university of groningen

faculty of mathematics and natural sciences

Scalable monitoring of a highly dynamic metric set

Master's thesis

August 2012

Student: P.C. Noordhuis

Primary supervisor: A. Lazovik, University of Groningen

Secondary supervisor: M. Lucovsky, VMware Inc.

CONTENTS

1 INTRODUCTION 1
2 PROBLEM DOMAIN 3
- 2.1 Why Monitoring? 3
- 2.2 Types of Failures 4
- 2.3 Types of Monitoring 6
  - 2.3.1 Interactive/Non-Interactive 7
  - 2.3.2 Active/Passive 7
- 2.4 Types of Data to Monitor 9
- 2.5 Context: Cloud Foundry 10
  - 2.5.1 Components 10
  - 2.5.2 Monitoring Challenges 12

3 RELATED WORK 13
- 3.1 Logs 13
- 3.2 Metrics 14

4 ANALYSIS & DESIGN 17
- 4.1 Data Model 17
  - 4.1.1 Logs or metrics? 17
  - 4.1.2 Dimensionality 18
  - 4.1.3 Encoding more dimensions into one 19
  - 4.1.4 Adding more context 19
- 4.2 Querying 23
  - 4.2.1 What about `grep` and `tail`? 23
  - 4.2.2 Goals 25
- 4.3 Storage And Retrieval 27
  - 4.3.1 Flat Files 29
  - 4.3.2 Relational Database 30
  - 4.3.3 Key/Value Database 31
  - 4.3.4 Evaluation 34
- 4.4 Distribution 36
  - 4.4.1 Topology 36
  - 4.4.2 Partitioning 42

5 IMPLEMENTATION 45
- 5.1 Time 45
- 5.2 Locality 47
- 5.3 Format 49
- 5.4 Indexing 52
- 5.5 Querying 53

6 CONCLUSION 55

BIBLIOGRAPHY 57

LIST OF FIGURES

Figure 1 Log entry formatted using the Common Log Format. Example taken from [20]. *Note: the \ denotes line continuation.* 20
Figure 2 Trade-off between read-time and write-time cost 22
Figure 3 Example use of `grep` and `tail`. (these lines use an abbreviated version of the Common Log Format) 24
Figure 4 What a querying the described system could look like. 26
Figure 5 Topology where all processes run on a single node 37
Figure 6 Topology where querying processes are distributed 38
Figure 7 Topology where log record production is distributed 38
Figure 8 Topology with a separate tier for storing processes 40
Figure 9 Topology with external reliable storage 41
Figure 10 Memory usage and a locality measure for storing log records with constant size from multiple streams in a single file. Streams are buffered, and buffers are flushed when the cumulative memory usage exceeds 64 kB. 49
Figure 11 Serialized format of groups of log records and log records themselves. 51

1 INTRODUCTION

Over the past decade, more companies have started applying virtualization in their data centers. Because of the continuously growing demand for computing resources, more efficient data center operation becomes increasingly important. Virtualization enables consolidation of physical resources, making it a key technique to allow data centers to operate more efficiently.

The ability to allocate resources at run-time makes scaling applications up or down less involved than it would be with a static set of resources. Without virtualization, applications require a static set of resources that is able to handle peak load, even when mostly idle. With virtualization the allocation of resources can closely follow the load pattern the application experiences, reducing cost when load is low.

Dynamic resource allocation can be done in the scope of a single virtual machine, where the amount of processing power and memory it can use can throttled by the hypervisor. While this can be useful for equally distributing the resources of a physical machine among many virtual machines it hosts, this approach is not often used to scale application. In practice, dynamic resource allocation is more often done by starting new virtual machines and shutting existing ones down.

While this does require the application in question to be able to run on multiple machines – be it virtual or physical – it eliminates the capacity ceiling as present for a single machine, while retaining fine grained control over the set of resources that is used.

Applications that run following this ephemeral approach tend to not depend on the underlying physical hardware and geographical location. Their scaling model can make individual virtual machines extremely short-lived.

The network topology that is associated with such application can be characterized as highly dynamic. This in contrast to older topologies, where physical machines are long-lived and bound to a single geographical location. Assumptions made in tooling that was created in the era of static network topologies may not hold true when applied to highly dynamic network topologies. In particular, how can applications that move through physical hardware and geographical locations be monitored without losing track of the big picture?

