

Continuous Pipelines at Google

Dan Dennison

dennison@google.com

12 May 2015

Table of Contents

[Abstract for SREcon Europe 2015](#)

[Abstract](#)

[Origin of the Pipeline Design Pattern](#)

[Initial Effect of Big Data on the Simple Pipeline Pattern](#)

[Challenges to the Periodic Pipeline Pattern](#)

[Trouble Caused By Uneven Work Distribution](#)

[Drawbacks of Periodic Pipelines in Distributed Environments](#)

[Monitoring Problems in Periodic Pipelines](#)

[“Thundering Herd” Problems](#)

[Moiré Load Pattern](#)

[Introduction to Google Workflow](#)

[Workflow as Model-View-Controller Pattern](#)

[Stages of Execution in Workflow](#)

[Workflow Correctness Guarantees](#)

[Ensuring Business Continuity](#)

[Summary and Concluding Remarks](#)

[Acknowledgements](#)

Abstract for SREcon Europe 2015

This presentation will focus on the real life challenges of managing data processing pipelines of depth and complexity. I'll cover the frequency continuum between periodic pipelines that run very infrequently up to continuous pipelines that never stop running, discussing the discontinuities along the axis that can produce significant operational problems. A fresh take on the master-slave model is presented as a better alternative to the periodic pipeline for reliable Big Data scaling.

Abstract

This article focuses on the real life challenges of managing data processing pipelines of depth and complexity. It considers the frequency continuum between periodic pipelines that run very infrequently through continuous pipelines that never stop running, and discusses the discontinuities that can produce significant operational problems. A fresh take on the

master-slave model is presented as a more reliable and better scaling alternative to the periodic pipeline for processing Big Data.

Origin of the Pipeline Design Pattern

The classic approach to data processing is to write a program that reads in data, transforms it in some desired way, and outputs new data. Typically, the program is scheduled to run under the control of a periodic scheduling program such as cron. This design pattern is called a “data pipeline.” Data pipelines go as far back as co-routines¹, the DTSS communication files², the UNIX pipe³, and later, ETL⁴ pipelines, but they have gained increased attention with the rise of “Big Data”, or “datasets that are so large and so complex that traditional data processing applications are inadequate”⁵.

Initial Effect of Big Data on the Simple Pipeline Pattern

Programs that perform periodic or continuous transformations on Big Data are usually referred to as “simple, one-phase pipelines.”

Given the scale and processing complexity inherent with Big Data, programs are typically organized into a chained series, with the output of one program becoming the input to the next. There may be varied rationale for this arrangement, but it is typically for ease of reasoning about the system and not usually geared toward operational efficiency. Programs organized this way are called “multi-phase pipelines,” since each program in the chain acts as a discrete, data processing phase.

The number of programs chained together in series is a measurement known as the “depth” of a pipeline. Thus, a shallow pipeline may only have one program with a corresponding pipeline depth measurement of 1, whereas a deep pipeline may have a pipeline depth in the tens or hundreds of programs.

Challenges to the Periodic Pipeline Pattern

Periodic pipelines are generally stable when there are sufficient workers for the volume of data and execution demand is within computational capacity. In addition, instabilities, such as processing bottlenecks, are avoided when the number of chained jobs and the relative throughput between jobs remain uniform.

Periodic pipelines are useful and practical, and we run them on a regular basis at Google. They are written with frameworks like MapReduce⁶, Flume⁷ and others.

