acmqueue

Bringing arbitrary compute to authoritative data

Many disparate use cases can be satisfied with a single storage system.

Mark Savage and David Pacheco, Joyent

While the term *big data* is vague enough to have lost much of its meaning, today's storage systems are growing more quickly and managing more data than ever before. Consumer devices generate large numbers of photos, videos, and other large digital assets. Machines are rapidly catching up to humans in data generation through extensive recording of system logs and metrics, as well as applications such as video capture and genome sequencing. Large data sets are now commonplace, and people increasingly want to run sophisticated analyses on the data. In this article, *big data* refers to a corpus of data large enough to benefit significantly from parallel computation across a fleet of systems, where the efficient orchestration of the computation is itself a considerable challenge.

The first problem with operating on big data is maintaining the infrastructure to store it durably and ensure its availability for computation, which may range from analytic query access to direct access over HTTP. While there is no universal solution to the storage problem, managing a storage system of record (that is, one hosting the primary copy of data that must never be lost) typically falls to enterprise storage solutions such as SANs (storage area networks). These solutions do not typically offer WAN (wide area network) access, however, and they often require extra infrastructure to ingest data from and export data to arbitrary clients; this is hard to scale with the data footprint.

Once a system of record is established, computation on large data sets often requires an ETL (extract-transform-load) step into a system that can actually perform the computation. Even when the original data format suffices, NAS (network-attached storage)- and SAN-based systems do not support in-place computation, instead necessitating an expensive copy over the network—essentially a nonstarter for petabyte-size data sets.

Finally, there's an important distinction between special-purpose and general-purpose compute interfaces, meaning the abstraction differences between a SQL-like interface and a Unix shell. While the former is powerful and the appropriate choice for many kinds of analyses, it is not appropriate for many ad-hoc tasks such as video transcoding, compression, or other actions that work on unstructured or semi-structured data. Existing systems for distributed computation typically require users to understand a complex framework (abandoning tools they are familiar with) or use a special-purpose interface (which often raises the aforementioned data-copy problem each time a new interface is desired).

The goal of this article is to describe a general-purpose, distributed storage system that supports arbitrary computation on data at rest. It begins by detailing the constraints, which direct much of the design; then describes an implementation called Manta, which can be thought of as a reference implementation but is certainly not the only way such a system could be constructed; and ends with the ways in which several common big-data problems can be solved using Manta.

CONSTRAINTS

The design constraints can be broadly divided into storage abstraction (how users store and fetch data), the programming model (how users express in-situ computation on the data), and how the system provides durable local storage (how the system durably stores bits on disk).

STORAGE ABSTRACTION: AN OBJECT STORE

Building a system to scale means ensuring that capacity can be increased by adding more hardware. Scaling without downtime and without increasing the impact of individual component failure requires scaling horizontally, but Posix file-system and block-storage semantics are very difficult to scale horizontally. Updates may be very small (a byte within a file system; a block of a few kilobytes within a block device) and frequent. Even without considering multiple clients, the tradeoff between operation latency and durability is difficult: the more data the client is allowed to buffer before transmitting to the server, the faster operations may execute but the more data is at risk in the face of a crash. With multiple clients operating on the same file, a similar tradeoff exists between operation latency and consistency.

Object storage provides a more constrained model that's simpler to scale. An object store resembles a file system in that it has a global namespace with objects (data blobs) and directories (hierarchical collections), but object stores do not support partial updates. Users can create new objects and delete old ones, but they cannot update the contents of an existing object without replacing it entirely. For strong consistency, an object-store implementation need ensure only that the namespace is transactional; objects themselves are immutable, and thus easily replicated and cached.

PROGRAMMING MODEL

Distributed computing. The MapReduce programming model³ forms the foundation of several distributed computing systems—for good reason. Fundamentally, MapReduce divides a distributed computation into the parts that transform individual pieces of data (*map* operations) and the parts that operate on lots of data at once (*reduce* operations). Both types of operations are transformations of input data that produce a set of output data. The original data is immutable, so there are no side effects. This has several nice properties for distributed computing:

- Users need to express only the parts of a computation that are specific to their problem, leaving the problem of data flow up to the system itself.
- Map operations are fully parallelizable. In practice, you can add hardware resources to increase parallelism almost linearly.
- In the event of a failure, map and reduce operations can be retried. There is no complex state to unwind after a failure—only output to ignore. Even in the event of a network partition when the system may not know whether an operation has failed, it can execute the operation more than once as long as it ignores the results of all but one copy.