In this thesis, we focus on the difficulties that arise when monitoring applications that run in highly dynamic network topologies, scale dynamically, and have a dynamic set of properties to keep track of.

We start with exploring various types of monitoring, and the context for the work in this thesis in Chapter 2. In Chapter 3, we briefly discuss existing monitoring techniques, their models and implementations. An exploration of different data models and storage techniques is presented in Chapter 4. In Chapter 5, the challenges in the implementation of a proof of concept are discussed. We conclude in Chapter 6.

2 PROBLEM DOMAIN

Why do systems need to be monitored? How can monitoring be executed? What are different types of data points that can be monitored? We discuss the rationale for monitoring systems and make an attempt to answer these questions in this chapter. Additionally, we introduce Cloud Foundry, the opportunities in monitoring it, and discuss the intricacies of the data that it produces and the challenges in capturing and analyzing this data.

2.1 WHY MONITORING?

System monitoring has different meaning for different people. A person involved with operating a system likely thinks of ensuring system availability when talking about monitoring, while a person who is involved with monetization of a system likely thinks about day-by-day sales when talking about monitoring. The sets of quality attributes people in different roles are interested are different, but may overlap. Both interpretations are concerned with visibility into a system: what has been happening recently, and what is happening *right now*. For the person involved with operating a system this means having access to historical and current values of properties that affect availability, such as resource usage on individual servers or network usage. For the person involved with monetizing a system this means having access to historical and current values of properties that affect revenue, such as sales, discounts, or effectiveness of advertising. Access to the current values of these properties alone is not enough; only when the values are put in historical perspective, a judgement can be made if the current values are abnormal. When they are considered abnormal, action needs to be taken. For operations, this can translate to adding extra hard disks when hard disks utilization reaches some threshold. For monetization, this can translate to adding specialized discounts when the sales for a particular product decreases. Regardless of the role these people have, they both follow the same steps to finally come to a solution, known as the OODA[1] loop [27]. First, an event is *observed*. Then, that event is put in context, or the observer *orients* itself. Orienting can both mean placing the event in a historical perspective, and comparing it to other events. After that, a *decision* is formulated and finally it is executed, or *acted* upon.

Proper system monitoring is instrumental in observing events in a timely fashion, and providing ample context to put events in per-

1 Observe, Orient, Decide, and Act

spective. In this thesis, the automated system monitoring that we talk about is only concerned with observing events and providing context. Automating the last two steps of the OODA loop – formulating decisions and acting on them – is outside the scope of this thesis.

The set of quality attributes that is relevant in monitoring some system highly depends on the nature of that system. For example, where responsiveness is important for interactive systems, it is less so for non-interactive systems. While responsiveness in itself can be a valuable attribute to keep track of, variations in its value depends on a variety of other attributes. Consider a web application, where requests are handled by a cluster of backend servers. A sudden decrease in responsiveness can be caused by a sudden increase in request volume. It can also be caused by an outage of one or more backend servers, resulting in all requests suddenly being handled by less capacity. It can also be that new software was recently deployed, or software configuration was modified, negatively impacting responsiveness. The list of possible causes for a decrease in responsiveness is endless. To be able to find out what the cause of such an event is, one needs as much context as possible to put the event in perspective to try to find out and explain the cause of the event. Only with enough context and intricate knowledge of the system that is monitored can abnormal values for attributes be explained. Where responsiveness is used here, other quality attributes can be substituted and the requirement for enough context remains the same.

A sudden spike in responsiveness of a system can be seen as a failure. Any unexpected value for a quality attribute can be seen as a failure. Failures are not necessarily black and white, or working or not working. The different types of failures are discussed in the next section.

2.2 TYPES OF FAILURES

Failures come in different types and forms. In Byzantine Fault Tolerance [19], failures are classified as being either failures by omission or failures by commission. Omission failures are failures when the system does not execute what is expected. Commission failures are failures where the system executes something that is not expected [25]. A system being unavailable is an example of a omission failure. A system that responds incorrectly is an example of a commission failure. Aside from these *Byzantine failures*, we also see failure to execute *within a certain amount of time* as failure. A single request for a system not returning within the allotted time is therefore another type of failure.