¹ Design of a Separable Transition Diagram Compiler, CACM Volume 6 Number 7 1963, <http://www.citemaster.net/get/ab3afb46-1d2a-11e4-ac35-00163e009cc7/p396-conway.pdf>

² The Dartmouth Time Sharing System, http://bitsavers.informatik.uni-stuttgart.de/pdf/dartmouth/The_Dartmouth_Time-Sharing_System_1980.pdf

³ A Research UNIX Reader, Doug McIlroy, <http://www.cs.dartmouth.edu/~doug/reader.pdf>

⁴ Wikipedia: Extract, transform, load, http://en.wikipedia.org/wiki/Extract,_transform,_load

⁵ Wikipedia: Big Data, http://en.wikipedia.org/wiki/Big_data

⁶ *MapReduce: Simplified Data Processing on Large Clusters*, <http://research.google.com/archive/mapreduce-osdi04.pdf>

However, the collective SRE experience has been that the periodic pipeline model is fragile. We have discovered that when a periodic pipeline is first installed with worker sizing, periodicity, chunking technique, and other parameters carefully tuned, the initial performance is reliable for a while. However, organic growth and change begin to stress the system, and problems arise. Examples of such problems include: jobs that exceed their run deadline, resource exhaustion, and hanging processing chunks, bringing with them corresponding operational load.

Trouble Caused By Uneven Work Distribution

The key breakthrough of Big Data is the widespread application of “embarrassingly parallel”⁸ algorithms to cut a large workload into chunks small enough to fit onto individual machines. Sometimes chunks require an uneven amount of resources relative to one another, and it is seldom obvious at first why particular chunks require different amounts of resources. For example, in a workload that is partitioned by customer, some customers may be much larger than others, and since customer is the point of indivisibility, end-to-end runtime is thus capped to the runtime of the largest customer.

If insufficient resources are assigned, whether caused by differences between machines in a cluster or overall allocation to the job, it often results in the “hanging chunk” problem since the typical pattern of user code is to wait for the total computation to be complete before progressing to the next pipeline stage. This can significantly delay pipeline completion time, since it is blocked on the worst-case performance as dictated by the chunking methodology in use.

If this problem is detected by engineers or cluster monitoring infrastructure, the response can make matters worse. For example, the “sensible” or “default” response to a “hanging chunk” is to immediately kill the job, and allow it to restart, since the blockage may well be the result of nondeterministic factors. However, since, by design, pipeline implementations usually don’t include checkpointing, work on all chunks will start over from the beginning, thereby wasting the time, CPU cycles, and human effort invested in the previous cycle.

Drawbacks of Periodic Pipelines in Distributed Environments

Big Data periodic pipelines are widely used at Google and so Google’s cluster management solution includes an alternative scheduling mechanism for them. This is necessary since, unlike continuously running pipelines, periodic pipelines typically run as lower priority batch jobs, and this designation works well for its purpose since batch work is not sensitive to latency in the way that Internet-facing web services are. In addition, to control cost, the cluster management system at Google, called Borg⁹, assigns batch work to available machines to maximize machine

⁷ *FlumeJava: Easy, Efficient Data-Parallel Pipelines*,
<http://pages.cs.wisc.edu/~akella/CS838/F12/838-CloudPapers/FlumeJava.pdf>

⁸ Moler, Cleve: Matrix Computation on Distributed Memory Multiprocessors, Hypercube Multiprocessors page 182

⁹ Large-scale cluster management at Google with Borg, <https://research.google.com/pubs/archive/43438.pdf>

workload. This priority can result in degraded startup latency, so pipeline jobs can potentially experience open-ended startup delays.

Jobs invoked using this mechanism have a number of natural limitations due to being scheduled in the gaps left by e.g. user-facing web service jobs, and have various distinct behaviors relating to the properties that flow from that, such as availability of low-latency resources, pricing, stability of access to resources, and so on. Execution cost is inversely proportional to requested startup delay, and directly proportional to resources consumed. Although it may work smoothly in practice, excessive use of the batch scheduler¹⁰ places jobs at risk of preemptions¹¹ when cluster load is high, due to starving other users of batch resources. In light of the risk tradeoffs, running a well-tuned periodic pipeline successfully is a delicate balance between high resource cost and risk of preemptions.

