query and transactionally modify data at the petabyte-scale, using tens of compute nodes:

- **Compression**: Columnar data in SAP IQ are compressed using the dictionary-encoding and the *n*-bit representation [47]. In addition, SAP IQ employs page-level compression to further reduce the amount of I/O that is required to process large volumes of data.
- **Partitioning**: Users of SAP IQ can create range-partitioned as well as hash-partitioned tables that have their unique benefits under different types of workloads.
- **Indexing**: SAP IQ supports a wide range of secondary indexes including the (tiered) High-Group (HG) index [21] that combines the power of B^+-trees [28, 38] with the scalability and compression of bitmaps [27]. It uses zone-maps [19] to early-prune pages that are not needed for a query. Furthermore, it supports a wide range of other *niche* indexes (e.g., DATE/TIME/DTTM tailored for datepart queries, CMP for two-column comparisons and TEXT for text indexing).
- **Prefetching**: During query processing, the system relies on prefetching to parallelize I/O as much as possible. Prefetching techniques have been specifically tuned for different column and index types [42], and they go far beyond sequential block-based prefetching.
- **Load Engine**: Being an OLAP system, it is imperative that loading data into SAP IQ is fast and efficient. Consequently, three decades of engineering work has been put into parallelizing SAP IQ’s load engine so that it can maximize CPU utilization during load.
- **Elasticity**: SAP IQ employs a distributed computing model over *shared* storage. In this model, compute nodes can be easily and independently added to the system to scale-out,

without changing the underlying storage system or the way data are partitioned.

When developing the cloud version of SAP IQ, we have decided to exploit the aforementioned strengths of the product as much as possible and avoid reinventing the wheel. Consequently, we have focused on identifying the changes needed to transition the product into a strong cloud-native engine. During this process, we have encountered some challenges: traditionally, SAP IQ has been designed to operate on shared storage devices with strong consistency guarantees (e.g., high-caliber storage area network devices). In other words, if a transaction writes data to a block on disk and then commits, it is expected that when subsequent transactions read that block, they retreive the latest version of the data that were written to that block.[1] Therefore, deploying SAP IQ on the cloud, as is, would have meant utilizing storage solutions such as NetApp [16] or AWS EFS [4] that provide a POSIX-compliant file interface and strong consistency guarantees, but at a much higher monetary cost. These costs can accumulate easily to diminish the economies of scale that one would expect on the cloud, which can be undesirable.

Instead, we have enhanced the design of SAP IQ to operate on cloud object stores such as AWS S3 [5] and Azure Blob Storage [11] (Figure 1). Object stores rely on a weaker consistency model, and may have higher latency; however, because of these design trade-offs, they are able to offer (i) better pricing, (ii) enhanced durability, (iii) improved elasticity, and (iv) higher throughput. By enhancing SAP IQ to operate under these design trade-offs, we have unlocked many of the opportunities offered by object stores. In a nutshell, we extend SAP IQ's buffer manager and transaction manager, and introduce a new caching layer that utilizes instance storage on AWS EC2 [3]. More specifically:

(1) We rely on the fact that SAP IQ makes a clear distinction between the logical and the physical representation of pages in the system to directly map logical pages to objects in object stores (Section 3);
(2) We modify our buffer manager, and data structures therein, to enforce a "never write an object twice" policy to handle the weaker consistency model used in object stores (Section 3.1);
(3) We introduce techniques for efficiently allocating object keys in a multi-node setting, as well as for persisting and recovering these keys (Section 3.2);
(4) We extend our transaction manager with mechanisms to keep track of pages stored on object stores that are no longer needed (i.e., in the context of multi-version concurrency control) and properly clean these pages up (Section 3.3);
(5) As a performance optimization, we introduce a second layer of cache, namely the Object Cache Manager (OCM), that acts as a read/write cache between the existing buffer manager and the object stores (Section 4); and
(6) We introduce a new feature wherein users are able to take frequent, near-instantaneous snapshots of their databases (Section 5).

Experiments using the TPC-H benchmark demonstrate that we can gain an order of magnitude reduction in data-at-rest storage costs while improving query and load performance.

[1]The same argument holds for transactions taking place on different nodes.

2 BACKGROUND

SAP IQ is a disk-based, columnar, fully relational database management system designed for complex online analytical processing (OLAP) workloads [18] (Figure 1). Users are able to instantiate a cluster of distributed servers, called a *multiplex*, to concurrently perform loads and queries in a scale-out fashion. In the multiplex configuration, there are three types of nodes: *coordinator*, *writer* and *reader*. The latter two are collectively called the *secondary* nodes. While writer nodes and the coordinator node are able to perform modifications on the database, reader nodes cannot. In a typical workload, the writer nodes are used for DML operations, and the coordinator node is reserved mostly for DDL operations. In multiplex, SAP IQ implements multi-version concurrency control (MVCC) with table-level versioning and snapshot isolation [25].

SAP IQ organizes physical storage using the concept of a *dbspace*. A dbspace is a collection of operating system files or raw devices. There are two types of dbspaces: *system* and *user* created dbspaces. The system dbspace is further divided into two parts: *temporary* and *main*. The temporary system dbspace is used to store the hash tables, output of sort runs, etc. produced by the query engine. Furthermore, the temporary system dbspace is used for sharing intermediate query results between the nodes in a multiplex. The main system dbspace stores important data structures such as the checkpoint blocks and the freelist. The freelist is a bitmap that keeps track of the allocated blocks across the dbspaces in a database: a bit set in the freelist indicates that the block is in use, whereas a clear bit indicates that the block is available for use.

The storage unit in SAP IQ is a *page*, and the system makes a clear distinction between the logical (in-memory) and the physical (on-disk) representation of a page. How a page is physically stored is transparent to the query engine and the layers above. The buffer manager responds to requests from the query engine in the form of ⟨logical-page-number, version-counter⟩ and is responsible for locating the correct version of a page from disk (or from the buffer cache). The buffer manager relies on a data structure called the *blockmap* to maintain the mappings between logical pages and a sequence of blocks on disk.[2] When a page is written out, the blockmap data structure is updated to reflect the page's physical location on disk and potentially the new compressed size of the page (in blocks). When a page is read from disk, it is cached in decompressed form in the buffer cache. The separation between the logical and physical representation of a page makes the implementation of MVCC simpler: dirty pages that are marked to be versioned are simply copied to a new physical location upon eviction from the buffer cache.

3 CLOUD-NATIVE STORAGE

With the cloud-native version of SAP IQ, we offer our users the ability to create *dbspaces* directly over object stores such as AWS S3 and Azure Blob Store. We refer to dbspaces stored on object stores as *cloud dbspaces*. To create a cloud dbspace in SAP IQ, the following statement can be used:

```sql
CREATE DBSPACE dbspaceName
USING OBJECT STORE "s3://bucketName"
```

[2]In SAP IQ, a page is stored physically as a contiguous set of blocks and can occupy anywhere between 1–16 blocks.

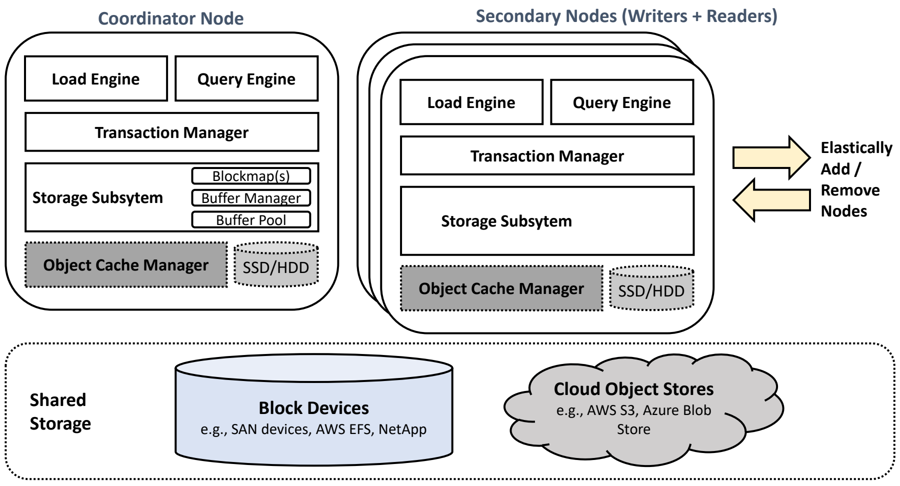

Figure 1: Architectural Overview of SAP IQ

Just like the on-premise version of SAP IQ, a database instance consists of one or more dbspaces. On the cloud, users may create dbspaces on different hyperscalers. For example, a user may create three dbspaces, one constructed over traditional storage volumes, the second over AWS S3 and the third over Azure Blob Store. This feature gives freedom to users; that is, users have the ability to choose a storage provider based on price and performance characteristics, as well as move data between different storage providers as needed.

