SwiftAnalytics: Optimizing Object Storage for Big Data Analytics

Lukas Rupprecht*, Rui Zhang‡, Bill Owen♯, Peter Pietzuch*, Dean Hildebrand‡
*Imperial College London, ‡IBM Research Almaden, ♯IBM Tucson
lr12@imperial.ac.uk, ruiz@us.ibm.com, billowen@us.ibm.com, prp@imperial.ac.uk, dhildeb@us.ibm.com

***Abstract*—Due to their scalability and low cost, object-based storage systems are an attractive storage solution and widely deployed. To gain valuable insight from the data residing in object storage but avoid expensive copying to a distributed filesystem (e.g. HDFS), it would be natural to directly use them as a storage backend for data-parallel analytics frameworks such as Spark or MapReduce. Unfortunately, executing data-parallel frameworks on object storage exhibits severe performance problems, reducing average job completion times by up to 6.5×.**

We identify the two most severe performance problems when running data-parallel frameworks on the OpenStack Swift object storage system in comparison to the HDFS distributed filesystem: (i) the fixed mapping of object names to storage nodes prevents local writes and adds delay when objects are renamed; (ii) the coarser granularity of objects compared to blocks reduces data locality during reads. We propose the SwiftAnalytics object storage system to address them: (i) it uses *locality-aware* writes to control an object's location and eliminate unnecessary I/O related to renames during job completion, speeding up analytics jobs by up to 5.1×; (ii) it transparently chunks objects into smaller sized parts to improve data-locality, leading to up to 3.4× faster reads.

I. Introduction

Object-based storage systems such as Amazon S3 [1], OpenStack Swift [2], and Cleversafe [3] have established themselves as a prominent solution for large-scale storage due to their almost unlimited scalability and cost-efficiency. With businesses storing more data in object storage, the insights that can be extracted from the data have tremendous value.

To avoid copying data into a distributed filesystem (DFS) for analysis first, new efforts have been made to use popular data-parallel analytics frameworks such as MapReduce [4] or Spark [5] directly on top of object storage [6] [7]. This saves both cost and administration effort as no additional data silo must be set up and data can be managed in one place.

Object-based storage systems differ fundamentally from the DFSs conventionally integrated with analytics frameworks. DFSs such as HDFS [8] were specifically designed to provide high read and write throughput for data processing applications; the focus of object storage is on high scalability coupled with a simple API to ease manageability. Object-based storage does not organize, locate or replicate data in the same way as DFSs: it uses a *hash function* to locate data, while a DFS relies on (centralized) *metadata servers*. The lack of metadata servers in object storage systems is the reason for their superior scalability and ease of deployment, but it is also the source of performance limitations for analytics workloads.

In this paper, it is our goal to help users reap the benefits of running analytics jobs on object storage by considering the performance challenges of such a setup. We target a colocated architecture, i.e. storage and compute nodes are deployed on the same cluster as this avoids the additional delay of remote data reads/writes and offers the best performance. We find that, if jobs are executed without changes to the storage layer, they experience a slow-down of up to 6.5×.

We identify two main problems affecting performance: (i) **write-locality**—object storage systems work on the principle of consistent hashing that allows locating objects across the cluster in a decentralized way. This creates a fixed mapping between object names and storage devices, which does not work well for some operations. For example, renaming an object degenerates to re-uploading the same file with a different name. As data-parallel frameworks require several renames of the job results, performance decreases significantly; (ii) **read locality**—object storage systems, in contrast to DFSs, do not typically chunk objects across all servers, but rather upload them in their entirety. This coarser storage granularity generates imbalances in the I/O load that reduces data locality.

Existing solutions have tried to solve the write-locality problem in data-parallel analytics jobs by simply avoiding expensive calls, rename in particular, during a job [9] [10]. However, these solutions assume no control over the object storage systems and hence operate at the layer between the data-parallel framework and the storage. This restricts their possibility for improvement. Instead, we propose to alter the object storage itself to robustly speed up analytics jobs.