Delays of up to a few hours might well be acceptable for pipelines that run daily, but as the scheduled execution frequency increases, the minimum time between executions can quickly reach the minimum average delay point, placing a lower bound on the latency that a periodic pipeline can expect to attain. Further, reducing the job execution interval below this effective lower bound will simply result in undesirable behavior rather than increased progress. The specific failure mode depends on the batch scheduling policy in use. For example as each run begins it could stack up on the cluster scheduler since the previous run is not complete, or worse, the currently executing and nearly finished run could be killed when the next execution is scheduled to begin, completely stopping all progress in the name of increasing executions.

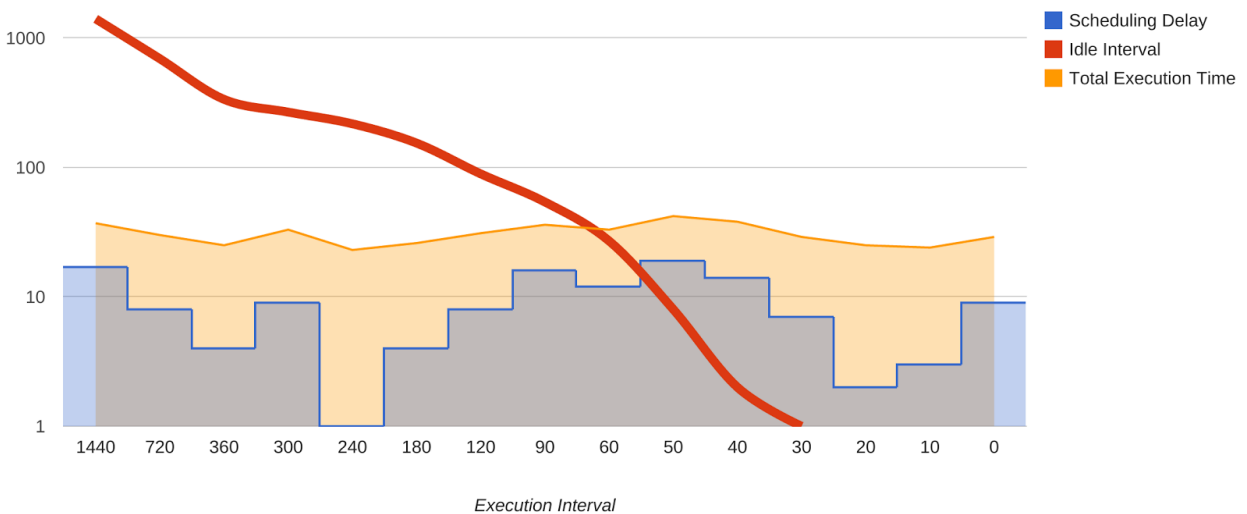


Diagram 1: Periodic Pipeline Execution Interval versus Idle Time (log scale)

¹⁰ *ACM Queue: Reliable Cron across the Planet*, <https://queue.acm.org/detail.cfm?id=2745840>

¹¹ Large-scale cluster management at Google with Borg, §2.5

Note in the diagram above where the red line intersects the blue scheduling delay. In this scenario, lowering the execution interval much below 40 minutes for this ~20 minute job results in potentially overlapping executions with the discussed consequences.

The solution to this problem is to secure sufficient server capacity for proper operation. However, resource acquisition in a shared, distributed environment is susceptible to supply and demand as would be expected. Consequentially, development teams tend to be reluctant to go through the processes of acquiring resources which they must contribute to a common pool and share.

Monitoring Problems in Periodic Pipelines

For pipelines of sufficient execution duration, having real-time information on runtime performance metrics can be as important, if not even more important, than knowing overall metrics since real-time data is important to providing operational support, including emergency response. In practice, the standard monitoring model involves collecting metrics during job execution, and reporting it only upon completion. If the job fails during execution, no statistics are provided.

Continuous pipelines do not share these problems since their tasks are constantly running and their telemetry is routinely designed so that real time metrics are available. While it is true that these problems are not inherent to periodic pipelines, they are nonetheless strongly associated in observed practice at Google.