Local computing. MapReduce is a useful model for distributed computing, but the question remains, how do users specify what a map or reduce operation actually does? The preference is for an interface that users are already familiar with, that can leverage the enormous body of existing software, and that imposes as few constraints as possible. Given these priorities, there is a surprising (if obvious) solution in the Unix CLI (command-line interface).

At its core, the CLI provides a simple interface for arbitrary computing with a few extra primitives:

- File descriptors for reading and writing streams of input and output, along with conventions around the primary input, the primary output, and a stream for reporting other messages.
- Pipes, connecting the output of one program to the input of another, effectively composing a new program.

This approach encourages creating many small tools that compose well to run more sophisticated computations. This is perhaps best illustrated by an example from CACM's "Programming Pearls" series called "A Literate Program",¹ in which the following problem is posed: "given a text file and an integer k , print the k most common words in the file (and the number of their occurrences) in decreasing frequency."

The article describes two solutions: first, a custom solution whose concise presentation (with explanation) spans seven pages; and second, a shell one-liner "written on the spot" that "worked on the first try" with a six-stage pipeline using `tr`, `sort`, `uniq`, and `sed`. The code and explanation total about a quarter of a page. This cannot be overstated: the shell-based solution is much faster for the author to construct and for subsequent readers to understand, modify, and compose with other programs.

Ironically, the canonical MapReduce example from section 2.1 of the original paper suggests a very similar problem: "Count the number of occurrences of each word in a large collection of documents." Much is to be gained from extending the Unix interface to work in a MapReduce-based distributed computing model, but there are several challenges.

- **Programming interface.** MapReduce usually operates on keys or tuples. While many Unix tools do operate on field-separated records, this is just a convention. Instead of MapReduce being applied to a large, uniform set of keys, it can be applied to *objects* in the storage system. To work with objects in a familiar way, they can be exposed as read-only files on `stdin` (standard input), as well as on the file system.

With the MapReduce computation expressed as a Unix pipeline, programs must be able to interact with the system as they can on any other Unix system, having free rein in an isolated operating-system container. They must have their own process namespace, file system, networking stack, and any other hardware and software resources visible to programs.

- **Multitenancy.** A production storage system must support concurrent computations for both a single user and multiple users, and it should not require manually scheduling time on the system to ensure quality of service. Concurrent computations should share all available resources (possibly subject to administrative policies), and users should be allowed to program as if each is the only user on the system. They must not be able to see or interfere with one another.

- **Side effects.** Since Unix programs can read and write to the local file system, fork processes, make network connections, and so on, they decidedly do have side effects. This can be dealt with by saying that whatever a program does while it's running, the only way it can interact with the distributed computation is through its input and output streams. When the program completes, anything that would affect the environment for subsequent programs must be unwound, including file-system changes, processes created, and the like. This is achieved with operating system-based virtualization, using containers that provide isolation and resource controls and a file system that supports efficient rollback (discussed later).

DURABLE LOCAL STORAGE

The desire to expose objects directly as files to user programs leads to the most significant difference between the system described here and existing object-storage systems, which is that the entirety of each object must be stored on a single system. The object can be stored on multiple servers for additional availability and durability, but the entire contents must be present on each system in order to support our programming model. This in turn requires that the local file system store data durably. While that sounds simple, the reality of physical media means that the local storage system requires a number of features to manage storage effectively and ensure data durability.

Checksums outside data blocks. Data believed to be written to disk can turn out to be corrupted in many ways when it is later read back, including corruption at the disk controller, in cables, or in drive firmware; accidental writes directly to the disk; environmental effects resulting in changes to bits on disk after they were written correctly (bit rot); bugs in firmware resulting in dropped writes (phantom writes) or misdirected writes; and so on. While block checksums protect against bit rot, they cannot detect a well-formed block written to the wrong place (as may happen with accidental overwrites, misdirected writes, or phantom writes). The file system must checksum data and store checksums separately from data blocks in order to detect that a given block is not only valid but also represents the data that's expected in that block.