The main challenge in supporting cloud dbspaces in SAP IQ has been the consistency model used in object stores such as AWS S3. More specifically, AWS S3 provides *eventual* but not strong *consistency* guarantees. In other words, if an S3 object is updated more than once, clients will eventually read the most up-to-date version of the object; however, in the meantime, they may read stale versions. Consider the following three scenarios:

(1) The object is read successfully and contains the latest version of the data,
(2) The object is read successfully but contains stale data, and
(3) The read operation fails with an error indicating that the object does not exist (even though it actually does).

Traditionally, SAP IQ has been designed to operate on shared storage devices with strong consistency guarantees; therefore, it can only handle the first scenario. To overcome this limitation, we have enhanced the design of SAP IQ as follows.

- **Directly Store Database Pages as Objects**: Instead of storing database pages as blocks in files, we directly store them as objects/blobs in object stores. This design eliminates the need for maintaining heavy-weight data structures such as the freelist. Specifically, whenever we flush a dirty page from a cloud dbspace, instead of going to the freelist to locate an available range of blocks, we simply obtain a new object key.
- **Never Write an Object Twice**: Every time a dirty page in a cloud dbspace is flushed from the cache, it is uploaded to the object store using a newly generated key. This scheme ensures that a page that is stored on an object store has exactly one version. Consequently, when SAP IQ's storage subsystem tries to read the page; it will either get the one and only version of that page or get an "object not found" error due to eventual consistency (thus, eliminating the second scenario). This use case is known as the *read-after-write* consistency. In case of an error, we have modified the storage subsystem to retry until the object is found, up to a configurable number of retries.
- **Generate Unique Object Keys**: We have introduced a new component called the *Object Key Generator* that is responsible for generating unique object keys. This component ensures that object keys are never reused and that appropriate metadata are persisted and restored across database restarts. Key generation is done through the coordinator node; therefore, if any of the secondary nodes requests a new key, it issues an RPC call into the coordinator. The coordinator is responsible for maintaining a list of keys that have been handed out to the secondary nodes. As transactions that originate on secondary nodes commit or rollback, the coordinator is notified so that the list can be updated. The coordinator relies on these metadata to correctly garbage collect pages that are no longer needed.

In the remainder of this section, we discuss the details of our approach. In particular, in Section 3.1, we present the data structures and components we have modified to enforce the *read-after-write* consistency in SAP IQ. In Section 3.2, we describe how unique object keys are generated in an efficient and scalable manner and how these keys are transactionally persisted. In Section 3.3, we describe our garbage collection gear, with examples highlighting various recovery scenarios such as transactions rolling back and transactions aborting due to server crashes or restarts.

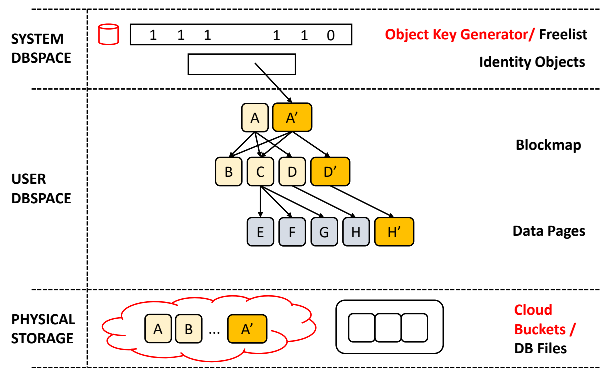

Figure 2: Lifecycle of Pages

3.1 Management of Pages in the Buffer Pool

Traditionally, database pages have been stored on shared block devices in SAP IQ. Consequently, data structures have been developed (i) to keep track of the free blocks on the storage system (freelist), and (ii) to maintain mappings between logical database pages and physical block numbers (blockmap). In the cloud version of SAP IQ, the freelist has a reduced role because the notion of "free blocks" does not apply to cloud dbspaces. In contrast, the blockmap object has an extended role because it is used to not only maintain mappings between logical database pages and physical block numbers (for conventional dbspaces) but also between logical database pages and object store keys (for cloud dbspaces).

In SAP IQ, new pages get created in-memory first; that is, the lifetime of a page starts in the buffer cache. When a page is modified, it is marked as dirty. The buffer manager maintains a list of all the dirty pages associated with active transactions. Before a transaction commits, all associated dirty pages are flushed to permanent storage, which could be a bucket in an object store or a dbfile on a conventional shared block device. This is necessary in an OLAP system such as SAP IQ, as the transaction log does not store the data that are updated (which can be very large in volume); instead, it stores the metadata. Therefore, for durability, data pages need to be flushed to permanent storage before a transaction commits. A dirty page can be flushed from the cache earlier as well (upon eviction), when the buffer manager needs to make room for a more recent page.

Whenever a dirty page that belongs to a cloud dbspace is flushed from the buffer cache, it is stored using a new object key in the object store. This unique key must be recorded in a metadata structure so that the correct version of the page can be obtained on future lookups. For a data page, the key is recorded in the blockmap object that owns the data page. The key of a blockmap page is recorded in its parent blockmap page (blockmap pages are organized as a tree). In case of a root blockmap page, the key is recorded in an identity object that is stored as part of the system catalog.

Consider the example in Figure 2, where A, B, C and D denote a tree of blockmap pages, and E, F, G and H denote some of the data pages that are maintained by this blockmap. Let us assume that we have dirtied page H. Now, due to cache pressure, H is being evicted from the cache and flushed to the underlying object store. Instead of updating the page in place, we create a new version of the page, namely H′. The new key needs to be recorded in the blockmap page that manages H, which is page D. This operation causes page D to be dirtied as well. Eventually, when we flush D, we will need to obtain a new key for D and version it as D′. This will force us to update the link in the parent blockmap page. Subsequently, A will be versioned as A′, and since A′ is a root blockmap page, the new key for A′ will be recorded in the identity object. The identity object is part of the *system* dbspace, which is always stored on devices with strong consistency guarantees; therefore, it can be updated in-place. During this process, pages A, D and H will be marked so that when the transaction commits, they can be garbage collected from the underlying object stores.

As part of MVCC, SAP IQ already supports copy-on-write semantics for data and blockmap pages when tables are versioned, and keeps multiple versions of pages and identity objects around. In that respect, the new model relies heavily on existing gear to enforce read-after-write consistency on object stores. However, there is a subtle difference between the two models: in the existing model, a page can be updated in-place (on disk) as long as it is modified as part of the same transaction/savepoint; whereas in the new model, every time a page is written out to an object store; it must be versioned.

Instead of extending the blockmap and the identity objects with yet another field, we have decided to overload the existing physical block number fields to store these keys. The physical block number field is a 64-bit integer. The maximum physical block number that is currently supported by IQ is $2^{48} - 1$; therefore, we have reserved the higher range $[2^{63}, 2^{64})$ for object keys. Assuming that we consume 10,000 object keys per second per multiplex node, using a cluster of 20 nodes, it would take more than 1.4 million years to consume the reserved range of object keys in a database instance.

The 64-bit key that is stored internally by SAP IQ is slightly different than the full key used on the object store. The reason is that AWS places a restriction on the number of requests (per second) a bucket can process for a "prefix" [12]. To overcome these limitations, we try to generate as many randomized prefixes as possible by prepending the 64-bit key with a prefix that is constructed by applying a computationally efficient hash function to the 64-bit value (e.g., Mersenne Twister [41]).

3.2 Object Key Generation

There are three requirements that the Object Key Generator needs to satisfy with respect to key generation:

(1) **64-bit Keys**: To store object keys in the existing blockmap data structure with minimal (file format) changes, we require object keys to be 64-bit integers.
(2) **Uniqueness**: Due to eventual consistency issues, we do not allow a page to be overwritten (i.e., written more than once) in object stores. Consequently, we rely on the Object Key Generator to hand out keys that are unique across the nodes in a multiplex cluster.