“Thundering Herd” Problems

Adding to execution and monitoring challenges is the “thundering herd” problem endemic to periodic pipelines. Given a large enough periodic pipeline, for each cycle, potentially thousands of workers immediately start work. If there are too many workers or if they are misconfigured they will subsequently overwhelm their own servers upon which they depend, underlying shared cluster services, and any networking infrastructure that may be serving them.

Making things even worse, if retry logic is not implemented, correctness problems can result, due to work being dropped upon failure since it isn't retried. On the other hand, even if retry logic is implemented, it will typically compound the issue by accelerating the rate at which the already overwhelmed servers receive requests, speeding their demise.

Engineers with limited experience managing pipelines tend to amplify this problem by adding more workers to their pipeline when it fails to complete within a desired period of time. Nothing is harder on cluster infrastructure and the site reliability engineers of various services than a buggy 10,000 worker pipeline job.

Moiré Load Pattern

Sometimes the thundering herd problem may not be obvious to spot in isolation. A related problem I call the “Moiré load pattern” occurs when two or more pipelines run simultaneously and their execution sequences occasionally overlap, causing them to simultaneously consume a common shared resource. This problem can occur even in continuous pipelines, although it is less common when load arrives more evenly.

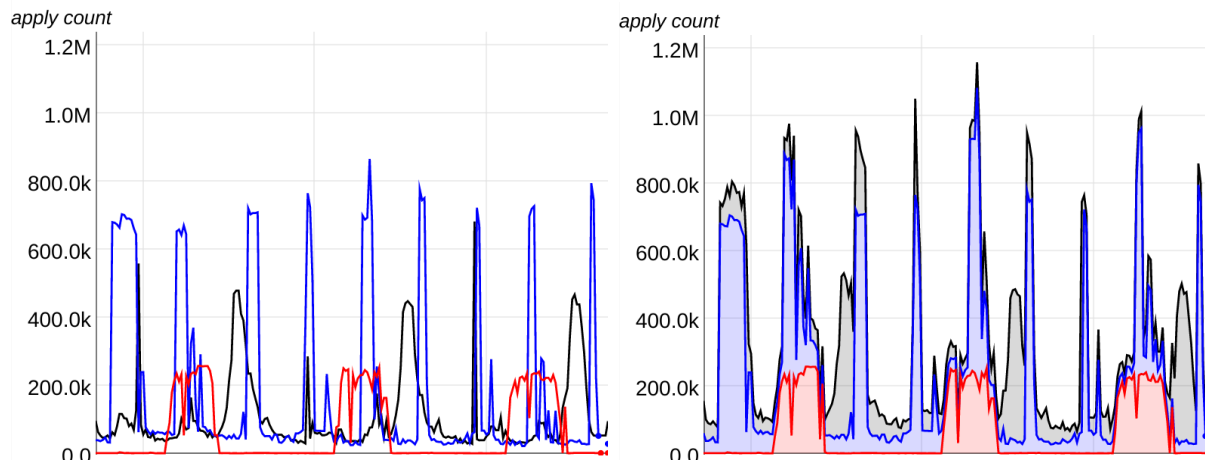


Diagram 2: Moiré load pattern in 3 pipelines versus shared infrastructure

Moiré load patterns are most apparent in plots of pipeline usage of shared resources. For example, note in the diagram above, that resource usage of three periodic pipelines are identified by red, blue, and gray lines. On the right, which is a stacked graph of the data on the left, the peak impact causing oncall pain occurs when the aggregate load nears 1.2M.

Introduction to Google Workflow

When an inherently one-shot batch pipeline is overwhelmed by business demands for continuously updated results, the pipeline development team will usually consider refactoring the original design to satisfy current demands, or moving to a continuous pipeline model. Unfortunately, business demands usually come at the least convenient time to refactor the pipeline system into an online continuous processing system. Newer and larger customers who are faced with forcing scaling issues typically also want to include new features, and expect that these requirements adhere to immovable deadlines. Anticipating this challenge, it's important to ascertain at the outset of designing a system involving a proposed data pipeline details such as: expected growth trajectory¹², demand for design modifications, additional resources, and expected latency requirements from the business.