We make the following contributions: We propose SwiftAnalytics, an enhanced object storage system based on OpenStack Swift that allows to seamlessly integrate analytics frameworks and object storage without significant performance penalties (§III). To address the write-locality problem, SwiftAnalytics uses *locality-aware writes*, which offer placement control to the object storage system and support efficient object renaming (§IV). Locality-aware writes break the dependency between the object name and its hash-defined location without introducing a centralized component and hence do not affect scalability. SwiftAnalytics achieves this by storing the additional location information in a decentralized way such that it can be located with negligible overhead. SwiftAnalytics solves read-locality with a straight-forward chunking scheme which, similar to HDFS, chunks objects into smaller parts to achieve

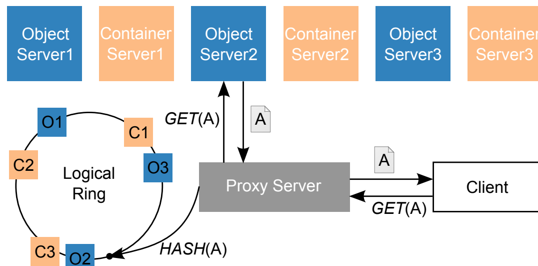

Fig. 1: Swift architecture

a better distribution across the cluster.

We deploy MapReduce and Spark on top of SwiftAnalytics on a local research testbed and evaluate its effectiveness using a set of workloads with different characteristics (§V). With locality-aware writes, we show that result data can be written up to 8.5× faster, translating to an overall improvement of job completion time by 5.1× compared to an optimized Swift deployment; object-chunking can provide an up to 3.4× improvement when reading input data. We discuss related work in §VI and then conclude the paper in §VII.

II. Data-parallel Processing on Object Storage

Next we briefly review the main concepts behind object-based storage (§II-A) and then discuss how data-parallel frameworks interact with the storage layer during a job (§II-B).

A. Object Storage Basics

In an object-based storage system, data is stored and exposed to clients at the granularity of *BLOBs* as compared to blocks in a traditional filesystem [11] [12]. A BLOB can be any kind of data such as images or documents and is usually immutable [13]. Interaction happens via a RESTful API, with facilities to store (`PUT`), retrieve (`GET`) and delete (`DELETE`) objects based on their keys. Object storage exposes a flat namespace in which the key is an arbitrary identifier for the object, in most cases, its name. An additional layer of hierarchy to group objects, called *containers* (Swift) or *buckets* (S3), is also provided. Data is replicated for reliability.

With "object storage", we refer to large-scale enterprise storage systems such as S3 [1], Walnut [12], or Haystack [14]. These storage systems provide highly scalable storage for unstructured data, which is of particular interest for large-scale analytics. We use *OpenStack Swift* [2] as the underlying storage layer for the analytics framework but the general concepts also apply to other enterprise-class object storage.

A major advantage of object storage over distributed filesystems is their scalability. Swift scales due to two properties: (i) it uses *consistent hashing* [15] to locate objects without a centralized metadata service. In consistent hashing, storage nodes are randomly assigned positions in a logical ring structure. The ring is discretized by the output domain of a hash function and closed at the smallest and largest hash values. The location of an object is then determined by hashing its name and assigning it to the node whose position is closest to the object's hash on the ring (see Figure 1); (ii) all object metadata, such as its creation time or checksum, is stored with an object instead of on a separate metadata server. These two properties allow object storage systems to keep all metadata decentralized and hence, no single scalability bottleneck exists.

Figure 1 shows the request processing path in Swift. When a client submits a request to retrieve an object A, it contacts a proxy server. While the proxy is a single entry point to the storage cluster, it is stateless and hence can be scaled arbitrarily. After receiving the request, the proxy determines the responsible object server for the object. It then forwards the request to that object server and returns its response to the client. Besides the object servers, the cluster has container servers that store the listings of objects grouped within a container. Containers are also located via the hash function.

B. Storage Interaction of Analytics Frameworks

Data-parallel frameworks such as Spark and Hadoop MapReduce expose a simple operator-based API. For example MapReduce consists of two operators, `map` and `reduce`, which are executed in two consecutive phases. Spark offers a wider set of operators, e.g. join or union. Operators are automatically parallelized across nodes in the cluster, and each parallel *task* executes the operator function on a partition of the input data.

