Distributed Computation on Dynamo-style Distributed Storage: Riak Pipe

Bryan Fink
Basho Technologies
bryan@basho.com

Abstract

The Dynamo model, as described by Amazon in 2007, has become a popular concept in the development of distributed storage systems. The model accounts for only CRUD operations, however. This paper describes a system called Riak Pipe that enables the use of generic functions in place of CRUD operations. This allows Dynamo-model users to exploit other resources, such as CPU time, available in their cluster, as well as to gain the efficiencies offered by data-local processing.

Categories and Subject Descriptors H.3.4 [*Information Storage and Retrieval*]: Systems and Software; D.1.3 [*Programming Techniques*]: Concurrent Programming; H.2.4 [*Database Management*]: Systems; D.2.11 [*Software Engineering*]: Software Architectures; H.2.3 [*Database Management*]: Languages

General Terms Design, Reliability

Keywords Dynamo, Erlang, Riak, MapReduce

1. Introduction

The Dynamo model, as described by Amazon in 2007[1], has become a popular concept in the development of distributed storage systems. The model accounts for only those operations necessary for storing and retrieving key-value pairs, the standard CRUD (create, read, update, delete) operations, however.

This paper introduces a model for allowing additional, generic operations to be distributed for processing on a Dynamo-style system. The system described, Riak Pipe, pairs generic processing functions with Dynamo's consistent hash functions, and furthermore provides a framework for chaining these generic functions into staged processing pipelines.

While use of a Dynamo-style key-value store in concert with Riak Pipe is not necessary, this paper also describes how doing so allows exploitation of its design to enable data-local processing. This technique is not only an improvement on the key-value store's capabilities, but it can also lower network utilization for many common queries.

Readers of this paper will find it useful to be somewhat familiar with the concepts described in the Dynamo paper. A review of the relevant Dynamo concepts is given in section 2.1.

Erlang'12, September 14, 2012, Copenhagen, Denmark.

2. Background

Riak Pipe is built atop an engine called Riak Core, which implements some of the basic structures needed to create a Dynamo-style storage system. Riak Pipe is designed to live on this system, next to a key-value store known as Riak or Riak KV. The following two sub-sections review the concepts of Dynamo that are relevant to Riak Pipe, and also discuss the use case in the Riak key-value store that drove Riak Pipe's creation.

2.1 Dynamo Review

The most relevant Dynamo topics for this paper are those of consistent hashing, data distribution via hash-space partitioning, and tunable replication management.

As a review, the Dynamo paper describes a distributed key-value store where not all members of the cluster maintain a copy of every key-value pair. The decision of which cluster members maintain a copy of a given key-value pair is made primarily via consistent hashing. At storage or lookup time, the hash of the key being fetched is compared to a mapping of hash space partitions to cluster members. As members are added to the cluster, this mapping is altered to shift responsibility for some portions of the stored data to the new members.

The number of additional copies is determined by a configurable value, named *N* in the paper. The cluster members that store the additional copies are chosen according to the same hash-space map as the first copy. The additional *N*-1 are simply stored on the owners of the successive partitions.

Dynamo's purpose for this model is to provide low-latency, high-availability access. Multiple cluster members are able to answer requests for each key, so slow responses from a subset of a key's *N* replicas need not affect performance, and even absent responses need not mean missing data.

2.2 Riak Key-Value Store

The Riak database[10] is a distributed key-value data storage system, built on the model described in *Dynamo: Amazon's Highly Available Key-value Store*[1]. It is written in Erlang, and has been in production use since 2008. It was open-sourced in 2009.

When developing applications that stored their data in Riak, we found that a common access pattern was to store a serialized object, containing many fields, under one key. Later, when the key was read from storage, many times it was to access just one field of that object. Furthermore, it was often the case that the value of that field was the key of another object that was then fetched immediately.

This access and reference pattern led to the development of a feature in Riak called "links". This included a standard representation compatible with Mark Nottingham's *Web Linking*[11] proposal, as well as a method for traversing several hops between a series of objects.

We quickly found that traversing many links could be done more efficiently if we told the node holding each object what type of links a query was interested in, and let it extract and filter them, rather than pulling each object across the network to a central location for processing. Not only did this lower network traffic by exploiting data locality for processing, but it also took advantage of the distributed resources to parallelize the processing.

Sending link query details to Riak nodes for processing was an eye-opener, and we very quickly shifted to a mode where any query, in the form of a function, could be sent to a Riak node for processing local data. After years of improvement driven by the realities of production deployment, it has been abstracted into a form usable either with or without attachment to the key-value store. This project is now called Riak Pipe.

3. Riak Pipe

Riak Pipe provides an abstraction for distributed computing on a Dynamo-style system, based on the simple concept of staged processing chains. Much like the familiar pipes that string together UNIX commands, Riak Pipe strings together processing functions. Each function constitutes one stage of the pipeline.

For example, consider a pipeline constructed with the processing functions *F* and *G*. Inputs to this pipeline are processed by function *F*. For each input, *F* may produce zero or more outputs. Each output of *F* is processed by function *G*. As before, *G* may also produce zero or more outputs for each input. Since *G* is the end of the pipeline, its outputs are delivered to a process that is designated as the "sink" when the pipe is constructed.