Double-parity (or better) software-based RAID. To get appropriate durability for each copy of an object at a reasonable cost, each node's storage should itself be redundant. A single spindle failure should not result in reduced availability or durability for objects on that server. To reconstruct bad copies of data blocks (detected with the just-described file system-managed checksums), the file system must be involved in the data reconstruction process, which implies software-based RAID (redundant array of independent disks).

Copy-on-write. Copy-on-write file systems support $O(1)$ point-in-time snapshots and efficient rollback to previous snapshots. This is a critical feature for unwinding changes made by user programs so that other users' programs can run later in the same container.

Pooled storage. Storage servers must provide temporary scratch space for user programs. Isolation demands that users get their own file systems, and the desire to support many tenants with flexible resource controls demands that these file systems be resized dynamically. As such, file systems should not be laid out directly on disks but rather should be flexibly allocated from a pool of storage.

Separate intent log. The economics of a large-scale storage system demands traditional disks for primary storage, but write performance can be improved significantly with SSDs (solid-state drives) as intent log devices. Streaming writes can leverage the considerable throughput of a large pool of spindles while synchronous writes (usually smaller) can be committed to the SSD and acked to the client before they make it to the spindles.

ADDITIONAL DESIGN GOALS

In addition to these hard requirements, the system needs several properties for consistency, availability, and durability.

CAP

The CAP theorem² specifies that in the face of network *partitions* (which will always happen), data services must choose between *consistency* and *availability*. The trend in distributed systems has been

skewed toward availability in recent years, and while availability (eventual consistency) is often the right choice for an application, choosing a weakly consistent model for a general-purpose storage system means that no application built on top of that storage system can ever be strongly consistent. Conversely, if a low-level storage system is strongly consistent, an application can always take measures to choose availability by implementing asynchronous writes (staging) and caching reads. For these reasons, a storage system of record must be strongly consistent.

AVAILABILITY AND FAILURE MODES

Choosing strong consistency does not mean that a system should abandon high availability. The system should survive failure of any single component, including an individual service instance, a physical server, a rack, or an entire data center. Every component must be replicated and scaled out across three or more data centers. In the classic $2F+1$ failure model,⁶ the system should support continuing operations with a single data center offline. (There may be a short reconvergence time, during which writes would be rejected, but afterward the system will continue operating without degradation.)

DATA ACCESS AND DURABILITY

Data should be exposed over a REST (representational state transfer) API. When the system returns HTTP/1.1 200 OK after a write, copies must exist on two physical servers, *and* copies of the metadata about the actual data must exist on two physical servers. These copies must also span data centers. As mentioned earlier, each physical copy must be resilient to individual drive failures.

IMPLEMENTATION

Manta (figure 1) is Joyent's implementation of the storage system whose design goals have just been described. While the architecture mostly falls straight out of the constraints and goals described, the implementation deals with several important details.

STORAGE

At a high level, users interact with a scalable Web API that is stateless and load balanced. Metadata about objects is stored in an application-partitioned database tier, and objects themselves are written synchronously to two or more storage nodes across the fleet.

FRONT END

To maintain high availability, most Manta components are stateless. The front end consists of a group of load balancers using HAProxy and custom API servers written in Node.js. Users' TLS (Transport Layer Security) sessions are terminated at the load balancers, and HTTP requests are handled by the API servers. There's also a read-only authentication/authorization cache backed by Redis. Every data center has at least one instance of each component, and load balancing across the load balancers themselves is handled coarsely using round-robin DNS. Since all of these components are stateless, new instances can be provisioned for additional availability or capacity.

Write requests are logically broken down into the following steps:

1. Ensure that the parent path (directory) exists.
2. Select a set of N storage nodes on which to store data. By default, $N = 2$, but users can specify more

FIGURE 1

Manta Services

or fewer copies. Each copy is highly durable (as already described), but copies are stored in separate data centers, so having more copies results in increased availability.

3. Synchronously stream data to the selected storage nodes.

4. Record durable metadata (indexed by the object's user-visible name) indicating which storage nodes contain the data.

Read requests are broken down into a similar series of steps:

1. Look up the requested object's metadata, which includes the set of storage nodes that have a copy of the object.

2. In parallel, contact each storage node hosting a copy of the data.

3. Stream data from the first node that responds.

Deletes and overwrites are processed by a background garbage-collection system. When a delete (or overwrite) request comes in, Manta records the event in a durable transaction log. The garbage-collection system later determines which of these objects are no longer referenced by the metadata tier and reclaims the corresponding space on the storage nodes.