¹² Jeff Dean's lecture on Software Engineering Advice from Building Large-Scale Distributed Systems is an excellent resource: <http://research.google.com/people/jeff/stanford-295-talk.pdf>

Faced with these needs, Google developed a system in 2003 called “Workflow” that makes continuous processing available at scale. Workflow uses the master-slave (workers) distributed systems design pattern¹³ combined with the system prevalence design pattern¹⁴. The combination of these design patterns enables very large scale transactional data pipelines, ensuring correctness, including exactly-once semantics.

Workflow as Model-View-Controller Pattern

Because of system prevalence, it can be useful to think of Workflow as the distributed systems equivalent of the model-view-controller pattern known from user interface development¹⁵. This design pattern divides a given software application into three interconnected parts to separate internal representations of information from the ways that information is presented to or accepted from the user.¹⁶

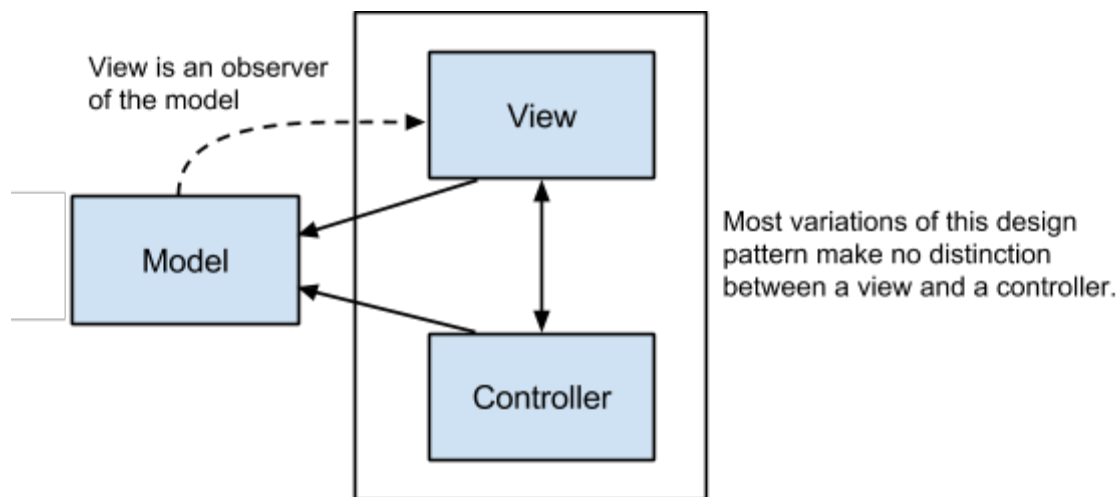


Diagram 3: The model-view-controller pattern used in user interface design.

Adapting this pattern for Workflow, the **model** is held in a server called “Task Master”. The Task Master uses the system prevalence pattern as discussed to hold all job state in memory for fast availability, while synchronously journaling mutations to persistent disk. The **view** are the workers that continually update the system state transactionally with the master according to their perspective as a subcomponent of the pipeline. Although all pipeline data may be stored in the Task Master, generally speaking for performance, only pointers to work are stored in the Task Master, with the actual input and output data stored elsewhere such as a common filesystem. Adding credence to the view analogy, the workers are completely stateless and can

¹³ *Master/Slave Computing on Grid*, <http://www.cs.ucsb.edu/~rich/publications/shao-hcw.pdf>

¹⁴ *Wikipedia: System Prevalence*, http://en.wikipedia.org/wiki/System_Prevalence

¹⁵ *The “model view controller” pattern used here is an analogy for distributed systems that was very loosely borrowed from Smalltalk, which was originally used to describe the design structure of graphical user interfaces. See GUI Architectures, Martin Fowler, 2008, http://martinfowler.com/eaDev/uiArchs.html*

¹⁶ *Wikipedia: Model-view-controller*, <http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>

be discarded at any time. A **controller** can optionally be added as a third system component to efficiently support a number of auxiliary system activities that affect the pipeline such as run-time scaling of the pipeline, snapshotting, workcycle state control, rolling back pipeline state, or even performing global interdiction for business continuity.