Input data is usually kept in a distributed filesystem to read data in parallel. HDFS is the most prominent choice as it was co-designed with Hadoop MapReduce and is now supported by all major frameworks. It has a master/slave architecture with the *NameNode* as the central entity. In HDFS, a file is chunked into fixed sized blocks of 128 MB [16] when uploaded. Blocks are distributed and replicated across the cluster. The NameNode is responsible for mapping file blocks to storage servers and keeps all metadata, including block locations, for the files. Analytics frameworks interact with the distributed storage at two stages during a job: (i) when reading the input data and (ii) when writing the results.

When reading input data, a number of *input tasks* is spawned to read a partition. In the default case, one input task reads a single HDFS block. As blocks provide a fine storage granularity, I/O load can be spread evenly to achieve high aggregate disk throughput. In addition, input tasks can be colocated on the nodes storing their input data to avoid network I/O and achieve high *data locality*.

When a job has finished, the tasks from the last operator write the results back to the distributed storage. Tasks follow a two-phase commit protocol to commit their outputs. First, a reduce task writes its output to a temporary location using chain-replication. The task then informs the application master, i.e. the node responsible for managing the job, that it has finished. If allowed to commit, the node moves the temporary file to its final location and exits. This protocol is necessary, if optimizations such as *speculative execution*, in which multiple tasks process the same input data, should be supported. It prevents speculative tasks from overwriting each others results. After all reduce tasks have finished, the application master renames the output directory from an internal staging directory to the output directory specified by the user.

III. SwiftAnalytics Design

Next, we present the design of SwiftAnalytics, an enhanced object storage system that solves the performance problems of analytics jobs when running on object storage.

A. Performance Issues for Analytics on Swift

We first describe the most severe problems, which guide the design of SwiftAnalytics (see [17] for more details).

Single proxy server. As described above, Swift uses a proxy server as an entry point to the cluster. Users interact with the cluster via the proxy and hence reads and writes of objects are redirected through the proxy. For an analytics job, during which multiple tasks read and write in parallel, the proxy server becomes the limiting factor. However, the proxy server is stateless and can thus be replicated. As a result, it is possible to run different proxy servers on each node in the cluster and have each task read/write through its local instance. This removes the single proxy bottleneck. We use this simple optimization as a default in our evaluation.

Read locality. In Swift, objects are not chunked when uploaded, which is in contrast to HDFS, which splits files into smaller blocks. Additionally, an object's location is determined through consistent hashing on the ring, i.e. its location is fixed while HDFS can dynamically select the storage device for a block. This provides a more fine-grained way for HDFS to distribute blocks, with the goal of achieving an even read I/O load across all nodes, compared to Swift, which causes skewed I/O and generates hotspots. This can slow down the map phase of a job by up to 4.6× (see §V).

Write locality. As Swift lacks a centralized metadata server and locates objects via consistent hashing, renaming an object becomes an expensive operation. Instead of updating a metadata entry, the object must be re-uploaded to the location specified by the hash of the new name. Combined with the rename-based two-phase commit protocol and the directory rename that the application master performs at the end of a job (see §II-B), analytics jobs experience a major performance decrease. We observe slowdowns of up to 15.8× during the reduce phase due to the additional copy overhead (see §V).

The design of SwiftAnalytics incorporates two additional features compared to standard object storage system to tackle the read- and write-locality problems: (i) transparent *object chunking* to increase parallelism during reads and (ii) efficient object renames by providing *placement control* to clients.

B. Object Chunking

A chunking mechanism allows clients to transparently split objects into smaller parts, similar to HDFS blocks, when uploading data to the object storage. However, contrary to HDFS, there is no central instance to keep a block map for an uploaded object that stores the parts and their locations for a single object. Hence, a decentralized mechanism is needed to collect and assemble all chunks of an object. Swift already offers a way to achieve this. To support arbitrary object sizes, objects larger than 5 GB are split, and the individual chunks are linked to from a manifest file, stored under the original object name. While this is typically used to manage large objects in Swift, it can also be used to transparently support small, equal-sized chunks for faster analytics.