The key to distributing this processing around the cluster is the "consistent hashing" concept of the Dynamo paper. Each stage of the pipeline, in addition to its processing function, also provides a hashing function. This hashing function is applied to each input for the stage, and its result determines what node in the cluster evaluates the processing function for the input.

That is, our previous example pipeline would not be a simple list of *F* and *G*, but would instead be a list of pairs, (*F-processing*, *F-hashing*) and (*G-processing*, *G-hashing*). Before applying any of the *-processing* functions to an input, the paired *-hashing* function is evaluated. The result of the *-hashing* function determines which node will perform the evaluation of the *-processing* function for that input, by comparing it against the same map used for consistent-hashing in the key-value store.

The pair of processing function and hashing function are together known as a *fitting* in Riak Pipe.

The following sections will describe Riak Pipe in more detail by explaining, respectively, how the system is structured, how that structure enables resource limitation to prevent unpredictable degradation in over-capacity situations, the effect of hashing functions on processing function utility, how cluster membership changes affect the running system, what happens in failure cases, and how the progress of pipes can be observed through logging and tracing.

3.1 Structure

Riak Pipe is structured as all Riak Core applications are: as a cooperating set of "vnodes" (short for "virtual node"). Each vnode is responsible for one partition of the consistent hashing space. When new machines are added to the cluster, they may claim one or more of those partitions, and then set up a vnode for each of their claims. Each vnode is an Erlang process, implemented as an OTP gen_server. Implementing the vnodes in this fashion makes it extremely simple for any piece of the system (client processes, other vnodes, etc.) to communicate with any vnode, directly, through simple message passing.

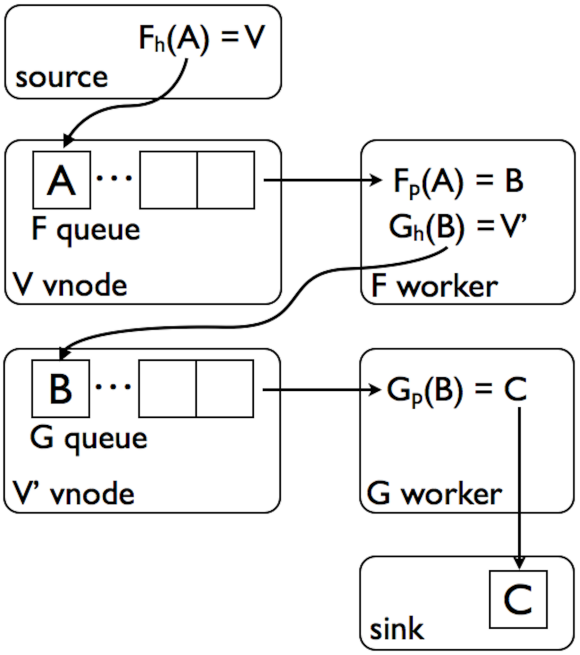

Figure 1. Sample flow of input *A* through fittings *F* and *G*, whose hash functions send their inputs to separate vnodes. Fh(A)=V represents the application of *F-hashing* to input *A*, producing a hash that maps to vnode *V*. Fp(A)=B represents the application of *F-processing* to input *A*, producing output *B*.

The main role of a Riak Pipe vnode is to maintain queues of inputs for stages of running pipelines. In comparison to the SEDA system, a Riak Pipe vnode is something like the "controller" managing the "event queue".[2]

The Riak Pipe vnode is also in charge of monitoring the worker processes that consume those queues. Each time an input is received for a new stage, the vnode creates a queue for it and also spawns a new worker process. The worker process is where the processing function is actually evaluated. As with vnodes, worker processes are simply Erlang processes, though they use the OTP gen_fsm behavior instead.

To return to our example from earlier, let's look at the process of sending input *A* through the pipeline of fittings (*F-processing*, *F-hashing*) and (*G-processing*, *G-hashing*). See figure 1.

First, apply *F-hashing* to *A* to produce *A-hash*. That *A-hash*, by means of the hash-partition map, will point to a vnode *V* running on some node *N*. Riak Pipe will thus send input *A* to vnode *V*. Vnode *V* will put *A* in a queue, and then start a worker for fitting *F* to consume that queue.

Next, we'll say that the worker applying *F-processing* to input *A* produces output *B*. The worker will apply the hashing function for the next fitting, *G-hashing*, and thereby learn which vnode should handle applying *G-processing* to input *B*. The worker sends *B* to that vnode, which does as before, queues the input and starts a worker to handle the processing.

Output from fitting *G* is then delivered to the sink process, which is normally a process separate from the vnodes and workers, meant solely for consuming the outputs of the pipeline.

This series of transfers of inputs between processes was made extremely simple by using Erlang for the implementation. Inter-process messages are a built-in primitive in Erlang, and they are structured in such a way that they have the same form regardless

of whether the processes are on the same machine. This means that workers do not need to have any special logic for sending to remote versus local vnodes, greatly simplifying the code.

Erlang also made transfering references to the processing and hashing functions easy. What other languages wrap in "reflection" framework, Erlang exposes via simply using tuples and atoms that may be either programmatically created or typed literally into program code or a read-eval-print loop. Indeed, sometimes it's too easy: nodes are happy to pass these references between each other, even if they have missing or outdated versions of the code. This can lead to surprising failures, if not kept in check.