METADATA

According to the design constraints, object metadata must be stored durably and modified transactionally, but still highly available and horizontally scalable. Manta divides metadata storage and access into three components:

1. A data-storage and replication engine.
2. A key/value interface that fronts (1).
3. A sharding layer for (2) to support horizontal scalability.

Data Storage and Replication. Since the metadata is highly structured, a classic B-tree representation is sufficient to support transactional updates and fast queries on indexed properties. To maintain availability, the system must also support synchronous replication to a secondary instance. After debating whether to build our own or use an off-the-shelf solution, we decided to use PostgreSQL, whose storage subsystem is proven and whose replication system satisfies these requirements. What PostgreSQL lacks is a built-in system to ensure automatic failover using leader election. We wrote software that leverages a consensus layer (Zookeeper) to manage a daisy chain of replicas. When a PostgreSQL peer crashes or exits the fleet, the peer that was directly behind it is promoted in the topology:

Master -> Synchronous Peer -> Async -> Async -> ...

Further, the topology is recorded in a well-known place by the leader, and the key/value layer (through which all clients access the database) knows how to use this information.

Moray: Key/Value Interface. On top of PostgreSQL is a thin key/value interface called Moray, which provides three functions:

- Knowledge of the underlying database-replication topology to ensure that writes are always directed to the PostgreSQL master.
- A simple interface without explicit transactions that supports optimistic concurrency control.
- A good-enough query interface using LDAP (Lightweight Directory Access Protocol) search filters.

For programmer simplicity, a PUT/GET/DELETE paradigm makes the key/value interface abstractions *buckets*, *keys*, and *objects*, where objects are always JSON (JavaScript Object Notation) documents. Indexing rules are defined for each bucket such that search requests are able to look for objects that match a filter evaluated on the JSON objects. Simple lookups are done directly by the bucket/key pair.

The following Node.js code demonstrates both the API and semantics:

```
// Create a bucket named "user_directory", and ensure that any objects
// written to it are inspected for the attributes "id", "name", and "email";
// if any are present, maintain an index on those attributes for future
// search requests
moray.putBucket('user_directory', {
  index: {
    id: 'number',
    name: 'string',
    email: 'string'
```



```

    }
  });

  // Write a sample user object to "user_directory" with the key
  // "mcavage" and value where there are both indexed and non-indexed
  // fields
  moray.putObject('user_directory', 'mcavage', {
    id: 123,
    name: 'Mark Cavage',
    email: 'mark.cavage@joyent.com',
    department: "engineering"
  });

  // Search for any objects that have an email address of "@joyent.com"
  var res = moray.search('user_directory', '(email=*@joyent.com)');
  res.on('record', function (obj) {
    console.log('Record found: %s', JSON.stringify(obj));
  });

```

This system is the only interface that other components in Manta use to store state. All metadata about user objects, storage node utilization, and compute job state is stored in PostgreSQL through Moray.

Electric Moray: Key/Value Interface. A collection of at least three replicated PostgreSQL instances plus one or more Moray instances is called a *shard*. At any given time, only one of the PostgreSQL instances in a shard is used for reads and writes, and the Moray instances direct all key-value operations to that instance. Durability and availability are provided within each shard, but in order to scale horizontally (i.e., to increase the storage footprint or read/write capacity of this system beyond what a single instance can provide), Manta uses several shards and performs application-level partitioning among them using a consistent hashing scheme.⁵

Recall that the user-facing API consists of directories and objects that behave largely like Posix directories and files. As with a Posix file system, in order to write an object, all parent directories must exist. The following CLI commands should look familiar:

```

$ mmkdir /mark/stor/foo/
$ echo "Hello, ACM" | mput /mark/stor/foo/bar.txt

```

To determine which shard holds the metadata for a given object, Manta performs a consistent hash operation on the directory name of the object. For example, given `/mark/stor/foo/bar.txt`, Manta hashes the string `/mark/stor/foo`. Metadata for all objects in a directory exists on a single shard.