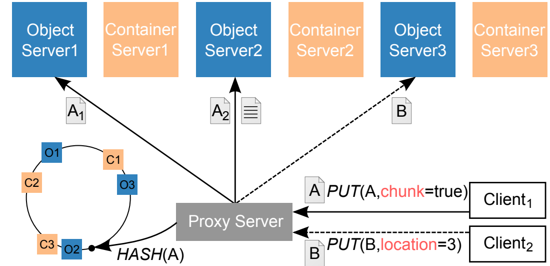

Fig. 2: SwiftAnalytics design

Figure 2 illustrates this process. On an upload, a client can specify to chunk the object which will cause the proxy server to split the incoming data into several parts, e.g. 128 MB blocks as in HDFS. The different parts will receive internal names according to which they are placed across the cluster. Additionally, a manifest object under the original name is created, which contains an ordered list of the names of all object parts. Upon a `GET`, the proxy retrieves the manifest file, parse it, and return the parts in order to the client.

C. Placement Control

Besides chunking, SwiftAnalytics provides object placement control to clients for specifying the object server on which an object should be stored (see Figure 2). This allows the implementation of efficient renaming. The reason why renaming is slow in object-based storage systems is that the object location depends on the name of the object. When an object receives a new name, its location changes, triggering a copy to a new object server. With placement control, the new object server can be specified explicitly to avoid the extra copy.

The main challenge for enabling placement control in object storage is to not introduce any centralized component which would limit scalability. Distributed filesystems such as HDFS have a central metadata service that stores a mapping between all file names in the system and their corresponding locations. This allows for fast renames as a rename only requires updating the name of the target file in that mapping. Given that a main advantage of object storage is its scalability due to the decentralized architecture, adding such a centralized service is undesirable. In the following section, we present *locality-aware writes*, a mechanism implemented in SwiftAnalytics to enable *decentralized* placement control.

A simple alternative to placement control for fast renames is a symlink-like solution, which we call *link files*. As in symlinks, a link file does not contain the actual contents but rather a pointer to the original file. When an existing object should be renamed, a new link file with a pointer to the original object is uploaded and replicated, leaving the actual data untouched. When the object is requested, the object storage system follows the link to return the original content. We compare SwiftAnalytics to link files in §V.

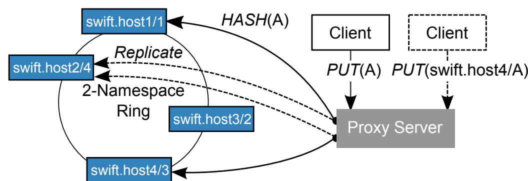

Fig. 3: Locality-aware writes scheme

IV. Locality-aware writes

Next we present locality-aware writes (§IV-A), how they can be used to provide fast renames (§IV-B), and their implications on object storage systems (§IV-C).

A. Overview

Locality-aware writes use a two-namespace approach to allow clients to specify the target location of an object. This is similar to the idea used in SkipNet [18]. However, we make significant adjustments to suit the needs of analytics on object storage. In its basic operation, a consistent-hashing based object storage system uses the numerical hash value of an object to determine its location, i.e. the hash IDs form a *numeric* namespace. This namespace is agnostic to any locality constraints in the cluster as it is only a logical overlay. In order to support locality, locality-aware writes use a second *lexicographic* namespace that explicitly captures the locations, i.e. the URIs of devices in the cluster. (see Figure 3).

On a `PUT`, a client can now set an additional request attribute, the *location ID* (LID), to specify on which node the object should be placed. If set, the LID has higher precedence than the numeric ID and bypass the hash placement. Figure 3 shows the two possible ways of uploading an object. To keep the object storage decentralized, the LID is stored as a metadata attribute *with* the object itself and its replicas. Hence, no centralized service to store LIDs is required.

Since locality-aware writes break the hash placement, object retrieval can now fail if an object has been stored using an LID. To be able to still locate an object, three techniques are used:

1) The LID is only used to determine the location of the *primary* replica. All *additional* replicas are normally placed according to the hash (see Figure 3). By accessing one of these additional replicas and reading the LID from their metadata, the primary replica can be located.
2) To avoid losing the object in case all additional replicas fail, the LID is added to the metadata of the object's container. Containers are replicated in the same way as objects so the LID is replicated another N times.
3) On deletion, a `DELETE` is sent out to all replicas simultaneously and, if a replica detects that an LID has been set on its metadata, it can forward the request to the corresponding location on which the primary replica is stored.