3.2 Backpressure Via Blocking Queues

The implementation of resource limiting, queue size management, is inseparable from the design of Riak Pipe, since the necessity for it was a primary design requirement. The Riak key-value store for which Riak Pipe was designed prides itself on maintaining predictable low latencies, even under heavy loads. Maintaining that predictability was of top priority.

The process of adding an input to one of the queues maintained by a vnode involves waiting for an acknowledgement that the enqueueing succeeded. This means that workers are required to operate no faster than the vnodes can accept their outputs.

In addition, each of the queues is bounded in size. If adding an item to a queue would push it over that bound, the item is instead held in a separate "waiting" queue. The acknowledgement for receipt for that input is then withheld until there is room for the input in the working queue. This means that workers cannot overwhelm their downstream counterparts.

This blocking behavior is important, because many pipelines begin with relatively trivial stages but end with relatively complex stages. In these situations, it's common for the upstream stages to outpace those downstream. This is especially true when the complexity of later stages is multiplied by the need to constrain the work to just one node, as we shall see later. Without backpressure to slow upstream processing, this outpacing manifests itself as escalating memory growth as unprocessed inputs pile up for the downstream, slower stages.

Since the number of nodes processing inputs for any given stage is bounded (because no vnode starts more than one worker for any stage), backpressure could eventually pause all processing at that stage. This can eventually exert pressure all the way to the head of the pipeline, if downstream processing is slow enough.

Slowing upstream stages on the surface seems like a negative for a given pipeline's performance, but it is actually a good thing when observed from a whole-system level. When multiple pipelines are sharing time on the same cluster, constraining the memory growth of input queues ensures fairer access to the system. Pausing stages that are producing output that cannot possibly be consumed in the near-term also frees up other resources, such as CPU and I/O bandwidth for processing inputs that are already waiting.

An alternate method of dealing with overflow is also available in Riak Pipe. Workers may specify at enqueue time whether the above blocking behavior should be used, or if instead inputs should simply be discarded when queues are over their thresholds. Instead of acknowledgements of receipt, errors indicating the inputs have been discarded are returned to the worker. This method is not often used in production pipelines today, except in some failure cases where a worker re-enqueues an input for its own stage (a circular, or recursive enqueueing). In this case, blocking while waiting for space in a queue could lead to deadlock where the worker ends up, either directly or indirectly, waiting on itself to clear space in the queue.

Implementing this behavior is another place where Erlang shines. In addition to being able to send messages to any other process, a process may easily put itself into a blocking state that will only end when either the reply it is waiting for arrives (via a facility known as "selective receive"), or it receives notification that the process it is waiting on has exited and will never deliver the reply (via a facility known as "monitors"). While the process is in this waiting state, it is unscheduled and consumes no processing time until it receives a message.

3.3 The Effect of Hash Function Choice

The choice of a consistent hashing function is very important for maximizing a processing function's potential. It is the tool by which data-locality is exploited, and by which parallelism is controlled.

For instance, a common consistent hash function is the SHA-1 function. This function has the effect of relatively evenly spreading inputs across the cluster, which can be appropriate for consuming as many parallel resources as possible. The opposite is also useful: a constant function, which produces the same output regardless of input, will send all data to a single place for processing. Such behavior is useful for operations like computing a sum total.

The only common aspect of these functions is that they are purely functional. They produce no side-effects, and they generate the same hash for the same input, regardless of time or location of evaluation. This is important, as the hashing function could be re-evaluated many times on different machines but must always point to the same vnode for processing an input.

Some specific pairings of processing and hash functions will be desribed in section 5.

3.4 Dynamic Cluster Membership

Using hash functions to decide where to process data has the same effect it does in a Dynamo datastore: it allows cluster membership to be dynamic, and to automatically rebalance load when nodes are added, removed, or temporarily unreachable. The hash function chooses the logical place to process the input, by pointing to a portion of the hash space. Other cluster machinery maps that choice to a physical machine, by tracking which machine has claimed each hash space portion.

Many processing functions are stateless. For example, the "get" and "transform" fittings discussed in section 5 need not track extra information between each input. Migrating their work from one machine to another is therefore trivial: stop processing in the old place, and start processing in the new place.

Other functions, such as the "reduce" fitting (also discussed in section 5), accumulate extra information that persists between inputs. Migrating these functions is slightly trickier, in that it requires two extra functions (beyond the processing and hashing functions) to be associated with the fitting. The first function, named *archive*, serializes the state at the old node into something suitable for transfering across the network. The second function, named *handoff* – after the hinted-handoff Dynamo mechanism it enables – accepts the archive data from another node and merges it into the state of the new node.

This is one area where purely-functional code shines. If a fitting's processing function is purely functional, as is a simple implementation of "reduce", then the archive function need only return the last result of the processing function's evaluation (the accumulator in the fold definition). Erlang also aids in making this easy by automatically encoding and decoding any term for sending messages across the network; no special serialization code is needed. The handoff function can be similarly simple if pure functions are used. For example, for the "reduce" fitting, the handoff function may be able to simply evaluate the processing function as if the archive were just another input.

As mentioned briefly earlier, this archive-and-handoff mechanism is used not only when adding or removing machines from a

cluster, but also when machines are temporarily unreachable. While a machine is unreachable, an alternate machine will be chosen to process the inputs it would have received. When the intended machine is once again reachable, processing can shift back to it via archive-and-handoff.