(3) **Monotonicity**: We expect the generated keys to be (strictly) monotonically increasing. This property allows us to use key-ranges as opposed to singleton keys as a space and performance optimization during creation and garbage collection of pages.

The coordinator node is responsible for generating object keys. Requests to generate object keys can come from either secondary nodes or the coordinator node itself. For efficiency, object keys are allocated in ranges in a monotonically increasing fashion, and are cached locally on each node.

When a secondary node requests a new object key, it first queries the locally cached range of keys for availability. If the cached range of keys is already exhausted, the secondary node makes an RPC call into the coordinator to request a new range, caches the response and consumes the next available key from the range. The number of keys requested (i.e., range size) starts at a default value; nevertheless, it can dynamically increase or decrease on subsequent RPC calls based on the load on the secondary node.

The RPC call to allocate a key range initiates a new transaction on the coordinator. As part of this transaction, two bookkeeping events take place: (i) the largest allocated object key is recorded in the transaction log; and (ii) a data structure that maintains the ranges that have been handed out to the secondary nodes (as well as to the coordinator) is updated and flushed to disk. Upon success, the transaction commits and the newly allocated key range is returned to the secondary node. These bookkeeping events serve to ensure that after crash recovery, the coordinator node continues to hand out ranges in a strictly monotonically increasing fashion, and that proper garbage collection takes place.

When the request for a new key range is initiated on the coordinator, the process is similar but with one optimization: the coordinator node does not need to make an RPC call on self; it can do the allocations directly. The operation is still embodied within a transaction. Therefore, if the coordinator node crashes, it is able to recover the maximum object key that was allocated across the multiplex cluster by recovering the maximum object key that was stored in the transaction log since last checkpoint.

3.3 Page Lifecycle Management

SAP IQ uses MVCC with snapshot isolation; therefore, when transactions modify data, new versions of tables are created. Older versions of a table continue to exist for as long as there are transactions still referencing those versions. The transaction manager is responsible for determining that an older version of a table is no longer referenced, and subsequently deleting the physical pages associated with that version.

In the cloud version of SAP IQ, we extend the transaction manager to keep track of pages stored on object stores to appropriately delete them when those pages are no longer needed (i.e., garbage collection). We distinguish between the two types of garbage collection events: (i) garbage collection of pages used by committed or rolled back transactions, and (ii) garbage collection in the presence of coordinator or writer node crashes (or force kills).

In the absence of crashes, the transaction manager relies on existing two data structures, namely, the *roll-forward*/*roll-back* (RF/RB) bitmaps, to determine which pages to delete from the underlying object stores. Each transaction has its own pair of RF/RB bitmaps: the RF bitmap records the pages that have been marked for deletion by the transaction whereas the RB bitmap records the pages that have been allocated. In the on-premise version of SAP IQ, these bitmaps record the range of blocks that a page occupies on shared block storage as a sequence of 1-bits in the bitmap. For a cloud page, we utilize the same data structures to record the object key, which is an integer in the range $[2^{63}, 2^{64})$, as a single bit in the bitmap. We distinguish between the two types of representations (i.e., sequence of physical block numbers vs. object keys) by simply looking at the range in which a bit is recorded.

When a transaction rolls back, pages that are recorded in its RB bitmap can be deleted immediately. When a transaction commits, however, the pages that have been marked for deletion by the transaction cannot be deleted right away due to the fact that other transactions may still be accessing those pages in an MVCC system with snapshot isolation. Instead, when a transaction commits, its RF/RB bitmaps are flushed to storage, the identities of the bitmaps are recorded in the transaction log, and the responsibility of garbage collection is passed onto the transaction manager.

The transaction manager maintains a chain of committed transactions with pointers to their RF/RB bitmaps, as well keeps track of the oldest transaction in the chain whose pages are still referenced by active transactions in the multiplex. When the oldest transaction in the chain is no longer referenced, its RF/RB bitmaps are used to compute the pages that can be deleted, including the pages on object stores, and the transaction is dropped from the chain.

Garbage collection in the presence of coordinator and writer node crashes involves more sophisticated steps. The reason is threefold:

(1) If the coordinator node crashes, the metadata structures maintained by the Object Key Generator, namely, the maximum object key and the active sets of object keys handed out to secondary nodes, must be recovered to a consistent state. In Section 3.2, we have already outlined how the coordinator node recovers the maximum object key in case of a crash.
(2) If a node goes down, an active transaction may get aborted before its RF/RB bitmaps are persisted. In that case, the allocations performed by the aborted transaction must be undone without the RB bitmap.
(3) If an active set of object keys that are handed out by the coordinator are not fully consumed when the crash happens, objects corresponding to unconsumed keys must be garbage collected.

To recover the active sets of object keys that have been handed out to secondary nodes, we rely on the RF/RB bitmaps. Note that the RF/RB bitmaps are already part of the crash recovery gear in the on-premise version of SAP IQ. That is, crash recovery starts from the last checkpoint, where a copy of the freelist has been persisted, and applies the RF/RB bitmaps of all committed transactions to the freelist in order. Pages in the RF bitmap are removed from the freelist (i.e., deallocations are applied to the freelist), and the pages in the RB bitmap are marked as "in-use" in the freelist. For pages on object stores, we do not have a freelist to maintain. Instead, the crash recovery gear uses the RF/RB bitmaps to recover the active

Clock	Event	Description	Active Set(s)
50	Checkpoint	Metadata including the active sets of keys are flushed to disk	∅
60	W_1 allocation	Key range 101–200 is allocated to W_1	W_1: {101–200}
70	T_1 begins on W_1	Objects with keys 101–130 are flushed; range is recorded in the RB bitmap of T_1	W_1: {101–200}
80	T_2 begins on W_1	Objects with keys 131–150 are used by T_2; range is recorded in the RB bitmap of T_2	W_1: {101–200}
90	T_1 commits	RF/RB bitmaps of T_1 are flushed to disk; active set is updated	W_1: {131–200}
100	T_3 begins on W_1	Objects with keys 151–160 are flushed; range is recorded in the RB bitmap of T_3	W_1: {131–200}
110	Coordinator crashes		∅
120	Coordinator recovers	Active set is recovered	W_1: {131–200}
130	T_2 rolls back	Objects with keys 131–150 are garbage collected; active set is *not* updated	W_1: {131–200}
140	W_1 crashes		W_1: {131–200}
150	W_1 restarts	Outstanding allocations for W_1 are garbage collected on the coordinator	∅

Table 1: Example of recovery and garbage collection. For sake of presentation, keys are not in the realistic $[2^{63}, 2^{64})$ range.

sets of object keys that have been handed out to secondary nodes. We explain the recovery mechanism through an example.

Consider the sequence of events in Table 1 in a multiplex cluster with a coordinator and a writer node (W_1). The example involves three transactions T_1–T_3 and two crash points. First, consider the crash on the coordinator (110). During crash recovery, the following steps are taken:

(1) The transaction log is replayed from the checkpointed state (50).
(2) At the time of checkpoint, the active set was empty; therefore, replay starts with an empty set. Please note that, in the general case, the set could contain elements from prior allocation events.
(3) When the allocation event is replayed (60), the active set is reconstructed as W_1: {101–200}.
(4) When the commit of T_1 is replayed, the active set is updated to W_1: {131–200} because the committed range {101–130} no longer needs to be tracked.
(5) There is no need for garbage collection (of cloud objects) because there were no active transactions on the coordinator that got aborted.

When a writer node crashes, if there were any active transactions on the node at the time of the crash, then the allocations performed by those transactions must be garbage collected since these transactions can never commit. Furthermore, any outstanding allocations on the node must also be garbage collected. Consider the crash on the writer node (140). When the writer node restarts (150), the following steps are taken:

(1) W_1 makes an RPC call into the coordinator to initiate the garbage collection process.
(2) When the coordinator receives the RPC call, it locates the active set for W_1.
(3) Every page in the range W_1: {131–200} is polled for garbage collection. If a page in the set exists, it is deleted from the underlying object store.