B. Local Rename and Upload Operations

Using locality-aware writes, SwiftAnalytics implements an efficient *local rename* strategy that works as follows: Swift internally uses the hash of an object's name to not only locate the responsible object server but also to find the object locally on disk by using the hash as the folder name in which the object is stored. In case of a rename operation, the object server will *locally move* the source file to its destination folder, given by the new hash value, using the Unix `mv` command. This is only a metadata operation and does not cause I/O.

SwiftAnalytics also provides fast *uploads* using the above approach. If a client runs co-located with an object server and has a locally stored object, it can simply move this object to the correct folder in Swift instead of copying it to a remote destination. As no I/O is generated during such a *local upload*, it is efficient. This is useful in case the storage system does not support streamed uploads, i.e. output tasks first need to write their job results to the local disk and then copy it to the object storage. While Swift does support streamed uploads, other systems such as S3 do not natively provide that functionality.

C. Implications on Object Storage Systems

Locality-aware writes come with two key implications for object storage regarding object replication and object retrieval.

Replication. While the local move saves one copy of the existing data, all additional $N-1$ replicas still need to be copied. Swift's reliability model requires a quorum of objects persisted on disk before it reports success. In a 3-way replicated case, at least one additional copy has to be written successfully before the rename can return. However, we found that in the case of an analytics job, these additional copies are wasteful as renames are executed shortly after each other. We identify three possible replication schemes that trade off reliability for performance:

1) Only the primary replica will be moved locally before the rename returns (1 replica).
2) The primary replica will be moved locally and an additional replica will be copied to its new destination. This scheme fulfills Swift's reliability requirements as on return, 2/3 of the replicas have been successfully written (2 replicas).
3) The primary replica will be moved locally and two additional replicas will be copied to their new destinations. This scheme provides additional reliability (3 replicas).

Eventually, all three schemes will have all replicas persisted on disk as Swift uses an asynchronous *replicator* process on each node to correctly handle replication failures. This process periodically scans the filesystem and makes sure objects are replicated correctly. This happens within a specified *consistency window*, which is 30 s by default. We evaluate the different strategies in the following section.

Object retrieval. When using locality-aware writes, `GET` requests for the primary replica of an object can fail and an additional request is needed to look up the LID. The number of additional requests depends on how load is distributed across replicas. In Swift, the proxy server randomly selects one of the replicas to be retrieved, i.e. on average 1/3 of initial requests will fail. We show in §V that the overhead of this additional request is small compared to the reduced completion time for analytics jobs.

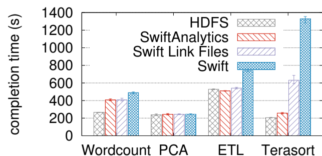

Fig. 4: Renaming strategies (MapReduce workloads)

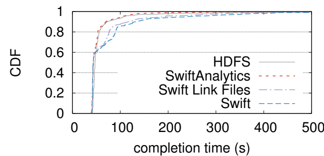

Fig. 5: Renaming strategies (Facebook workloads)

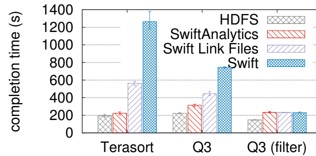

Fig. 6: Renaming strategies (Spark workloads)

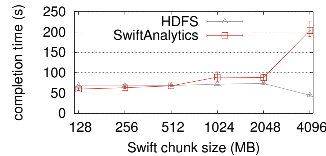

Fig. 7: Varying object sizes

TABLE I: Base workload properties

	Wordcount	PCA	Terasort	ETL
Data read	100 GB	32 GB	32 GB	60 GB
Data written	2.1 GB	1.3 MB	32 GB	7.9 GB
Map tasks	64	64	64	64
Reduce tasks	64	1	64	64

V. Evaluation

We evaluate SwiftAnalytics and compare its performance to vanilla Swift, Swift with link files, and HDFS. We study five aspects: the effectiveness of local renames, the effectiveness of object chunking, the different replication schemes, the effectiveness of local uploads, and the overhead.

Experimental Setup. We deploy Swift 2.2.1 and Hadoop 2.6.0 on a cluster with 16 nodes. Each node has 4 cores at 3.1 GHz, 16 GB of memory, and a 500 GB hard disk, interconnected via 1 Gbps Ethernet. Each node runs a Hadoop DataNode, a Hadoop NodeManager, and a Swift object server and container server. We provision an additional node to run Hadoop's master services. The replication factor is set to 3. To run MapReduce on Swift, we use a recently developed connector [19]. The local filesystems are XFS.