3.5 Failure Scenarios

When a cluster node is unreachable or encounters errors, an alternate node will be chosen to process the inputs normally destined for it. The method for choosing the alternate node is primarily driven by the selection of adjacent hash-space claims.

The variable *N* is used in the description of Dynamo-style storage systems to denote the number of replicas that should be made of a given value. The replicas are stored on the nodes that have claimed the subsequent adjacent portions of the hash space to which the key for the object originally hashed.

Riak Pipe also uses *N* to map inputs to nodes in a similar manner. It will attempt to process each input on the node claiming the hash space indicated by the hash function, but if that node is unavailable or processing fails, processing will be attempted on the node claiming the next adjacent hash space. Retries will continue down the line until processing succeeds or all *N* attempts have failed.

This use of *N* has obvious applications for fittings that read replicated data from an associated key-value store. For fittings that do not deal with replicated data, *N*'s value is less clear, but it has been useful for avoiding hot spots when resources are tight. For example, should processing require making a request of an external program, one node may have a backlog of queries outstanding for that program already. "Failing" over to another node via *N* configuration may allow the input to succeed in processing on a less busy node.

3.6 Logging and Tracing

A long string of functions would be difficult to debug without some variety of introspection. Riak Pipe includes just such a facility in the form of logging and tracing.

Via logging, any vnode or worker process can, at any time, send a message to the sink (the same process receiving the pipeline's results). This can be used to signal failure, note progress, or as a simple "printf"-type debugging utility.

Slightly more refined is the tracing facility. Tracing is built on top of logging, but includes the ability to define filters. Filters are used to selectively enable logging messages. For example, a sink might only enable error trace messages, or only messages originating from a certain fitting. Filtering is done at the source of the message, so fitting implementations can feel free to include many informative trace messages without increasing cluster traffic when they are disabled.

4. Example Usage

Riak Pipe does not have a direct interface available outside of a Riak cluster (though see section 5 for a discussion of one indirect interface). To use the system, code must be run directly on a cluster node. Even so, usage is quite simple.

As an example, the code in figure 2 sets up a pipeline that performs the trivial task of tallying the number of words that start with each letter.

The *fitting_spec* record is defined in Riak Pipe's include file. One instance is used to define each stage of the pipeline. Fields in the record configure how the stage behaves. Shown here are the tag applied to outputs and logs sent to the sink (*name*), the Erlang module that implements the behavior (*module*), an optional static argument for the module (*arg*), and the consistent hash function

```
Classify
   = fun(I, P, FD) when is_list(I) ->
         %% split on space
         Words = string:tokens(I, " "),

         %% classify by first character
         Outputs = [ {hd(Word), 1}
                     || Word <- Words ],

         %% sned output
         [ riak_pipe_vnode_worker:send_output(
              Output, P, FD)
           || Output <- Outputs ],
         ok;

       (_, _, _) ->
         %% ignore non-string input
         ok
     end,

Sum = fun(_Key, Inputs, _P, _FD) ->
          %% output is 1-element list containing
          %% sum of inputs
          %% (no need to send output here - the
          %%  reduce module does it for us)
          {ok, [lists:sum(Inputs)]}
      end,

Spec = [#fitting_spec{name   = classification,
                      module = riak_pipe_w_xform,
                      arg    = Classify},
        #fitting_spec{name   = summary,
                      module = riak_pipe_w_reduce,
                      arg    = Sum,
                     chashfun={riak_pipe_w_reduce,
                               chashfun}}],
Opts = [],
{ok, Pipe} = riak_pipe:exec(Spec, Opts).
```

Figure 2. Setting up a two-stage pipeline

(*chashfun*). Others include the size of its bounded queue (*q_limit*), and the *N* value (*nval*).

This pipe has two stages: "classification" and "summary". The classification stage is implemented by the *riak_pipe_w_xform* module. That module expects a function as an argument, and evaluates that function on each input. The function specified tokenizes strings by the space character, and then sends 2-tuples as output, where the first element of the tuple is the character begining the token, and the second element is a 1. That is, if this fitting receives the string, "foo bar baz", it will produce three outputs: {*$f, 1*},{*$b, 1*}, and {*$b, 1*} again (once for "bar" and once for "baz").

The summary stage is implemented by the *riak_pipe_w_reduce* module. That module expects a function as an argument, and evaluates that function on the list of inputs it has received for each tag, producing a new tagged tuple with the result of the function. The function specified here simply produces the sum of the inputs. That is, if this fitting receives {*$f, 1*},{*$b, 1*}, {*$b, 1*} as input, it will produce {*$f, [1]*} and {*$b, [2]*}.

Note the specification of consistent hashing functions for each stage. The classification stage uses the default has function, to simply distribute classification work across the cluster. The summary stage uses a special hash function, though, that depends only on the first element of the input tuple. In this way, all tuples containing the

```
ok = riak_pipe:queue_work(
        Pipe, "four score and seven"),
ok = riak_pipe:queue_work(
        Pipe, "peter piper picked a peck"),
ok = riak_pipe:queue_work(
        Pipe, "counting chickens"),
ok = riak_pipe:queue_work(
        Pipe, "adding sheep").
```

Figure 3. Sending inputs to a pipeline.

```
riak_pipe:eoi(Pipe).
```

Figure 4. Flushing and shutting down a pipeline.

```
{eoi, Results, []}
   = riak_pipe:collect_results(Pipe),
[{summary,{$a,[3]}},
 {summary,{$c,[2]}},
 {summary,{$f,[1]}},
 {summary,{$p,[4]}},
 {summary,{$s,[3]}}]
   = lists:sort(Results).
```

Figure 5. Collecting pipeline results

same first element arrive at the same vnode for summary, but tuples containing different first elements may be processed by different vnodes, to continue to benefit from parallel processing.