Note that some of the pages in the range {151–160} may not have been flushed to the underlying object stores, which is fine. Every page in the range will be polled by the coordinator as a potential candidate for garbage collection. Furthermore, note that pages in the range {131–150} have already been garbage collected when T_2 rolled back, but this information was deliberately not communicated to the coordinator. Consequently, when W_1 was restarted, the same range was polled for garbage collection. This is a conscious optimization to reduce the amount of inter-node communication for transactions rolling back, which is expected to be more frequent than node restarts.

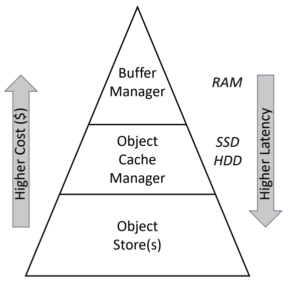

Figure 3: Memory Hierarchy

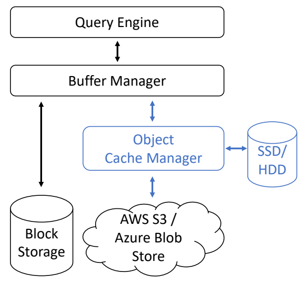

Figure 4: OCM Architecture

4 OBJECT CACHE MANAGER

By storing user data directly on object stores such as AWS S3 and Azure Blob Store, the cloud-native version of SAP IQ aims to significantly reduce storage cost while benefiting from the elasticity and scale-out properties of object stores. On the downside, the I/O characteristics of object stores are different than those of conventional storage solutions such as hard disk or solid state drives. In particular, while it is possible to achieve a much better (i.e., higher) throughput using object stores, individual read and write operations may incur a worse (i.e., higher) latency. These characteristics can have a negative impact on query performance. Potentially, one could overcome these limitations by introducing parallelism, for example, in the form of more aggressive prefetching during reads. However, these kind of optimizations put extra burden on the buffer manager, which is undesirable because the buffer manager uses RAM, and RAM is expensive on the cloud.

To counteract the impact of high latency *without* utilizing more RAM, we introduce the Object Cache Manager (OCM). The OCM is a disk-based extension to SAP IQ's buffer manager. Just like the buffer manager, in a multiplex environment, each server has its own OCM, and the objects cached by one OCM instance are not shared by another. Underneath, the OCM uses a fast solid state drive (SSD) or a hard disk drive (HDD) that is locally attached to the compute instance. Latency on the locally-attached SSD or HDD is significantly lower than object stores, and pricing is more affordable than RAM. Consequently, we are able to achieve better query performance with little extra cost (Figure 3).

The OCM is a read and write cache (Figure 4). During a read, a page is first looked up in SAP IQ's conventional buffer manager, which resides on RAM. If the page is not found, then the page is looked up in the OCM. If the page is already cached in the OCM, it is read from the locally attached storage and returned to the caller. If the page is not found, then the OCM reads the page from the underlying object store, returns it to the caller and *asynchronously* caches the new page in the OCM's disk storage for future lookups. The page is cached also in RAM, in SAP IQ's conventional buffer manager. This *read-through* semantics significantly reduces the read latency for pages that are cached in the OCM.

The OCM supports write operations in two modes: write-back and write-through. In the write-back mode, the page is *synchronously* written to the locally attached storage of the OCM while being *asynchronously* written to the underlying object store. The latency of the write operation in the write-back mode is determined by the latency of the locally attached storage (i.e., writes are fast). In the write-through mode, the page is *synchronously* written to the object store while being cached *asynchronously* in the locally attached storage. In the latter mode, the latency of the write operation is determined by the latency of the underlying object store (i.e., writes are slower).

As described above, pages are cached in the OCM during both read and write operations. The OCM relies on the least-recently used (LRU) eviction policy to free up space for new pages [45], which is aligned with SAP IQ's buffer manager. The OCM maintains a single LRU list between reads and writes with the assumption that pages that have been written out or read more recently by the OCM are more likely to be read again. By design, a page that has been read-through the OCM cannot be written out again with the same object key; therefore, caching pages in the OCM primarily benefits read operations. Furthermore, during a write-back operation, a page is not added to the LRU list until it has been successfully written to the underlying object store. This optimization prevents unnecessary build-up of pages in the OCM cache (e.g., pages of failed/rolled-back transactions).

The reason why the OCM supports two modes of writes is as follows: transactions interact with SAP IQ's buffer manager in three phases, namely, (i) warm-up, (ii) churn, and (iii) commit. In the warm-up phase, pages start to fill up the buffer manager's cache in RAM. In the churn phase, the least recently used pages are evicted from the cache to make room for more recent pages. In the commit phase, the buffer manager flushes out pages that have been dirtied by the committing transaction, including the ones in the OCM. SAP IQ is primarily an OLAP system, where long running transactions are normal and expected. Consequently, the churn phase constitutes the longest period during a transaction, and it must be optimized. For this reason, pages that are evicted due to cache pressure during the churn phase, are written out using the *write-back* mode of the OCM, which helps keep the latency low. On the other hand, during the commit phase, the OCM must ensure that the dirty pages are flushed out to the underlying object store; therefore, it relies on the *write-through* mode, which prioritizes writes to the object store over the caching done in the locally attached storage.

In the presence of multiple transactions, the OCM prioritizes the write operations of committing transactions. A transaction indicates its desire to start the commit phase, by sending a *FlushForCommit* signal to the OCM. Upon receiving this signal, the OCM moves the dirty pages belonging to the committing transaction to the head of a write queue; thus, prioritizing all previously started background jobs for that transaction. Furthermore, it switches the mode of write from *write-back* to *write-through*. This switch in mode ensures that all subsequent write requests that come from the committing transaction are executed directly on the underlying object store as opposed to being treated as background jobs.

The OCM is intended solely as a performance optimization; therefore, its presence or lack thereof, does not affect transactional consistency. In the absence of the OCM, write operations are issued directly to the object stores. A failed write is retried; but after a pre-determined number of failures of the same page, the transaction is rolled back. Later on, the transaction manager garbage collects pages that belong to rolled back transactions. In the presence of the OCM, write operations take two forms: (i) writes to the locally attached storage for caching; and (ii) writes to the underlying object store. If a write to the locally attached storage fails, the error is ignored, and the page is written directly to the object store. If a write to an object store fails, however, the transaction is rolled back after a pre-determined number of retries. These checks guarantee that all dirty pages of committed transactions have been successfully stored on the object stores.

When encryption is enabled, the buffer manager of SAP IQ hands over pages to the OCM in encrypted form; and the pages are decrypted upon being read from the OCM. Consequently, neither the pages that are cached in the locally attached storage nor the ones that are persisted on the object stores, can unintentionally expose user data.

5 SNAPSHOTS

On the cloud, the ability to store user data on object stores has prompted us to revisit our backup strategy. In particular, the on-premise version of SAP IQ supports numerous backup features (e.g., full, incremental, incremental since full, virtual and decoupled) that are built into the product to provide fault-tolerance in the case of software and hardware failures. In contrast, most object stores already provide fault-tolerance in the form of replication, thus, relieving the burden from SAP IQ. Consequently, while continuing to support conventional backups, on the cloud, we enhance our product by supporting (i) frequent and near-instantaneous *snapshots*, and (ii) the ability to go back in time to a consistent snapshot using *point-in-time restore*. We consider the ability to create *read-only views* over past snapshots as future work.

To support *frequent* and *near-instantaneous* snapshots on the cloud, we capitalize on the fact that storing data on object stores is affordable; hence, we can defer the deletion of pages from object stores for a user-defined retention period. To implement this

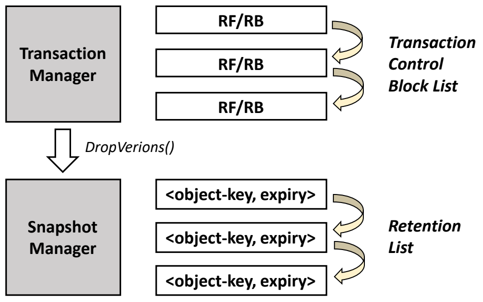

Figure 5: Snapshot Manager

change, we extend SAP IQ's *transaction manager* and introduce a new component called the *snapshot manager*. Recall that the transaction manager is responsible for determining when an older version of a table is no longer referenced, and subsequently deleting the physical pages associated with that version. When a version of a page is dropped from the transaction manager, instead of deleting the page from the underlying object store, we retain the page and transfer its ownership to the *snapshot manager* (Figure 5). The snapshot manager is then responsible for permanently deleting the page in a background process when the retention period for that page expires. Since different pages fall into the ownership of the snapshot manager at different points in time, the snapshot manager maintains a FIFO list containing records of the form ⟨object-key, expiry⟩ to garbage collect pages when they are no longer needed. Just like the user data, this list of metadata is also stored on object stores. As pages are permanently deleted, the list is pruned.