While the impact of these failures greatly differs, they all violate a specific quality of service constraint. We can define such failures as the *inability to meet a quality of service constraint*. In the first example,

this constraint can be formulated as: the system must be available. In the second example, this constraint can be formulated as: every request must result in a corresponding correct response. In the third example, this constraint can be formulated as: every request must return to the requester within 100 milliseconds. It depends on what the quality of service constraints for a system are, before it can definitively be said if a system is experiences failure or not.

Also important to consider in determining whether or not a system is experiencing a failure or not is the duration that it happens, or the rate at which it happens. When we take unavailability, a system can be required to be available 99.99% per week. This means that every week approximately 1 minute of unavailability is allowed. If the system becomes unavailable, it depends on this constraint whether or not the unavailability is considered to be a failure or not. Similarly, it is unlikely that a single request failing to return within 100 milliseconds should be interpreted as a failure. However, when a significant percentage, say 10%, of the requests fail to return within those 100 milliseconds, it is more likely that this should be interpreted as a failure. A short, or single instance of a failure to meet a quality of service constraint is unlikely to be a problem in itself. Rather, the duration of a violation of a constraint, or the rate of violation of a constraint is more likely to be indicative of a failure. Therefore, the previous definition can be refined to be the inability to meet a quality of service constraint *for some period of time*, or *at some rate*.

There is a wide variety of causes for any type of failure. To illustrate this, we discuss several examples. System failures can be caused by failures in the computer hardware that is used to run the system. Different components failing can result in different system failures. For example, mechanical hard disk spindles can wear out and cause hard disk failure, or a cooling failure can cause the CPU to overheat. These are typical hardware failures that result in omission failures for the system; it no longer executes what it is expected to execute. Other operational examples include power outages caused by natural disaster, a third party accidentally shutting down power, or a backup generator running out of fuel. Lastly, human error may cause systems to fail not only only the hardware level but also on the software level. Authors of any hardware or software component may have forgotten to implement an edge case scenario, that is only triggered every few years. There is no way to predict these kinds of failures, and it is only a matter of time before they happen. These failures are all examples of omission failures, where as a result a subset of the system experiences an outage. Another type of failure that should be considered is the case where the system is no longer accessible for external users, but seems to be fully functional otherwise. This can be caused by either a high load on the system itself blocking out some of its users, or by (intermittent) network failures. This is just as important from

the users point of view as any other failure. Commission failures can for example be caused by a spontaneous bit flip in one of the server's internal memory [32]. Arguably, such a failure is worse than a failure by omission since it may result in users being presented with incorrect data, or data that belongs to other users. Also, such failures that corrupt the state of a system may propagate through the system when they are not caught. Propagating corrupt data throughout a system may result in a cascading failure, taking an entire system down (a failure of this type happened to Amazon S3 in 2008 [2]).

2.3 TYPES OF MONITORING

Because it is impossible to prevent every possible failure, it is arguably less important to minimize probability of failure, and more important to have plans to resolve failures and minimize outage duration.

To be able to detect imminent failures, a system developer or administrator needs to keep a close look on the system's internals, the platform it runs on and the network it is attached to. When failures are detected soon after they happen, the duration of an outage can be minimized. When a failure can be detected before it happens, an outage may even be avoided.

Periodically checking if a system is reachable for its users is one of the ways an administrator can set up monitoring. However, with this data alone, it can be hard to find the root cause for failures after they have happened. Other system properties can be much more valuable in deducing causality. Think of properties such as CPU usage, memory usage, or network usage. When these properties are available over time, in combination with the binary property of reachability, an administrator might conclude that his systems have run out of memory or that network bandwidth is insufficient. Following this line of thought, we can say that the more properties are being monitored, the better a system administrator is able to detect causality in the event of failure, or even is able to prevent failure in the first place.

There are several different types of monitoring that can be applied. Only after establishing what these different types are, how they work and what their benefits are, can we determine how a system is best monitored. For example, some system administrators may like to continuously watch a computer screen showing the output of the `top(1)` tool. Others rather receive an email when a periodic check failed. While this is a simplistic example, these are both techniques to monitor a system, and can be identified as *system monitoring*.

2.3.1 *Interactive/Non-Interactive*

The first distinction that we can make is between interactive and non-interactive monitoring systems [33]. System failure is not a binary condition; failures may be transient. Transient failures on the software level may be caused by sudden peak load, buggy software, or resource exhaustion, to name a few. Because transient failures can come and go at a moments notice, a small and static set of system properties that is sampled every 5 seconds may not reveal enough information to find the root cause for a failure. Rather, when a transient failure is observed a system administrator or application developer wants to introspect a larger number of properties, sampled at a higher frequency, to be able to more quickly determine the root cause. monitoring systems that facilitate drilling down in real-time data can be said to fall in the interactive category.