The *Pipe* value returned from *riak_pipe:exec/2* is a handle to a pipeline that implements the spec that was passed in. Using that handle, inputs can be sent to the pipe using *riak_pipe:queue_work/2*, as seeing in figure 3.

This code sends four inputs into the pipeline. When finished, the pipeline should be flushed and shutdown with *riak_pipe:eoi/1*, as in figure 4.

Finally, outputs must be collected. They have been sent to the current process as messages, because no alternate sink was designated at startup time. They could be read with a simple *receive* clause, but a convenience method is provided, called *riak_pipe:collect_results/1*, as in figure 5.

As the output shows, the input contained one word starting with *a*, two words starting with *c*, 1 with *f*, 4 with *p*, and 3 with *s*.

It is left as an exercise to the reader to re-run this trivial simulation to experiment with additional features like the logging and tracing system (see *riak_pipe_log:log/2* and *riak_pipe_log:trace/3*) as well as the status system (see *riak_pipe:status/1*), which will give information on live running behavior.

5. Practical Application in Riak KV

Riak Pipe's primary, and original, use case is to power a scatter-gather query system in the Riak key-value store. Many fittings are used in this query system, but the three at the core are "get", "transform", and "reduce".

5.1 Key-Value Lookup Fitting, a.k.a. "Get"

The first and most obvious fitting is simply a functional implementation of the common Dynamo-key-value-store fetching operation, "get". The processing function expects bucket-key pairs as inputs, and does the work of fetching their values to emit as outputs.

In order to enable this function to perform the task it's designed for, it has to be run from a place where it can access the storage for the key it is given as input. While Erlang makes this easy from "anywhere", the process will be much more efficient if it happens on the node storing the key.

The answer to co-locating the evaluation of this function with the data it needs is simply to use the same consistent hashing function that the key-value store used to store the key. In this way, the key as input to the pipe will be sent to the same vnode on the same node to which the key as a key-value query would have been sent.

The code for this fitting is contained in a module named *riak_kv_pipe_get*. It exports the hashing function *bkey_chash/1* to hash bucket-key pairs to their storage location.

The module also exports a function called *bkey_nval* which produces the number of replicas that should be stored of a given bucket-key pair's value. Using this function in the fitting's *nval* specification allows the lookup to be retried on nodes storing those replicas should any fetch fail.

5.2 Transform Fitting

Riak KV follows every use of its "get" fitting with a "transform" fitting. The transform fitting allows user code to inspect the value that was fetched by the get fitting, and then to produce one or more computed results based on it. These two fittings together make up what Riak KV calls a "map" phase.

For the transform fitting, any hash function would suffice – even a random number generator, though that violates the "consistent" part of the hash function – just to spread the processing load among the cluster's CPU resources. However, it is more efficient to exploit data locality again by keeping the input on the node where it was produced, saving a network hop.

Riak KV uses a shortcut that Riak Pipe includes for this sort of hashing. By using the atom *follow* in a fitting's *chashfun* specification field, Riak Pipe is able to skip the recalculation of and output's hash, and simply delivers it to the vnode that produced it.

Riak KV's transform fitting is defined in the *riak_kv_mrc_map* module.

5.3 Reduce Fitting

If "get" and "transform" make up the "scatter" part of the query system, "reduce" makes up the "gather" part. The purpose of reduce, as defined by the *riak_kv_w_reduce* module, is to allow code to produce an aggregation of results from earlier phases.

User code for the reduce phase accepts a list of inputs and produces a (possibly empty) list of outputs. The function may be run repeatedly, accepting new inputs as well as re-consuming its previous outputs. The idea is to continuously "reduce" new inputs to a common output. In some ways, it can be thought of similarly to the common "fold" list operator, constantly revising an "accumulator".

Riak KV uses a static hash function for this type of fitting. All outputs produced by the upstream fitting are delivered to a single vnode for processing during a reduce phase. The reduce phase also does not emit any output until the end-of-inputs signal comes through the pipe. This behavior allows this fitting to be used in many ways, including sorting and unique-ifying lists.

5.4 Using Get, Transform, and Reduce Fittings

An application that stores documents created by people at a company will serve as a demonstration for this scatter-gather query system. There are three record types: company, employee, document. Company records reference employee records, which in turn reference documents. A simple query would be counting the number of documents created by employees at a company which have been marked as "archived".

```
[{map,
  {modfun, company, employees},
   none,
   false},
 {map,
  {modfun, employee, documents},
  none,
  false},
 {map,
  {modfun, document, is_archived},
  none,
  false},
 {reduce,
  {moddun, riak_kv_mapreduce, reduce_sum},
  none,
  true}]
```

Figure 6. A Riak KV MapReduce query to count archived documents

Riak KV exposes an interface where the user specifies "map" phases that lookup and extract data from key-value pairs, and "reduce" phases that aggregate map phase results. In its syntax, the query just described looks like figure 6.

This is translated at query time to a Riak Pipe pipeline composed of seven stages, as seen in figure 7.