This design pattern is shown in the following diagram:

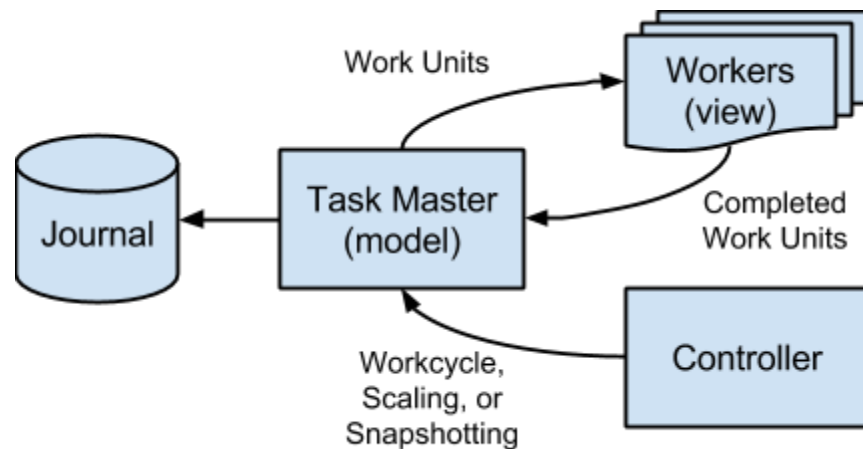


Diagram 4: The Model-View-Controller design pattern as adapted for Google Workflow.

Stages of Execution in Workflow

Pipeline depth can be increased to any level inside Workflow by subdividing processing into task groups held in the Task Master. Each task group holds the work corresponding to a pipeline stage which can perform arbitrary operations on some piece of data. It's relatively straightforward to perform mapping, shuffling, sorting, splitting, merging, or any other operation in any stage.

A given stage usually has some worker type associated with it. There can be multiple concurrent instances of a given worker type and workers can be self scheduled in the sense that they can look for different types of work and choose which to do.

The worker consumes work units from a previous stage and produces output units. The output can be an end point or input for some other processing stage. Within the system it's easy to guarantee that all work is executed, or at least, reflected in permanent state, exactly once.

Workflow Correctness Guarantees

It's not practical to store all pipeline state inside the Task Master, since Task Master retains all state resident in memory for performance. However, a double correctness guarantee persists because the master holds a collection of pointers to uniquely named data, and each work unit has a uniquely held lease. Workers obtain a lease when acquiring work and may only commit work pertaining to tasks for which they currently possess a valid lease.

To avoid the situation where an orphaned worker may continue working on a work unit, destroying the work of the current worker, each output file opened by a worker has a unique name. In this way, even orphaned workers can continue writing independently of the master until they attempt to commit. Upon commit, they will be unable to do so, because another worker holds the lease for that work unit. Furthermore, orphaned workers cannot destroy the work produced by a valid worker since the unique filename scheme ensures every worker is writing to a distinct file. In this way the double correctness guarantee holds: the output files are always unique, and the pipeline state is always correct by virtue of tasks with leases.

As if a double correctness guarantee isn't enough, Workflow also versions all tasks, so that if either the task updates, or the task lease changes, each operation yields a new unique task replacing the previous one, with a new id assigned to it. Note that since all pipeline configuration in Workflow is stored inside the Task Master in the same form as the work units themselves, to commit work successfully, a worker must own an active lease and reference the task id number of the configuration it used to produce its result. If the configuration changed while it was working on the work unit, all workers of the type will be unable to commit despite owning current leases. Thus, all work performed after a configuration change is consistent with the new configuration, at the cost of work being thrown away by those workers unfortunate enough to have the old leases.

These measures provide a triple correctness guarantee: configuration, lease ownership, and filename uniqueness. However, even this isn't enough for all cases.