The system that manages the mapping between object names and shards is called Electric Moray. It offers the same interface as Moray (and ultimately directs all requests to a Moray instance) but deals with routing requests to the right shard.

STORAGE SERVER

Storage nodes are primarily responsible for maintaining the durability of a single copy of an object and providing the isolated compute containers in which to run user programs. For storage, these nodes provide a straightforward HTTP interface over the local disks, which are managed by the ZFS file system. Objects are written using a random (UUIDv4) name. On a write, a successful status code is returned to the calling operation only when `fsync(2)` has completed and the data is known to be safely on disk.

Manta leverages ZFS to solve the local durability problem. This satisfies all of the constraints described previously, including checksums to detect all forms of block-level corruption, software RAID that automatically detects and repairs such corruption as data is read, and pooled storage to support dynamically resizing file systems to satisfy user requests for additional scratch space. The triple-parity software RAID implementation can sustain steady-state operation (including writes) in the face of up to three concurrent disk failures, though systems are configured with double parity for better cost efficiency.

COMPUTE

Users run computation in Manta by submitting *jobs*. To support a MapReduce-like model, jobs are made up of one or more *phases*, each of which is either a map phase or a reduce phase. Map phases are divided into *tasks*—one for each input object. Users can specify how many reduce tasks they want for a given reduce phase.

An important design goal is that jobs should be able to operate on an arbitrary number of objects, with an arbitrary number of tasks, producing an arbitrary number of errors or outputs, meaning that a quantity should scale with the resources of the system rather than being limited for algorithmic reasons. To achieve this, the interfaces to list inputs, outputs, and errors are necessarily streaming and paginated, rather than providing complete listings in a single request. Internally, inputs, outputs, errors, and tasks are operated on in groups, and it must never be required for all of them (or any particular group of them) to be stored in memory.

As a concrete example, here's a map-only job that searches for instances of a given URL (`/login`) in a set of Apache-format request logs named in the local file `inputs.txt`. A user would run this from the local system:

```
$ mjob create -o -m 'grep -w /login' < inputs.txt
added 240 inputs to 4e55b83b-a04b-c25c-9169-d4b4641d915d
10.10.0.34 - - [14/Apr/2014:11:07:48 +1000] "GET /login HTTP/1.0" 200 6433 "-"
"nodejs"
...
```

The `mjob create` client command runs a new Manta job. The `-m` option specifies a map phase defined by the following script. The `-o` option causes the command to wait until the job completes and then print the contents of the job's outputs. This is the closest analog to running a command locally in the shell, except that it operates in parallel and the `grep` actually runs on the storage server. The `inputs.txt` file contains a list of objects to run the job on. Since this is a map job, the

system will run `grep -w /login` for each object in `inputs.txt`, automatically parallelizing as best it can.

In the API, the job looks like this:

```
{
  "id": "ea5290f9-4f1b-e52c-9c47-ffe9fc6f7905",
  "state": "running",
  "inputDone": true,
  "stats": {
    "errors": 0,
    "outputs": 0,
    "retries": 0,
    "tasks": 240,
    "tasksDone": 0
  },
  "phases": [ {
    "exec": "grep -w /login",
    "type": "map"
  } ],
  ...
}
```

When it finishes, `state` will be `done`, and `tasksDone` will equal `tasks`.

Executing this job is done in two parts: the distributed orchestration of the job and the execution of the user's `grep` script on each object.

Job orchestration. When the user runs `mjob create`, the Manta Web service receives client requests to create the job, to add 240 inputs to the job, and to indicate that there will be no more job inputs. The job is assigned to a particular job supervisor, which is responsible for coordinating the distributed execution of the job. As inputs are added to the job, the supervisor resolves each object's name to the internal uuid (universally unique identifier) that identifies the object and checks whether the user is allowed to access that object. Assuming the user is authorized for that object, the supervisor locates all copies of the object in the fleet, selects one, and issues a task to an *agent* running on the server where that copy is stored. This process is repeated for each input object, distributing work across the fleet.

The agent on the storage server accepts the task and runs the user's script in an isolated compute container. It records any outputs emitted as part of executing the script. When the task has finished running, the agent marks it completed. (This process is described in more detail later.)

The supervisor *commits* the completed task, marking its outputs as final job outputs. When there are no more unprocessed inputs and no uncommitted tasks, the supervisor declares the job done.