The input to this pipe line is a bucket-key pair naming the company value to lookup, such as {*"company","basho"*}. The first stage, {*kvget_map,0*}, would consume that input and lookup up the value for the companybasho object, and send that value to the next fitting, {*xform_map,0*}. That fitting applies the *company:employees/3* function referenced in "map" phase specification to the companybasho object. That function will produce one or more bucket-key pairs naming employees to look up, such as {*"employee", "bryan"*}, {*"employee", "scott"*}, and {*"employee", "joseph"*}. The {*kvget_map,1*} phase then looks up each of these objects, and sends them to the {*xform_map,1*} fitting, where the keys to the documents they reference are extracted and sent on. The {*kvget_map,2*} fitting looks up each of those documents, and the {*xform_map,2*} fitting examines each to see if it has been "archived". If the document has been archived, the {*xform_map,2*} fitting emits a one. Finally, all ones are collected by the *3* fitting, where the *riak_kv_mapreduce:reduce_sum/2* function adds them together.

Seven stages may seem like overkill for such a simple process. For example, the "kvget_map" and "xform_map" stages seem like they could be one combined stage, and surely the final could be left off the end, and the sum could be done by the sink. Each of those extra stages has a tangible benefit, though. Separating "kvget_map" and "xform_map" allows a bit of concurrent processing: one worker can wait on the lag of a disk seek, while the other is busy performing the computation on the last record that was read. Adding the "reduce" stage on the end, instead of accepting all ones and zeros at the sink both simplifies the sink's responsibilities, and also provides backpressure to prevent the unsummed list of ones and zeros from growing unboundedly if CPU is so limited that the simple sum operation goes unscheduled.

5.5 Analysis

The primary goal of Riak Pipe was to solve some problems seen when making large scatter-gather queries on Riak KV. The system in place before Riak Pipe had no backpressure mechanisms, and was therefore prone to intermediate result pile-ups. That system also used a custom "multi-get" KV operation, which often starved

```
[#fitting_spec{
     name = {kvget_map,0},
     module = riak_kv_pipe_get,
     chashfun = {riak_kv_pipe_get,bkey_chash},
     nval = {riak_kv_pipe_get,bkey_nval}},
 #fitting_spec{
     name = {xform_map,0},
     module = riak_kv_mrc_map,
     arg = {{modfun,company,employees},none},
     chashfun = follow},
 #fitting_spec{
     name = {kvget_map,1},
     module = riak_kv_pipe_get,
     chashfun = {riak_kv_pipe_get,bkey_chash},
     nval = {riak_kv_pipe_get,bkey_nval}},
 #fitting_spec{
     name = {xform_map,1},
     module = riak_kv_mrc_map,
     arg = {{modfun,employee,documents},none},
     chashfun = follow},
 #fitting_spec{
     name = {kvget_map,2},
     module = riak_kv_pipe_get,
     chashfun = {riak_kv_pipe_get,bkey_chash},
     nval = {riak_kv_pipe_get,bkey_nval}},
 #fitting_spec{
     name = {xform_map,2},
     module = riak_kv_mrc_map,
     arg = {{modfun,document,is_archived},none},
     chashfun = follow},
 #fitting_spec{
     name = 3,
     module = riak_kv_w_reduce,
     arg = {rct,
            #Fun<riak_kv_mapreduce.reduce_sum.2>,
            none},
     chashfun = <<11,79,99,133,216,174,35,189,0,
                  148,142,99,121,200,64,255,188,
                  166,40,70>>}]
```

Figure 7. A translation of figure 6 to Riak Pipe

other requests that were attempting to access KV vnodes. These two effects led to the unpredictable throughput that can be seen in figure 8.

These numbers were gathered from a Riak cluster of five Amazon EC2 m1.large nodes. The graph shows the throughput of a 50:50 mix of read and update Riak KV operations, as seen from a sixth EC2 m1.large node using the protocol buffers interface. The first 260 seconds are those operations running alone, with no scatter-gather queries in operation. After the 260 second mark, a scatter-gather query over 100,000 keys is run. The throughput not only drops, but also becomes wildly erratic, even pausing for a full ten seconds at the 800 second mark. The log for the node handling the query also shows warning messages about abnormally long garbage collection times. In fact, those garbage collections are so intense that they prevent the scatter-gather query from completing before the end of the 20 minute run.

The same test was was run again, this time using Riak Pipe for the scatter-gather query. The results of this run are in figure 9. This time, the throughput of the KV operations still drops when the scatter-gather queries start (at the 340 second mark), but it does not drop as far, and also does not become as erratic. No garbage collection time warnings (or any other type of warning) appears in