We use four base workloads, as summarized in Table I. The first two workloads, *Wordcount* and *Terasort*, are standard benchmarks, taken from the HiBench suite [20], and include input data; the third workload is *principal component analysis* (PCA). We use the MapReduce PCA implementation from the HIPI project [21], and a 32 GB subset of the 2010 ImageNet dataset [22]; as a fourth workload, we use an *ETL transformation* from the TPC-DI benchmark [23]. We implement a variation of the `CustomerDim` transformation, which scans an XML file for `New` actions and extracts the required fields in CSV format for loading into a data warehouse. Additionally, we use the SWIM workload injector tool [24] to replay a Facebook MapReduce trace of 500 jobs and deploy Spark as another example for a data-parallel framework.

Effectiveness of Local Renames. We start by analyzing the effectiveness of local renames. Note that for this experiment, we use the 1 replica strategy, i.e. do not actively replicate after the local move. We will discuss this choice below in detail.

For the four base workloads, SwiftAnalytics offers the most benefit for ETL and Terasort (see Figure 4) as these write the most result data. It matches the performance of HDFS and considerably outperforms the baseline Swift. While Swift with link files also matches HDFS's performance for ETL, Terasort incurs the heaviest I/O load during renaming in which case SwiftAnalytics outperforms link files by a factor of 2.4× due to its local renames. For Wordcount and PCA, SwiftAnalytics and link files perform similar to HDFS because for these jobs, the output data is small and not affected by the renames. The main reason for the difference between link files and SwiftAnalytics is that link files still replicate data on upload, which gives them better reliability guarantees but lower performance.

Figure 5 shows the CDF of job completion times for the Facebook MapReduce traces. For 60% of jobs, all four setups perform equally. After that, the results start to diverge, and we observe a similar trend as for the above base workloads. Around the 90th percentile, SwiftAnalytics offers an improvement of 1.8× over Swift and 1.5× over link files, showing that it can provide significant benefit for a production workload.

As SwiftAnalytics is a generic solution that optimizes the storage layer, we expect its benefits to also apply to other frameworks. To verify that, we deploy Spark and run Terasort and query 3 from the TPC-H benchmark (Q3) with and without the selection filters to vary the size of the query result. Figure 6 confirms our hypothesis. For Terasort and Q3 without filter, SwiftAnalytics outperforms both vanilla Swift and link files and reaches performance close to HDFS. When the output size is small (Q3 filter), renaming does not affect performance and all setups perform nearly the same.

Effectiveness of Object Chunking. Next, we evaluate the effectiveness of object chunking. To vary the object-to-node ratio, we use a smaller scale Wordcount job on 8 GB of input data and vary the object chunk size. A large chunk size produces a low number of objects and hence, a low object-to-node ratio. Figure 7 shows the map times for the job.

For a large chunk size of 4096 MB the input data is only stored in 2 objects in Swift, which leads to highly skewed access. In that case, SwiftAnalytics experiences a 4.6× slowdown compared to HDFS. As the chunk size decreases, more objects are available, and the performance of SwiftAnalytics converges to that of HDFS. This confirms that chunking is necessary for an even distribution of data across the cluster.

Replication Strategies. We now discuss our choice for the replication strategy. Figure 8 shows the completion times for the base MapReduce workloads for each of the replication methods, as presented in §IV-C. We can see that job completion time increases linearly with each additional replica.

This shows that there is a trade-off between performance and reliability. We argue that, for analytics jobs, less reliability is tolerable. First, as we know that there is one existing copy of the data and within the next consistency window, all replicas

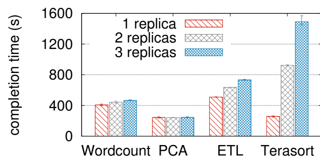

Fig. 8: Replication strategies (MapReduce workloads)

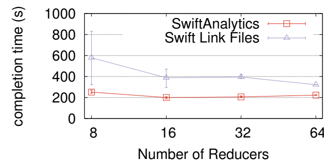

Fig. 9: Comparison under 1-way replication

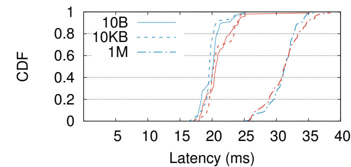

Fig. 10: Request latencies for different object sizes

are generated. Second, if results are lost during this window, it is easy to regenerate them. Figure 8 shows that it is faster to run a Terasort job twice with only a single replica than running it once with an additional replica. For these reasons, we use local renames with no replication as the default.