For example, what if the Task Master's network address changed, and a different Task Master replaced it at the same address? What if a memory corruption altered the IP address or port number and it so happened to result in another Task Master on the other end? Even more common, what if someone (mis)configured their Task Master setup by inserting a load balancer in front of a set of independent Task Masters?

For such cases, a fourth and final correctness guarantee has been added to Workflow - the server token. The server token is a unique identifier for this particular Task Master, and is located in the task metadata of each and every task. Both client and server check this token to ensure no rogue or incorrectly configured Task Master introduces corruption into the pipeline. The last case of a load balancer is a very subtle misconfiguration, where everything will appear to work fine until there is a task identifier collision.

To summarize, the four Workflow correctness guarantees are:

1. Consistent worker output through configuration tasks used as barriers to predicate work on
2. All work committed requires a currently valid lease held by the worker
3. Output files are uniquely named by the workers
4. The Task Master itself is validated by checking a server token on every operation

At this point, it may occur to you that it would be simpler to forgo the specialized Task Master and use Spanner¹⁷ or another database. However, what makes Workflow special is that each task is unique and immutable. These twin properties prevent many potentially subtle issues with wide-scale work distribution from occurring.

For example, the lease obtained by the worker is part of the task itself, requiring a brand new task for even lease changes. If a database is used directly and its transaction logs act like a “journal”, this requires that each and every read to be part of a long running transaction. This is most certainly possible, but terribly inefficient.

Ensuring Business Continuity

Big Data pipelines need to keep processing despite failures of all types, including fiber cuts, weather events, and cascading power grid failures. These types of failures can disable datacenters that experience them. In addition, pipelines that do not employ system prevalence to obtain strong guarantees about job completion are often disabled and enter an undefined state. This architecture gap makes for a brittle business continuity strategy, and entails costly mass duplication of effort to restore pipelines and data.

Workflow resolves this problem conclusively for continuous processing pipelines. To obtain global consistency, the Task Master stores its journals on Spanner, using it as a globally available, globally consistent, but low throughput filesystem. To determine which Task Master can write, each Task Master uses the distributed lock service called Chubby¹⁸ to elect the writer and the result is persisted in Spanner. Finally, clients look up the current Task Master using internal naming services.

Since Spanner does not make for a high throughput filesystem, globally distributed Workflows employ two or more local Workflows running in distinct clusters, and a notion of reference tasks stored in the global Workflow. As units of work (tasks) are consumed through a pipeline, equivalent reference tasks are inserted into the global Workflow by the binary labeled ‘stage 1’ in the diagram below. As tasks are finished, the reference tasks are transactionally removed from the global Workflow as depicted in ‘stage n’ of the diagram. If the tasks cannot be removed from the global Workflow, the local Workflow will block until the global workflow becomes available again, ensuring transactional correctness.

To automate failover, a helper binary labeled ‘stage 1’ in the diagram below runs inside of each local Workflow. The local Workflow is otherwise unaltered, as described by the ‘do work’ box in the diagram. This helper binary acts as a ‘controller’ in the MVC sense, and is responsible for creating reference tasks as well as updating a special heartbeat task inside of the global

¹⁷ *Spanner: Google's Globally-Distributed Database*, <http://research.google.com/archive/spanner.html>

¹⁸ *The Chubby lock service for loosely coupled distributed systems*, <http://research.google.com/en/us/archive/chubby-osdi06.pdf>

Workflow. If the heartbeat task is not updated within the timeout period, the remote Workflow's helper binary will seize the work in progress as documented by the reference tasks and the pipeline continues, unhindered by whatever the environment may do to it.