If a task fails, it will be retried a few times, preferably on different servers. If it keeps failing, an error is produced.

Multiphase map jobs are similar except that the outputs of each first-phase map task become inputs to a new second-phase map task, and only the outputs of the second phase become outputs of the job.

Reduce tasks. Reducers run as mappers, except that the input for a reducer is not completely known until the previous phase has already completed. Also, reducers can read an arbitrary number of inputs so the inputs themselves are dispatched as individual records, and a separate end-of-input must be issued before the reducer can complete.

Suppose that you want to modify the previous example to report the number of times each IP address requested `/sample.txt`. You could change the map phase to emit only the column of IP addresses and have the reduce phase count the number of distinct values:

```
$ mjob create -o \
  -m 'grep -w /sample.txt | cut -d" " -f 1' \
  -r 'sort | uniq -c' < inputs.txt
added 240 inputs to 4e55b83b-a04b-c25c-9169-d4b4641d915d
135 10.10.0.34
  5 10.10.0.37
 23 10.10.0.33
...
```

Notice that the two parts of this problem are exactly what you would run to solve the same problem on a single system using the shell. The distributed version could actually apply the reducer with the mapper, and replace this reducer with an `awk` script that sums the values per column, but this performance optimization is often not necessary.

For this two-phase job, the first phase works as just described. For the reduce phase, a single reduce task is issued to an agent somewhere in the fleet when the job is created. As map tasks complete, their outputs (themselves temporary Manta objects) are marked as inputs to the reducer. The reducer emits a single output object, then gets marked completed. As before, the supervisor commits the result, producing the job's sole output object, and the job is marked done.

Local execution. The agent on each storage server maintains a fixed set of compute zones in which user scripts can be run. When a map task arrives, the agent locates the file representing the input object on the local file system, finds a free compute zone, maps the object into the local file system, and runs the user's script, redirecting `stdin` from the input file and `stdout` to a local file. When the script exits, assuming it succeeds, the output file is saved as an object in the object store, recorded as an output from the task, and the task is marked completed. If there is more work to be done for the same job, the agent may choose to run it in the same compute zone without cleaning up after the first task. When there is no more work to do, or the agent decides to repurpose the compute zone for another job, the compute zone is halted, the file system rolled back to its pristine state, and the zone is booted again to run the next task. Since the compute zones are isolated from one another and are fully rebooted and rolled back between jobs, there is no way for users' jobs to see or interfere with other jobs running in the system.

Internal communication within the compute framework. The system uses the sharded, replicated database component (Moray/PostgreSQL shards) to manage job state. There are buckets for jobs, job inputs, tasks, task inputs (for reduce tasks), task outputs, and errors. Supervisors and agents poll for new records applicable to them. For example, supervisors poll for tasks assigned to them that have been completed but not committed, and agents poll for tasks assigned to them that have

been dispatched but not accepted. When a job completes, the lists of inputs, outputs, and errors are archived into Manta objects, and the corresponding database records are removed to limit the database to only current working state. The poll-based approach has obvious performance downsides, but the implementation is relatively easy to ensure correct, and the performance can be greatly improved without changing the basic design (e.g., by reducing poll intervals and using push notification to trigger an immediate poll).

While this mechanism has proven reliable, it is also responsible for most of the system's latency from job input to the corresponding output (two to three seconds for a 500-ms user program). The workload has also triggered pathological performance issues in the PostgreSQL layer, leading to brownout situations where jobs that normally take a few seconds to complete could take minutes. Now that they are understood, these pathologies can generally be avoided or mitigated.

EXAMPLE: LOG ANALYSIS FOR BANDWIDTH AND COMPUTE METERING

Manta itself provides an example of a common use case for big-data systems in the form of continuous log analysis. Per-user metrics for bandwidth and compute usage are computed using MapReduce jobs operating over internal log files, which are themselves pushed into Manta every hour. These jobs take the internal log files as inputs, parse the log entries, count metrics such as bandwidth and compute time used, and produce per-user, per-hour access logs, as well as aggregated usage reports that are also used for billing. The reduce process is spread over two phases for increased parallelism. Having this critical use case in mind while building Manta greatly informed the design of the system.