Effectiveness of Local Uploads. The main reason why SwiftAnalytics outperforms link files is that link files leverage 3-way replication in Swift. To compare the two approaches under equal conditions, we deploy Swift with link files with a replication factor of 1 and run the Terasort job with a varying number of reducers. Figure 9 shows the results.

We see that SwiftAnalytics still performs better than Swift with link files. While 1-way replication improves job completion time with link files by a factor of 2, it is still 45% slower compared to SwiftAnalytics with 64 reducers. This discrepancy increases for smaller numbers of reducers and reaches 129% for 8 reducers. SwiftAnalytics produces stable completion times in spite of fewer reducers, whereas the variance for link files increases heavily. The reason is that when using Swift with link files, objects still are written locally and then uploaded, creating an additional copy, whereas SwiftAnalytics can leverage its local upload mechanism to avoid that copy. The effect of the additional I/Os generated by link files are exacerbated when fewer reducers are used, because a single reducer has to perform more I/Os.

Overhead. Finally, we assess the overhead of locality-aware writes by comparing the latency of `GET` requests in Swift and SwiftAnalytics. We create objects of different sizes and then upload these both with and without an LID. A client sends continuous `GET` requests to the objects, and we measure the latency. The CDFs are shown in Figure 10. While incurring some overhead, local renames only require an additional lookup in 1/3 of all cases, as Swift already performs random load balancing. Additionally, the overhead is in the range of milliseconds and becomes negligible for larger objects.

VI. Related Work

Object Storage Systems. Many object storage architectures have been proposed in the literature such as Dynamo [25], Rados [26], Haystack [14], or F4 [13]. While different in their purpose, they do not provide efficient analytics. Walnut [12] differs from the above as it proposes to unify different storage backends, including HDFS. This supports the motivation of this paper, however, Walnut's main focus is on the unification of different architectures and not primarily on the performance of analytics applications.

Analytics on Different Storage. Existing research investigates analytics on various storage architectures other than HDFS such as PVFS [27] and IBM Spectrum Scale [28]. Neither system performs well out-of-the-box and similar to Swift, optimizations are needed to match HDFS. In MixApart [29], the authors use a disk cache and a transfer scheduler to serve an analytics framework directly out of an enterprise storage backend. Although object storage systems can be one possible backend, it has not been explicitly considered.

Placement Control. The approach of two separate namespaces is shared with SkipNet [18] and its extension [30]. SkipNet uses two namespaces to control placement in a distributed hash table (DHT) and enables content and path locality. However, SkipNet does not provide fast rename and upload operations and is targeted at DHTs and not object storage. The CRUSH algorithm [31] in Ceph [32] uses hierarchies of buckets and placement rules to identify replica locations in a cluster. While the rules enable coarse placement control, CRUSH does not allow to pick the actual storage device.

Avoiding Renames. The Hadoop/Spark community tries to solve the performance problems by avoiding renames. The `DirectOutputCommitter` [9] directly writes results to the object storage without a two-phase commit. However, this can lead to data loss when speculative execution is used [33]. The Stocator project [10] is an optimized object storage connector. It supports writing results directly and prevents race conditions by appending a unique suffix to the output of each output task. While this avoids renames, it changes the output file name, making it more difficult to retrieve the results of a job.

VII. Conclusion

We have presented SwiftAnalytics, an object storage system that serves as an efficient storage layer for data-parallel analytics frameworks such as Spark and MapReduce. SwiftAnalytics provides mechanisms to improve read and write performance by transparently chunking objects and providing placement control respectively. We showed that by using locality-aware writes, placement control, can be introduced efficiently without breaking the decentralized architecture of object storage systems. The two techniques help queries on SwiftAnalytics to achieve completion times close to their performance on HDFS.