In this new model where pages are retained on object stores longer, taking a snapshot involves the following simplified set of operations:

- Backing-up the metadata utilized by the snapshot manager; and
- Taking a full backup of the system catalog and all non-cloud dbspaces including the system dbspace. We do not backup cloud dbspaces.

Data backed up during a snapshot operation are automatically deleted by the snapshot manager when the snapshot expires (i.e., when the retention period for that snapshot ends).

On a database where all user dbspaces are on the cloud, taking a snapshot can be near-instantaneous. The reason is that the only dbspace that needs to be backed up in full is the system dbspace, which has become significantly smaller due to the reduced role of the freelist. Consequently, we can afford to take snapshots frequently–be it automatic or user-driven.

To restore a database from a snapshot that was taken within the retention period, it is sufficient to restore the metadata of the snapshot manager, the catalog and the system or any non-cloud dbspace files (if applicable). All the blockmap and data pages in cloud dbspaces that are referenced by the identity objects in the catalog (Figure 2) would already have been retained in the underlying object stores. After the restore, the database will no longer need the pages that have been created between the snapshot and the restore operations. Since the Object Key Generator hands out keys in a monotonically increasing fashion, the range of keys to be garbage collected can be computed from the keys used during the snapshot and the restore operations (the former is recorded as part of the metadata that is stored during the snapshot).

6 EVALUATION

In this section, we demonstrate that by adapting SAP IQ to the weaker consistency model employed by object stores, we can achieve significant savings in monetary costs while at the same time, improving query/load performance and scalability on the cloud. To evaluate our approach, we have designed four sets of experiments on an *internal, development* build of our software:

(1) We compare the monetary cost and the query/load performance for when user dbspaces are stored on object stores versus when they are stored on block storage volumes on the cloud;
(2) We evaluate the benefits of the OCM, in particular, how it masks the high latency incurred on object stores;
(3) We analyze the *scale-up* behavior of our approach in a single-node setup by increasing the compute power assigned to that node; and
(4) We evaluate the *scale-out* characteristics of our approach in a multiplex setup by increasing the number of compute nodes.

In our experiments, we use the TPC-H benchmark [20]. The TPC-H tables are created as range-partitionied, and High-Group (HG) indexes [21] are created on the following columns: *o_custkey*, *n_regionkey*, *s_nationkey*, *c_nationkey*, *ps_suppkey*, *ps_partkey* and *l_orderkey*. For loads, all input files are stored in an S3 bucket.[3] Unless otherwise stated, we rely on AWS EC2 [3] instances for compute and AWS S3 [5] for the underlying storage for cloud dbspaces; and system dbspaces (main and temporary) are stored on a *gp2* volume on AWS EBS [2]. Each SAP IQ server is assigned a dedicated EC2 instance, where ¾ of the RAM is reserved for SAP IQ's buffer manager and large memory allocations, and the rest is shared dynamically between the operating system and the heap allocations originating from SAP IQ. The OCM is configured to utilize the maximum SSD available on the EC2 instance, where multiple SSD devices are bundled together using a RAID 0 volume.

In the first experiment, we use a single compute instance (m5ad.-24xlarge) to load the TPC-H dataset at scale factor 1000 into a cloud dbspace that is stored on AWS S3 (i.e., object store). After the data are loaded, we run the 22 benchmark queries sequentially (i.e., power mode). Subsequently, we repeat the experiment with dbspaces on AWS EBS and AWS EFS [4] (i.e., block storage volumes on the cloud). For EBS, we use a 1TB *gp2* volume, and for EFS, we use a standard volume that is charged based on actual size utilization.

Table 2 depicts the load and query execution times for all three runs; Table 3 shows the cost of loading the dataset and then the cost of sequentially running the set of benchmark queries once; and Table 4 shows the monthly cost of storing the data in the respective volumes.[4] The cost of loading and querying data includes the cost of

[3]During loads, the network bandwidth will be shared between the reads from the input files in the S3 bucket and dbspace I/Os.
[4]Costs are calculated based on the publicly available prices listed by Amazon.

Storage Volume	Load	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}	Q_{11}
AWS S3	2,657.2	163.0	6.5	128.7	59.1	78.3	6.9	0.6	185.2	73.1	25.1	4.3
AWS EBS	4,294.1	185.5	0.3	81.3	94.8	282.6	19.5	5.0	482.6	476.5	154.8	27.0
AWS EFS	12,677.2	457.4	1.1	172.3	231.5	662.0	44.6	13.1	1,125.2	1,164.2	380.4	67.8
Storage Volume		Q_{12}	Q_{13}	Q_{14}	Q_{15}	Q_{16}	Q_{17}	Q_{18}	Q_{19}	Q_{20}	Q_{21}	Q_{22}
AWS S3		3.3	81.4	7.3	16.6	32.6	26.0	122.7	0.4	16.3	385.7	32.6
AWS EBS		12.1	175.7	41.9	45.2	73.6	178.1	341.4	0.1	80.9	543.9	56.0
AWS EFS		27.9	422.0	101.6	103.2	154.7	430.3	818.6	0.2	194.9	713.0	103.4

Table 2: Load and query execution times (in seconds) using the TPC-H benchmark at scale factor 1000

Volume	Load Cost (USD)	Query Cost (USD)
AWS S3	15.18	2.35
AWS EBS	5.04	3.88
AWS EFS	15.39	8.53

Table 3: Compute cost of loading the TPC-H data at scale factor 1000 and the compute cost of sequentially running the set of benchmark queries once

	Objects	Percentage
Cache Misses	962,573	25.5%
Cache Hits	2,807,368	74.5%
Evictions	962,589	

Table 5: Utilization of the OCM during the execution of the TPC-H benchmark queries

Volume	Monthly Storage Cost (USD)
AWS S3	12.05
AWS EBS	51.80
AWS EFS	155.40

Table 4: Monthly cost of storing the data(-at-rest) in the respective volumes

keeping the EC2 instances up and running during the corresponding time period, the cost associated with the EBS volumes that are created for system dbspaces (main and temporary), and the extra charges due to S3 requests (e.g., PUT, GET). The monthly data-at-rest costs are computed based on the size of the compressed data stored in the user dbspaces multiplied by the monthly rates provided by Amazon.

Based on these results, we make multiple observations. First, storing and querying data are cheaper when the dbspaces are stored on S3. In general, the cost of storing data on S3 is much cheaper than EBS or EFS; therefore, the former part of the observation is trivial. Querying data from S3 incurs an additional cost associated with every S3 GET request, but this was amortized as we were able to execute the workload faster on S3 (explained below). For loads, our solution on S3 incurs a higher cost than EBS (due to S3 PUT requests) but it is still cheaper than EFS. Realistically speaking, EBS can only be used as a storage volume in a non-elastic setup because even though it allows multiple instances in the same availability zone to be attached to the same volume, users need to pay a higher premium to get that service, and IOPS is capped based on the provisioning of the volume and not the number of nodes in the system. Furthermore, even in the case of a single-node setup, EBS cannot offer durability across multiple availability zones but that comes with S3 and EFS.

Our second observation is that SAP IQ runs much faster on S3 when loading and querying data. This is somewhat a non-trivial finding because latency on S3 can be significantly higher than the latency on EBS and EFS. Nevertheless, if utilized properly, S3 scales well, and it provides a better throughput than EBS and EFS as IOPS can be significantly throttled on the latter two.[5] To get the best out of S3, a system needs to make use of (i) prefixing, (ii) parallelism and (iii) local caching, and SAP IQ does all three: it prepends object keys with hashed prefixes before storing them on S3, it relies aggressively on parallel I/O and prefetching, and it implements local caching in the form of the OCM. With these optimizations, SAP IQ is able to achieve better throughput on S3, which translates to faster load and query execution times. These two observations validate the thesis of the paper. That is, by enhancing SAP IQ to operate under the weaker consistency model, we have unlocked many of the opportunities offered by object stores (e.g., better pricing, better throughput, etc.).