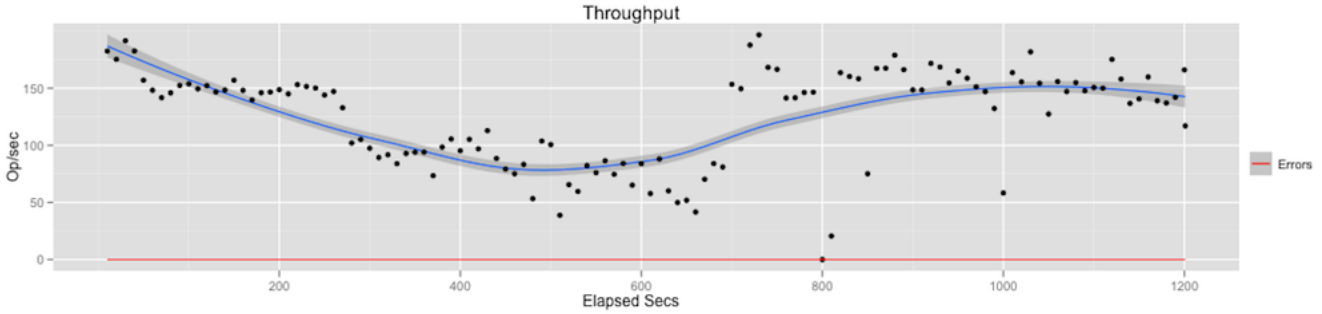

Figure 8. Throughput of KV operations before and during concurrent scatter-gather operation on pre-pipe system.

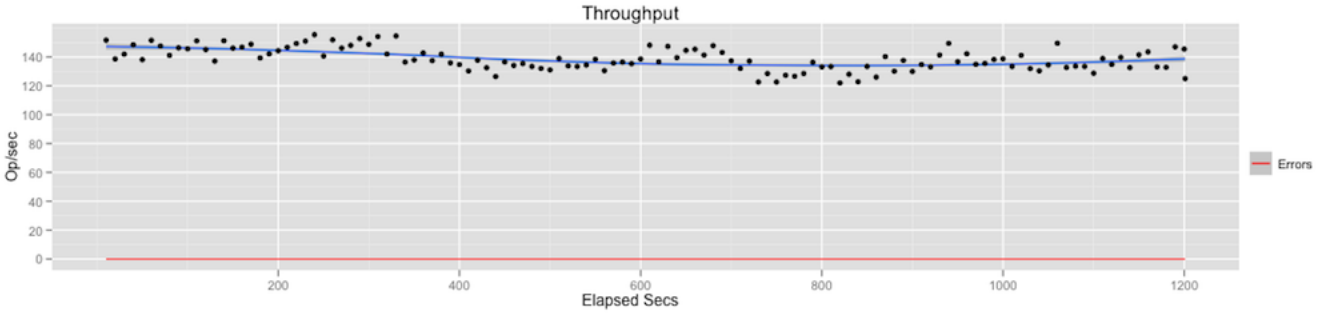

Figure 9. Throughput of KV operations before and during concurrent scatter-gather operation on Riak Pipe system.

the log. The scatter-gather query is run repeatedly for the rest of the test, each query lasting approximately 150 seconds.

A secondary goal of Riak Pipe was to make the code for such distributed queries more understandable. The previous system had a reputation for being difficult to reason about, from both a data-flow standpoint and a lifecycle standpoint.

The proof that Riak Pipe's implementation was more understandable came in the development of a new feature of Riak KV's scatter-gather system, known as pre-reduce. This feature allows a "reduce" phase to be run in two parts: the first is in a parallel manner, local to the workers of the previous phase, and the second is the normal single-location aggregation of those pre-reduced results. The change to enable this feature was made by simply adding an extra fitting to the pipeline, with the same parameters as the normal reduce fitting, but with a different consistent hash function that spread the work instead of consolidating it.

6. Related Work

Distributed computation is an area of active research, and has been for some time. The work most relevant to Riak Pipe's work is SEDA, an architecture described at Berkeley in 2001; Storm, a new application designed to consume modern social media streams; RabbitMQ and ZeroMQ, tools for passing messages between actors; and Hadoop, a popular distributed computation system.

6.1 SEDA

The seminal work on staged processing systems is the SEDA system, described by Welsh, Culler, and Brewer in 2001[2]. SEDA, short for *staged event-driven architecture*, describes a system for sharing the resources available to a server by describing work as a network of staged processing. The configuration of each stage describes not only what the stage does, but what resources it needs to complete its work. During operation, the system as a whole can then adapt to the load that it receives in a manner that guarantees certain performance characteristics.

For example, the paper discusses handling HTTP requests. Other servers allowed latency to grow or to become unpredictable as load increased above the server's capabilities. The SEDA system, meanwhile, traded some best-case low-load latency in exchange for providing predictable low latency once the server became saturated. SEDA was able to accomplish this by use of bounded queues and blocking or rejection of requests that would not fit in the server's capacity.

Riak Pipe uses the same mechanism to accomplish the same goal. Breaking work into stages allows for clean definition of which processing may happen in parallel, and also encourages component reuse. Bounded queues at each stage provide a means for limiting the amount of in-flight work. Backpressure can be provided via blocking on queue entry, to slow upstream production, or rejection of new work can signal that the system is over capacity, without interruping other work already in progress.

One major departure from SEDA is the absence of an adaptive scheduler in Riak Pipe. This is primarily due to Riak Pipe's choice of implementation language, Erlang. The Erlang VM does a reasonable job of fairly scheduling processes for execution. By merely limiting the number of active workers and delivering inputs to them as available, the VM does a good enough job of scheduling that implementing an alternate was not necessary for the version described here. Of further interest is that, while Riak Pipe's choice of Erlang is primarily one to enable easy integration with the Riak key-value store, the secondary reason is the same as SEDA's choice of Java, namely, "the software engineering and robustness benefits of using [Erlang] have more than outweighed the performance tradeoffs."[3]

Functional purity where possible, combined with simple threading abstractions, have made Riak Pipe's implementation easy and obvious to reason about.