VIII. Acknowledgments

The authors wish to thank the anonymous reviewers whose feedback and advice has helped improve this work. This work was in part supported by grant EP/K032968 ("NaaS: Network-as-a-Service in the Cloud") from the UK Engineering and Physical Sciences Research Council (EPSRC).

REFERENCES

[1] "Amazon S3," https://aws.amazon.com/s3/.

[2] "OpenStack Swift," http://swift.openstack.org/.

[3] "Cleversafe Object Storage," https://www.cleversafe.com/.

[4] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004.

[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *NSDI*, 2012.

[6] GCP Blog, "Easier, faster and lower cost Big Data processing with the Google Cloud Storage Connector for Hadoop," bit.ly/1RHQsY3, 2014.

[7] Netflix Tech Blog, "Hadoop Platform as a Service in the Cloud ," http://techblog.netflix.com/2013/01/hadoop-platform-as-service-in-cloud.html, 2013.

[8] "Apache Hadoop HDFS," bit.ly/1SeZUfS.

[9] Arnon Rotem-Gal-Oz, "The Bleeding Edge: Spark, Parquet and S3," http://bit.ly/2cO9Gd4, 2015.

[10] Gil Vernik, "Stocator – Fast Lane for Connecting Object Stores to Spark," http://bit.ly/1TNIUkY, 2016.

[11] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran, "Object Storage: The Future Building Block for Storage Systems," in *LGDI*, 2005.

[12] J. Chen, C. Douglas, M. Mutsuzaki, P. Quaid, R. Ramakrishnan, S. Rao, and R. Sears, "Walnut: A Unified Cloud Object Store," in *SIGMOD*, 2012.

[13] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang *et al.*, "f4: Facebook's Warm BLOB Storage System," in *OSDI*, 2014.

[14] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel *et al.*, "Finding a needle in haystack: Facebook's photo storage." in *OSDI*, 2010.

[15] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," in *STOC*, 1997.

[16] "HDFS Default Configuration," https://hadoop.apache.org/docs/r2.6.0/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml.

[17] L. Rupprecht, R. Zhang, and D. Hildebrand, "Big Data Analytics on Object Stores: A Performance Study," in *SC*, 2014.

[18] N. J. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman, "Skipnet: A Scalable Overlay Network with Practical Locality Properties," in *USITS*, 2003.

[19] "Hadoop OpenStack Support: Swift Object Store," bit.ly/1RHDNOF.

[20] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench Benchmark Suite: Characterization of the MapReduce-based Data Analysis," in *ICDEW*, 2010.

[21] "HIPI: Hadoop Image Processing Interface," http://hipi.cs.virginia.edu/.

[22] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR*, 2009.

[23] M. Poess, T. Rabl, H.-A. Jacobsen, and B. Caufield, "TPC-DI: The First Industry Benchmark for Data Integration," *PVLDB*, vol. 7, no. 13, 2014.

[24] "Statistical Workload Injector for MapReduce (SWIM)," https://github.com/SWIMProjectUCB/SWIM.

[25] 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 *SOSP*, 2007.

[26] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, "Rados: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters," in *PDSW*, 2007.

[27] W. Tantisiriroj, S. W. Son, S. Patil, S. J. Lang, G. Gibson, and R. B. Ross, "On the Duality of Data-intensive File System Design: Reconciling HDFS and PVFS," in *SC*, 2011.

[28] R. Ananthanarayanan, K. Gupta, P. Pandey, H. Pucha, P. Sarkar, M. Shah, and R. Tewari, "Cloud Analytics: Do We Really Need to Reinvent the Storage Stack?" in *HotCloud*, 2009.

[29] M. Mihailescu, G. Soundararajan, and C. Amza, "MixApart: Decoupled Analytics for Shared Storage Systems," in *FAST*, 2013.

[30] S. Zhou, G. R. Ganger, and P. A. Steenkiste, "Location-based node ids: Enabling explicit locality in dhts," Carnegie Mellon University, Tech. Rep., 2003.

[31] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data," in *SC*, 2006.

[32] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A Scalable, High-performance Distributed File System," in *SOSP*, 2006.

[33] SPARK-10063 JIRA, "Remove DirectParquetOutputCommitter," https://issues.apache.org/jira/browse/SPARK-10063, 2016.