While running the 22 benchmark queries is much faster on S3 (geometric mean of *23.2* seconds compared to *52.1* on EBS and *119.3* on EFS), there are exceptions: Q_2 and Q_{19} are short running queries where parallelism and prefetching have not been sufficient to mask the latency of S3, which highlights an area of improvement. Q_3 is a long running query; therefore, one would have expected this query to be the fastest on S3, but instead, it was fastest on EBS. Upon further inspection, we note that the OCM was to blame in this case. If OCM is disabled, the query takes *58.0* seconds to be executed on S3, which is significanly faster than the execution on EBS and EFS. We analyze this behavior in more detail in the upcoming experiment.

In the second experiment, we evaluate the effectiveness of the OCM. In this experiment, we rely on two configurations: to stress the OCM, we use the m5ad.4xlarge instance that has lower RAM capacity (64GB), and to monitor the general behavior, we use the m5ad.24xlarge instance. In the first part of the experiment, we disable the OCM and execute the TPC-H queries sequentially over a cloud dbspace stored on S3 at scale factor 1000. In the second part, we enable the OCM on the NVMe SSD's that are attached to the instance and repeat the experiment.

[5]On EBS, the maximum IOPS that is supported is a function of the volume type, and on standard EFS volumes, the IOPS is a function of the space that is utilized.

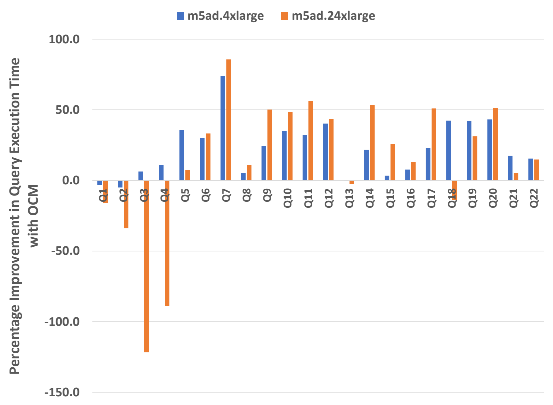

Figure 6: Impact of the OCM on Query Execution Times

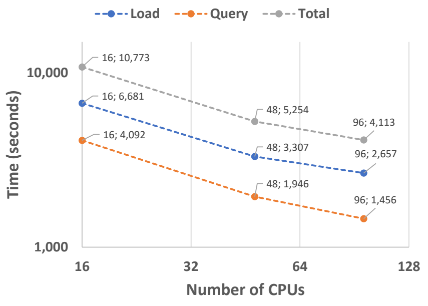

Figure 7: Scale-up Behavior

Figure 6 compares the query execution times between the runs with and without OCM, and Table 5 depicts the cache hit rates. Overall, we observe 25.8% and 25.6% improvement in query execution times (geometric mean) due to the OCM during the m5ad.4xlarge and m5ad.24xlarge runs, respectively. During both runs, we observe a warm-up behavior where the disk cache of the OCM gets filled up with objects that are *read-through* the OCM, which results in worse performance for the first few queries but gradual improvement in performance for the remaining.

In the m5ad.24xlarge run, we see a significant degradation in performance for Q_3 and Q_4 (recall our earlier observation). Upon further inspection, we note the following:

- During the execution of these queries, the OCM cache is relatively cold; therefore, most of the objects are being *read-through* the OCM, which means that we go to S3 to read the objects while writing them *asynchronously* to the OCM disk cache;
- We verify that the overhead associated with scheduling the asynchronous write operation is less than 5%, which alone does not explain the slowdown; and
- We observe that there are cache hits in the OCM during the execution of these queries but surprisingly, the latency of reads is significantly higher on the SSD devices than on S3.

Based on these observations, we conclude that under heavy load, where the OCM saturates the underlying SSD devices with a significant volume of (asynchronous) writes, reads for cache hits might suffer, thus, resulting in overall degradation in query performance. To addresss this issue, as future work, we plan to extend the system to monitor the read latency on the OCM disk cache as well as on the underlying object store, and dynamically re-route requests to the object store when the system determines that the OCM read latency exceeds that of the object store.

The degradation in performance during the warm-up phase on the m5ad.4xlarge instance is less pronounced. More specifically, with the OCM, the slowdown is less than or equal to 5.0% for Q_1 and Q_2, and the performance is improved for Q_3 and Q_4. There are two major differences between the two instances. The first difference is that the smaller m5ad.4xlarge instance has fewer CPUs, which puts less demand on the OCM to begin with. Second, on the m5ad.-4xlarge instance, SAP IQ operates on a much smaller buffer cache (RAM), which affects not only the query plans (i.e., sort-merge vs. hash joins) but also the way the data flow through OCM. In particular, having a smaller buffer cache puts more pressure on the OCM, but the demand on the OCM is more evenly spread out across the execution of the query. In contrast, with a larger buffer cache, there is less pressure on the OCM, but requests come in bursts, which, for reasons discussed earlier, can hurt performance. A combination of these factors are likely to contribute to the differences we notice between the two runs of the experiment. Some of these brownout problems may be addressed by introducing a persistent or distributed cache such as Alluxio [37], or by implementing a proactive warmup mechanism such as the one used in Amazon Redshift [35]. However, these enhancements come with additional storage and/or compute costs especially when node restarts are infrequent. Studying these trade-offs is left as future work.

The OCM plays an important role from a monetary point of view as well. Specifically, during the execution of the benchmark queries, the OCM has been able to avert 2,807,368 GET requests on S3 (with a hit rate of 74.5%), which corresponds to a total reduction of $1.12 in cost (i.e., 32% savings).

In the third experiment, we evaluate the *scale-up* characteristics of our approach. For this experiment, we load the TPC-H benchmark data at scale factor 1000 into a cloud dbspace on S3 and run the 22 benchmark queries. We repeat the experiment on instances with increasingly larger capacity (i.e., RAM, SSD and CPU), namely, m5ad.4xlarge, m5ad.12xlarge and m5ad.24xlarge.

In Figure 7, we plot (i) the load time, (ii) the time to sequentially execute the 22 benchmark queries, and (iii) the total time to run the full benchmark suite as a function of the number of CPUs (note the log-log scale). We observe almost-linear scalability; nevertheless, the performance gains going from 48 to 96 CPUs are slightly less than the ones going from 16 to 48 CPUs. The difference is more visible during the load phase. During the experiment, we have observed that the network bandwidth was saturated at slightly more than 9 Gigabits per second (Figure 8). Since the md5ad.24xlarge

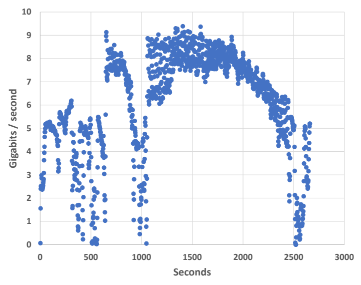

Figure 8: Network Bandwidth Utilization During Load

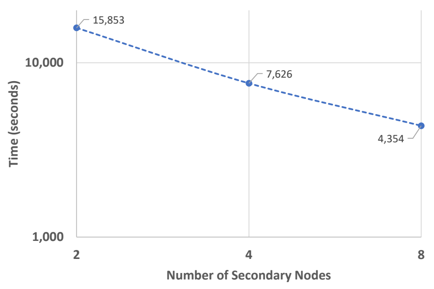

Figure 9: Scale-out Behavior

node supports up to a bandwidth of 20 Gigabits per second, we believe that the limitations are intrinsic to our system (e.g., due to the 512KB restriction on page sizes in SAP IQ, changing which is beyond the scope of this paper). Therefore, to achieve better scalability, we conclude that one must start adding more nodes into the system (i.e., scale-out), which is evaluated next.