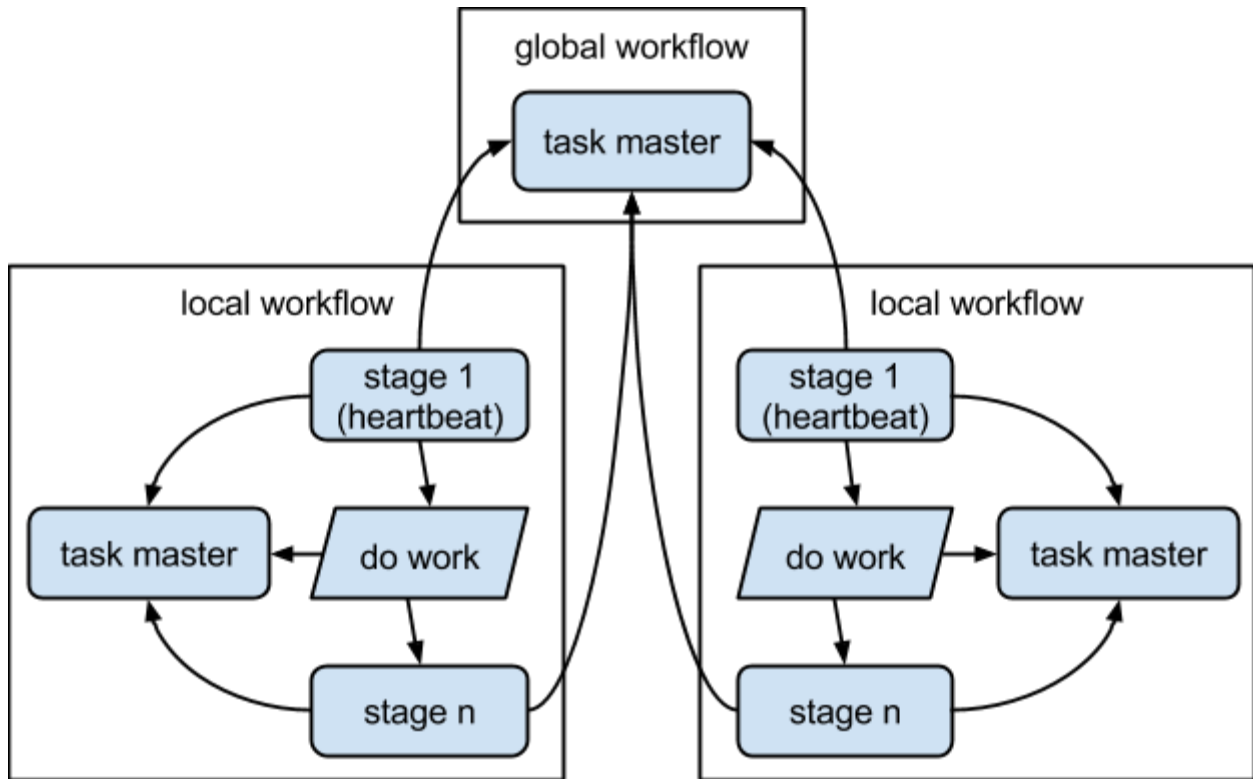


Diagram 5: An example of distributed data and process flow using Workflow pipelines.

Summary and Concluding Remarks

Big Data pipelines present many new and interesting challenges to Google Site Reliability Engineering.

If there is a single take-away, it is that periodic pipelines have specific value. However, if a data processing problem is continuous or will organically grow to become continuous, it should not be performed on a periodic pipeline. Instead, seek and use an alternative pipeline technology with characteristics like Workflow, described in this article.

We have found that continuous data processing with strong guarantees, like Workflow, performs and scales well on distributed, cluster infrastructure; routinely produces results that users can rely on; and is a stable and reliable system for the Site Reliability Engineering team to manage and maintain.

Acknowledgements

This paper owes much to the important lessons learned day to day in a culture that values both the amazing products people use every day worldwide and strong engineering practices. Many

colleagues have helped shape our thoughts here, and the benefit of accumulated group wisdom cannot be overstated. I would like to specifically recognize the following reviewers: Dylan Curley, Tim Harvey, Arnar Mar Hrafnkelsson, Chris Jones, Niall Murphy, Dieter Pearcey, Graham Poulter, Abdul Salem, and Todd Underwood.