EXAMPLE: CUSTOMER COHORT RETENTION

One customer has presented extensively on a data-analytics system built on top of Manta.⁴ It uses rsyslog to log user actions daily from more than 40 front-end Web servers. These logs, representing about 5 to 20 million events per day in ASCII, pipe-delimited files totaling about 1 GB, are uploaded to Manta. Typical business questions include “How many unique users performed a given action from an iPad on a given date?” and “How many people who signed up four weeks ago are still active?” Because of their simple format, these questions are answered with 5- to 10-line scripts using only `grep(1)`, `awk(1)`, `sort(1)`, `uniq(1)`, `comm(1)`, and `wc(1)`. Results are generally available in seconds to one minute.

EXAMPLE: VIDEO ANALYSIS AND TRANSCODING

A number of customers have used Manta to transcode videos. We've also been doing this internally for a side project analyzing screen captures of Mario Kart 64 games (<http://kartlytics.com/>). The program to analyze videos was built on top of the FFmpeg suite of tools and libraries for video processing. The application is a typical MapReduce job that maps over video files and produces JSON files describing what happened in the video. The resulting files are combined to produce a single summary file, which presents all the results on a Web site. A separate map-only job transcodes the original, high-quality video files into much smaller, Web-quality WebM files. The entire corpus is analyzed in 10 minutes (fully parallelized across the Manta fleet), compared with the better part of a day on a single system, without having to modify the analysis software itself.

CONCLUSION

Unifying the storage system of record with in-situ computation and adapting the ubiquitous Unix shell environment to distributed computing has resulted in many disparate use cases being satisfied with a single storage system. These include traditional log analysis, as well as asset hosting (as for a content-delivery network), image manipulation (e.g., on the same assets hosted for the Web), and video transcoding. Achieving this required unconventional design choices such as storing objects directly as raw files rather than erasure-encoded across multiple servers. While there are undoubtedly other ways to build such a system, they would necessarily require reliable local storage and mature operating system-based virtualization. While previous systems have either leveraged enterprise-grade *network* storage (which was viewed as a nonstarter because it requires moving data in order to compute on it) or built custom storage systems to work around otherwise flaky or inefficient local storage, using an architecture based on ZFS allowed us to focus on the distributed systems part of the problem and (we believe) produce a more general-purpose system.

There's a great deal of additional work to be done, including richer access controls, support for more software packages out of the box, and improved performance. But Joyent and its customers have already found the system valuable for analyzing both business and technical data.

REFERENCES

1. Bentley, J., Knuth, D., McIlroy, D. 1986. Programming Pearls: a literate program. *Communications of the ACM* 29(6): 471-483; <http://dl.acm.org/citation.cfm?id=315654>.
2. Brewer, E. A. 2000. Toward robust distributed systems. Keynote speech. Nineteenth ACM Symposium on Principles of Distributed Computing; <http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf>.
3. Dean, J., Ghemawat, S. 2004. MapReduce: simplified data processing on large clusters. Sixth Symposium on Operating System Design and Implementation; <http://research.google.com/archive/mapreduce.html>.
4. Gredeskoul, K. 2013. Using Manta to scale event-based data collection and analysis; <http://www.slideshare.net/kigster/ss-26329742>.
5. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. Published in *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*: 654-663; <http://dl.acm.org/citation.cfm?id=258660>.
6. Lamport, L. 2006. Lower bounds for asynchronous consensus. Microsoft Research; <http://research.microsoft.com/en-us/um/people/lamport/pubs/lower-bound.pdf>.

SEE ALSO

Fontaine, T. J. 2013. 550 regression tests in 4 minutes with Joyent Manta; <http://www.joyent.com/blog/550-regression-tests-in-4-minutes-with-joyent-manta>.
 Robbins, C. 2013. Keeping the npm registry awesome; <http://blog.nodejs.org/2013/11/26/npm-post-mortem/>.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

MARK CAVAGE is Joyent's vice president of engineering, where he oversees development of the company's core technologies: Node.js, SmartOS, SmartDataCenter, and Manta. He was previously a senior software engineer with Amazon Web Services, where he led the Identity and Access Management team.

DAVID PACHECO is a software engineer at Joyent, where he works primarily on the Manta storage service. He also works on Joyent's SmartDataCenter product and tools for tracing and debugging Node.js programs in production.

© 2014 ACM 1542-7730/14/0600 \$10.00