In the fourth and last experiment, we execute the TPC-H benchmark at scale factor 1000 in throughput mode. In particular, we construct 8 query streams that consist of pseudo-random permutations of the benchmark queries, and execute these streams in parallel. Since we are interested in understanding the scale-out behavior of the system, we rely on a multiplex setup with a coordinator node and 2, 4 and 8 secondary nodes, respectively. The query streams are balanced evenly across the secondary nodes; that is, in the case of 2 secondary nodes for example, each secondary node gets to execute 4 query streams in parallel. For the secondary nodes, we spawn off m5ad.4xlarge instances but for the coordinator node, a smaller r5.large instance suffices as we do not execute the queries directly on the coordinator. In addition, system dbspaces need to be stored on a shared volume that can be accessed from all multiplex nodes. Therefore, we use an EFS volume as opposed to EBS.

Figure 9 displays the total time to concurrently execute all query streams as a function of the number of secondary nodes (note the log-log scale). We observe that the system scales very well as secondary nodes are added: doubling the number of secondary nodes almost halves the time it takes to execute the workload. One of the reasons why we are able to achieve good scalability is because we are able to increase the combined throughput on S3 – almost in an unlimited way – when nodes are added to the setup. If the user dbspaces were to be stored on shared block storage volumes on the cloud (e.g., EBS, EFS, NetApp, etc.), we would not have been able to achieve the same degree of scale-out because on these volumes, the throughput is a function of the data size or the provision type. In other words, the way to increase throughput on these volumes would be to increase the data size or to pay more for provisioned IOPS/bandwidth. In either case, the independence between the scalability of compute and the scalability of storage would have been broken.

7 RELATED WORK

Disk-based, columnar database management systems systems such as *IQ*, *Vertica* and *MonetDB* were developed in the early 90's to support online analytical processing (OLAP) workloads [18, 36, 39, 49]. As RAM became cheaper and more accessible at large capacities, in-memory column stores such as *SAP HANA* have emerged [34]. More recently, with the advancement of cloud technologies, and in particular, cheap, elastically scalable object storage solutions such as Amazon Simple Storage Service (S3) [5], Azure Blob Storage [11], Google Cloud Storage [15] and Alibaba Cloud Object Storage Service (OSS) [1], there has been a significant effort to develop database management systems on the cloud [23, 31, 40, 51, 53]. We classify these development efforts into three major streams:

1. Big Data systems such as NoSQL systems, distributed query engines or emerging datalake systems [7, 10, 13, 24, 26, 29, 33, 37, 44, 46],
2. Data warehouses offered purely as software-as-a-service (SaaS) products [14, 30, 35], and
3. Relational OLAP systems transitioning to the cloud [52].

Next, we discuss some of these offerings in more detail.

Apache Hive was initially designed to be an open source NoSQL engine, focusing on highly parallel Extract-Transform-Load (ETL) or batch processing workloads, where the system would execute Hadoop [9] or MapReduce [32] jobs over data stored on the Hadoop File System (HDFS) [48]. Being a NoSQL engine with no ACID guarantees, the system would easily scale out to thousands of compute nodes [50]. Over time, there has been an increasing demand for a more complete support for the SQL language, as well as for providing transactional guarantees [26]. To this end, the system has evolved to support snapshot isolation [25] in the case of workloads involving single-statement transactions [26].

PNUTS (Sherpa) went through a similar journey, where it started out as a massively distributed key-value store and evolved into a system that can store distributed tables, ordered and partitioned on primary keys [29]. PNUTS supports single-row transactions only, and its API is not SQL-compatible.

Presto a distributed query engine that has been developed at Facebook [46]. Presto allows querying data directly at its source; therefore, it is not truly a relational engine. It supports a wide subset of the SQL dialect, has a sophisticated query optimizer that can perform predicate pushdown, and the system can scale-out well by exploiting inter-node parallelism [46]. It uses techniques such as compiled query plans [43] to significantly speed up query processing.

Delta Lake is a data lake solution that is developed directly over cloud object stores [24]. The system offers the ability to modify Delta tables in an ACID manner where table contents as well as the write-ahead transaction logs are stored as Apache Parquet [8] objects. The system supports features such as time travel and caching data and log objects beyond the lifetime of compute nodes.

Snowflake is a data warehousing platform that is built from scratch to provide a pure software-as-a-service experience on the cloud [30, 54]. Snowflake is a relational OLAP engine with support for both SQL and ACID transactions. Snowflake relies on a shared-nothing compute model on AWS EC2 [3] and shared storage on AWS S3 [5]; therefore, storage and compute resources can be scaled independently and elastically [30]. In Snowflake, data are laid out using the Partition Attributes Across (PAX) approach [22] where the tables are first horizontally partitioned, and then within each partition, data are stored on AWS S3 using the columnar format [30].

Redshift is Amazon's software-as-a-service OLAP engine [35]. It is built on top of other AWS services including EC2 for compute, S3 for storage and Amazon Simple Workflow (SWF) [6] for monitoring, metering and tooling. The system is designed with simplicity in mind. In particular, it tries to simplify database administration and tuning as much as possible with automated "provisioning, patching, monitoring, repair and backup and restore" [35]. Data are distributed and replicated across three tiers. Each data block is stored in local storage on both a primary node assigned to that block and a secondary node. Finally, it is also stored on S3. All three tiers are available for reads during querying. In Redshift, queries are compiled at the leader node into machine code [43] and the executables are shipped to query nodes for execution.

Vertica is a columnar relational database management system designed for analytical workloads [39]. In the enterprise mode, its shared-nothing storage system, which is built using locally attached disks, allows the sytem to execute workloads in a massively parallel fashion [39]. The downside is that storage cannot be scaled independently of compute, which is an undesirable property on the cloud. Consequently, for its *EON* mode (which is their offering on the cloud), Vertica has significantly re-architected its storage, metadata, and fault tolerance subsystems to be able to operate on shared storage (i.e., AWS S3) [52]. The query execution engine remains largely unchanged, which relies on local caching to shield the effects of the changes made in the underlying storage subsystems [52].

In summary, Big Data systems are designed for exploiting the massive parallelism offered by the cloud, often at the expense of ACID guarantees or the full SQL syntax. While these systems can scale-out to thousands of compute nodes, they are not truly relational. Data warehouses that are built from bottom up to offer pure SaaS experience on the cloud are maturing, but they face different design challenges. For example, in Snowflake, the choice of the partition size can amplify the trade-offs between supporting frequent updates vs. efficient querying. Likewise, Redshift benefits significantly from query-time compilation [43] and aggressive intra-query parallelism on single-user workloads; nevertheless, as *Tan et al.* point out, these optimizations may play to the system's disadvantage on heterogeneous or changing workloads [51]. While systems like Vertica and SAP IQ are mature relational OLAP systems, they were not initially designed to operate on the cloud: Vertica's journey has been primarily focused on converting its shared-nothing architecture to operate on shared storage on AWS S3. In contrast, SAP IQ has relied on a shared storage architecture from the beginning but had strong consistency requirements which had to be addressed to be able to utilize object stores.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we have described our experience in transforming SAP IQ into a cloud RDBMS, with a particular emphasis on supporting elastically scalable object stores such as AWS S3 and Azure Blob Storage. Traditionally, SAP IQ has been designed to operate on a shared storage model with strong consistency guarantees and on devices that offer low I/O latency. In contrast, object stores support a weaker consistency model and have a much higher I/O latency. To overcome these limitations, we have (i) enforced a direct mapping from logical pages in SAP IQ to objects in object stores, (ii) enhanced SAP IQ's *copy-on-write* semantics to guarantee that objects are never written twice, thus, dealing with the eventual consistency issues of object stores, (iii) extended our transaction manager to appropriately manage the lifecycle of pages that are stored on object stores (i.e., garbage collection), and (iv) implemented a second layer of read/write cache, namely, the Object Cache Manager, that utilizes locally attached SSD/HDD to mitigate the high I/O latency on object stores. Using these techniques, we have been able to achieve an order of magnitude reduction in data-at-rest storage costs while improving query and load performance.

As future work, we consider multiple areas of improvement. First, we plan to extend our snapshot gear to be able to create read-only views over past snapshots in an existing database without having to recover the database from a snapshot. Second, we consider implementing the snapshot manager as an elastically scalable micro-service, outside of the IQ engine. Third, we examine the possibility of creating cloud dbspaces with custom page sizes. The requirement of having a unified page size across the whole database was primarily driven by the characteristics of shared block devices that do not necessarily apply to object stores. Having dbspaces with different page sizes will allow users to fine-tune their databases for mixed workloads where, for example, some tables are updated frequently in small batches, some tables are updated less frequently in larger batches, and the remaining tables are mostly read-only.