Further differences between Riak Pipe and SEDA are largely due to Riak Pipe's goal of *distributed* computation (SEDA's focus is on a single machine), and Riak Pipe's relatively young age (there has not yet been time to consider the more advanced performance enhancements found in SEDA, such as batch scheduling).

6.2 Storm

Storm[4] is another recent entrant to the world of staged processing. Storm uses the concepts of *spouts* (stream emitters) and *bolts* (stream consumers and producers) to describe networks of data processors. Storm was designed to consume live streams of data, and to produce realtime analysis of them.

While Riak Pipe could be used in much the same way as Storm, the two make significantly different decisions about cluster management and processing guarantees.

Storm's cluster management is a tiered system of "master" and "worker" nodes, and the coordination between them relies on an Apache ZooKeeper[5] cluster. Riak Pipe, meanwhile, organizes all nodes as equal peers and coordinates them using ideas from the Dynamo system. While Riak Pipe's choice is primarily an artifact of its integration with the Riak key-value storage system, it also makes the administration of a Riak Pipe cluster (upgrade, repair, etc.) much simpler to reason about.

A key benefit to Storm's design is its *at-least-once* processing guarantee. Every output of a stream is guaranteed to be processed at least once by the bolts consuming it, so no intermediate results will be lost. Riak Pipe does not offer this guarantee, instead opting to log errors about lost inputs, allowing the application using the pipeline to decide what to do with them.

6.3 Message Queues

RabbitMQ[6] and ZeroMQ[7] embody message passing systems that can be used to coordinate staged data-flow computation. They have been used successfully by many projects to transfer data between processing stages simply, even when those stages are written in different languages or running on different operating systems.

Riak Pipe forgoes the use of either of these message queue libraries, primarily because of its choice of Erlang as an implementation language. Erlang has built in primitives for passing messages between processes, which work both within a VM instance and between VM instances, even when those instances are on separate physical machines. While the delivery guarantees for Erlang messages are different than those of RabbitMQ and ZeroMQ, they proved acceptable for Riak Pipe's use.

6.4 Hadoop

Hadoop[8] is a distributed computation system that also provides its own storage system (HDFS) in order to exploit data-locality for large batch processing. Hadoop's primary focus is on large-scale batch processing, as opposed to online or realtime stream processing.

Hadoop and Riak Pipe continue to grow toward one another. While Riak Pipe's initial implementation is focused on online processing, batch processing is a field where growth is desired. Many Hadoop users have also found ways to use or improve parts of Hadoop to make it suitable for some online processing.[9]

7. Future Work

Riak Pipe's implementation is in production and serving well in its current form, but there are still improvements we look forward to exploring.

Among these improvements are several of the more advanced performance ideas covered in the SEDA paper. Batched input transfer between workers and vnodes, for example, is likely to reduce overhead and latency. Preallocation of limited resources will likely improve the performance of predictable query loads.

Other improvements involve interfacing with external, non-Erlang applications. Some work has been completed to allow access to a Javascript VM from the fittings that the Riak key-value store's query system uses, but a more generic method would broaden the field. Operating a pipeline also currently requires directly running Erlang code on a node in the cluster. Providing an interface for controlling pipelines over a generic protocol like HTTP would expand the set of applications that could use Riak Pipe.

There is also the constant desire to be more user-friendly. Better metrics for pipeline performance would help administrators. Visual exploration tools would help with rapid development for both new and experienced users.

A. Code Availability

The code for Riak Pipe is provided by Basho Technologies under the Apache 2 license, and is available at `http://github.com/basho/riak_pipe`. There you will also find additional documentation on using Riak Pipe, as well as links to other components, such as Riak Core, that are required to compose a running system. The easiest path to experimenting with Riak Pipe is to download and install Basho's Riak key-value store, a free implementation of a Dynamo-style storage system that includes Riak Pipe as a component.

Acknowledgments

Thanks to Justin Sheehy, Steve Vinoski, Scott Fritchie, Amanda Zoellner, Joseph Blomstedt, and Sean Cribbs for reviewing this paper. Thanks also to Scott for helping test and implement Riak Pipe and its use in the Riak key-value store. Thanks to David Smith for the initial prod and later benchmarking.

References

[1] 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. *SOSP 2007, October 14-17, 2007, Stevenson, Washington, USA.*

[2] M. Welsh, D. Culler, E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *SOSP 2001, October 21-24, 2001, Chateau Lake Louise, Canada.*

[3] *ibid*, section 3.5.

[4] Nathan Marz, Backtype. Storm. `https://github.com/nathanmarz/storm`.

[5] The Apache Software Foundation. ZooKeeper. `http://zookeeper.apache.org/`

[6] VMware, Inc. RabbitMQ `http://www.rabbitmq.com/`

[7] iMatix Corporation. ZeroMQ. `http://www.zeromq.org/`

[8] The Apache Software Foundation. Hadoop. `http://hadoop.apache.org/`

[9] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, R. Sears. MapReduce Online. 2010. `http://www.neilconway.org/docs/nsdi2010_hop.pdf`

[10] Basho Technologies, Inc. Riak. `http://wiki.basho.com/Riak.html`

[11] M. Nottingham. Web Linking. Internet Engineering Task Force, Request for Comments 5988. `http://tools.ietf.org/html/rfc5988`

[12] J. Dean, and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.*