ACKNOWLEDGMENTS

The authors would like to thank Calvin Hua, Sammed Kanwade, Rahulkumar Rank, Ankit Sharma, Nishant Sharma, Sagar Shedge and Ashutosh Singh for their significant engineering contributions to this project. The authors are also grateful for the efforts of the countless architects, developers, development managers and product managers that have worked on SAP IQ (formerly known as Sybase IQ) in the past three decades, upon whose contributions this project is founded. The well thought-out design of SAP IQ's storage subsytem, in particular, the choice of a shared storage model as well as the abstraction made by the blockmap objects allowed this project to progress in a timely fashion. Lastly, the authors are grateful for the anonymous feedback provided by the reviewers.

REFERENCES

[1] Alibaba Object Storage Service. https://www.alibabacloud.com/product/oss/.

[2] Amazon Elastic Block Store (EBS). https://aws.amazon.com/ebs/.

[3] Amazon Elastic Compute Cloud (EC2). https://aws.amazon.com/ec2/.

[4] Amazon Elastic File System (EFS). https://aws.amazon.com/efs/.

[5] Amazon Simple Storage Service (S3). http://aws.amazon.com/s3/.

[6] Amazon Simple Workflow Service (SWF). https://aws.amazon.com/swf/.

[7] Apache Hudi. https://hudi.apache.org/.

[8] Apache Parquet. https://parquet.apache.org/.

[9] Apache Software Foundation, Hadoop. https://hadoop.apache.org/.

[10] Apache Spark – Lightning-fast unified analytics engine. https://spark.apache.org/.

[11] Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs/.

[12] Best practices design patterns: Optimizing Amazon S3 performance. https://docs.aws.amazon.com/AmazonS3/latest/dev/optimizing-performance.html.

[13] Dremio. https://www.dremio.com/product/.

[14] Google BigQuery. https://cloud.google.com/bigquery/.

[15] Google Cloud Storage. https://cloud.google.com/storage/.

[16] NetApp – Data Management Solutions for the Cloud. https://www.netapp.com/.

[17] SAP HANA Cloud. https://www.sap.com/products/hana/cloud.html.

[18] SAP IQ. https://www.sap.com/canada/products/sybase-iq-big-data-management.html.

[19] SAP IQ Performance and Tuning Guide – Zone Maps. https://help.sap.com/viewer/a8982cc084f21015a7b4b7fcdeb0953d/16.1.3.0/en-US/6604c2567d66453da391dee00dcf5d5c.html.

[20] TPC Benchmark H (decision support) standard specification. http://www.tpc.org/tpch/.

[21] A. Agarwal, S. A. Kirk, B. French, N. Marathe, S. Mungikar, and K. Mittal. Tiered index management, 2013. US Patent 10061792B2.

[22] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data page layouts for relational databases on deep memory hierarchies. *VLDB J.*, 11(3):198–215, 2002.

[23] P. Antonopoulos, A. Budovski, C. Diaconu, A. H. Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U. F. Minhas, N. Prakash, V. Purohit, H. Qu, C. S. Ravella, K. Reisteter, S. Shrotri, D. Tang, and V. Wakade. Socrates: The new SQL server in the cloud. In *Proceedings of the 2019 ACM International Conference on Management of Data, SIGMOD*, pages 1743–1756, 2019.

[24] M. Armbrust, T. Das, S. Paranjpye, R. Xin, S. Zhu, A. Ghodsi, B. Yavuz, M. Murthy, J. Torres, L. Sun, P. A. Boncz, M. Mokhtar, H. V. Hovell, A. Ionescu, A. Luszczak, M. Switakowski, T. Ueshin, X. Li, M. Szafranski, P. Senster, and M. Zaharia. Delta lake: High-performance ACID table storage over cloud object stores. *Proc. VLDB Endow.*, 13(12):3411–3424, 2020.

[25] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM International Conference on Management of Data, SIGMOD*, pages 1–10, 1995.

[26] J. Camacho-Rodríguez, A. Chauhan, A. Gates, E. Koifman, O. O'Malley, V. Garg, Z. Haindrich, S. Shelukhin, P. Jayachandran, S. Seth, D. Jaiswal, S. Bouguerra, N. Bangarwa, S. Hariappan, A. Agarwal, J. Dere, D. Dai, T. Nair, N. Dembla, G. Vijayaraghavan, and G. Hagleitner. Apache Hive: From MapReduce to enterprise-grade big data warehousing. In *Proceedings of the 2019 ACM International Conference on Management of Data, SIGMOD*, pages 1773–1786, 2019.

[27] C. Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *Proceedings of the 1998 ACM International Conference on Management of Data, SIGMOD*, pages 355–366, 1998.

[28] D. Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[29] B. F. Cooper, P. P. S. Narayan, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS to sherpa: Lessons from yahoo!'s cloud database. *Proc. VLDB Endow.*, 12(12):2300–2307, 2019.

[30] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. The snowflake elastic data warehouse. In *Proceedings of the 2016 ACM International Conference on Management of Data, SIGMOD*, pages 215–226. ACM, 2016.

[31] S. Das, M. Grbic, I. Ilic, I. Jovandic, A. Jovanovic, V. R. Narasayya, M. Radulovic, M. Stikic, G. Xu, and S. Chaudhuri. Automatically indexing millions of databases in microsoft azure SQL database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD*, pages 666–679, 2019.

[32] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation, OSDI*, pages 137–150, 2004.

[33] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 2007 ACM Symposium on Operating Systems Principles, SOSP*, pages 205–220, 2007.

[34] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, 2011.

[35] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon Redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM International Conference on Management of Data, SIGMOD*, pages 1917–1923, 2015.

[36] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[37] C. Jia and H. Li. Virtual distributed file system: Alluxio. In *Encyclopedia of Big Data Technologies*. Springer, 2019.

[38] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[39] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandier, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *Proc. VLDB Endow.*, 5(12):1790–1801, 2012.

[40] F. Li. Cloud native database systems at alibaba: Opportunities and challenges. *Proc. VLDB Endow.*, 12(12):2263–2272, 2019.

[41] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[42] S. Mungikar and B. French. Smart pre-fetch for sequential access on BTree, 2013. US Patent 9552298B2.

[43] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, 2011.

[44] R. Ramakrishnan, B. Sridharan, J. R. Douceur, P. Kasturi, B. Krishnamachari-Sampath, K. Krishnamoorthy, P. Li, M. Manu, S. Michaylov, R. Ramos, N. Sharman, Z. Xu, Y. Barakat, C. Douglas, R. Draves, S. S. Naidu, S. Shastry, A. Sikaria, S. Sun, and R. Venkatesan. Azure data lake store: A hyperscale distributed file service for big data analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD*, pages 51–63, 2017.

[45] G. M. Sacco. Buffer management. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 277–282. Springer US, 2009.

[46] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner. Presto: SQL on everything. In *Proceedings of the 2019 IEEE International Conference on Data Engineering, ICDE*, pages 1802–1813, 2019.

[47] M. Sharique, A. K. Goel, and M. Andrei. Rollover strategies in a n-bit dictionary compressed column store, 2013. US Patent 9489409B2.

[48] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE Conference on Mass Storage Systems and Technologies, MSST*, pages 1–10, 2010.

[49] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented DBMS. In *Proceedings of 2005 International Conference on Very Large Data Bases, VLDB*, pages 553–564, 2005.

[50] M. Stonebraker, D. J. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[51] J. Tan, T. Ghanem, M. Perron, X. Yu, M. Stonebraker, D. J. DeWitt, M. Serafini, A. Aboulnaga, and T. Kraska. Choosing A cloud DBMS: architectures and tradeoffs. *Proc. VLDB Endow.*, 12(12):2170–2182, 2019.

[52] B. Vandiver, S. Prasad, P. Rana, E. Zik, A. Saeidi, P. Parimal, S. Pantela, and J. Dave. Eon mode: Bringing the vertica columnar database to the cloud. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 ACM International Conference on Management of Data, SIGMOD*, pages 797–809, 2018.

[53] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD*, pages 1041–1052, 2017.

[54] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes. Building an elastic query engine on disaggregated storage. In *Proceedings of the 2020 Symposium on Networked Systems Design and Implementation, USENIX*, pages 449–462, 2020.