Next to making data available in real time, the possibility to drill down to individual processes, their resource consumption and internal activity makes this category of monitoring tools invaluable for resolving transient failures. Tools in this category are typically started by administrators when they are needed instead of running continuously. The reason for this is that sampling a large set of properties at a high frequency can come at a high cost in terms of resources. While it may be acceptable to put a temporary load on a system for immediate monitoring purposes, imposing such a load permanently can be too costly.

Opposite to interactive monitoring is non-interactive monitoring. This term covers monitoring systems that instead of being run adhoc and by administrators, run constantly and without intervention. These tools typically sample and store values obtained from a static set of properties periodically, to be inspected at some time in the future. The properties that are sampled and stored by non-interactive tools usually are the primary source of information to detect failures or failure-like behavior. Consequently, administrators who observe such values using non-interactive monitoring systems may then be urged to dig deeper and use interactive tooling for further investigation.

Systems not necessarily belong to the interactive or non-interactive category exclusively, and can perfectly both provide interactive insight in real-time data and non-interactive insight in historical data.

2.3.2 *Active/Passive*

The next distinction to make is one between active and passive monitoring systems. Tools that fall in the active category can be thought to be focused on externalizing the state that they monitor. State that is expected does not need to be externalized; operators of a system

don't need to be notified when the system operates as expected. However, unexpected state does need to be externalized. When a state is observed that is not expected, an active monitoring tool acts on it. Acting on unexpected state can range from executing a corrective measure to move back to a desired state, to paging on-duty system operators. As an example, consider a cluster of web servers fronted by a load balancer. If a web server process on one of the web servers unexpectedly quits, the corrective action can be to restart it. If one of the web servers in the cluster experiences an outage, the corrective action the monitoring system can take is to remove that server from the configuration of the load balancer. As long as the total number of servers in the cluster is large, and the load can be handled by the remaining servers, this does not necessarily require on-duty operators to be paged. If more servers in the cluster experience an outage, and the capacity margin to handle extra outages becomes dangerously low, this likely requires immediate action from on-duty operators. It depends on the type of system if corrective measures *can* be taken, and also if they *should* be taken. Regardless of what the externalizing action of this type of monitoring system is, as long as some action is taken when an unexpected state is observed this type can be categorized as an active monitoring system.

A superset of the category of active monitoring systems is the category of passive monitoring systems. This category of monitoring systems does not necessarily make a distinction between expected and unexpected state, and only observes state. It is a superset because externalizing observed state can be done additional to observing state. Systems in this category generally persist observed state and allow it to be queried, and with that allow other tools to make interpretations of both historical and real-time state.

In this distinction systems only in the passive category tend to be more focused towards tracking state over a longer term. Longer term data can be used to perform trend analysis and make pre-emptive scaling decisions if needed. On the other hand, systems that also belong to the active category tend to be more focused towards tracking state in the short term. With short term data it is easier to uncover immediate issues, that in some cases may be actionable for the system itself.

An example of an active monitoring system is Nagios [26]. Not only is it able to take corrective action, it also tracks observed state over time and allows its users to visualize it. An example of a passive monitoring system is Ganglia [14]. It allows its users to visualize historical and near real-time data, but cannot judge if observed state is expected or not and take corrective action.

2.4 TYPES OF DATA TO MONITOR

Having discussed different types of monitoring systems, it is important to take a look at the different types of data that can be monitored.

Operating systems expose a wide variety of metrics that are worth keeping track of. Many Unix systems have a virtual file system `/proc` that exposes many kernel level parameters regarding CPU usage, (virtual) memory usage, hard disk usage, network usage, etcetera. Because of the vast quantity of available kernel level parameters, it is not always necessary to keep track of every single one of them. Besides the numeric parameters exposed via `/proc`, it is also useful to keep track of log messages emitted by the kernel. In failure scenarios this can provide invaluable leads to tracking down root cause, as they usually contain stack information that can influence the direction of an investigation. The same rationale holds for application level metrics and log messages. While the operating system alone can point in the right direction when investigating a failure cause, parameters specific to an application give much more context regarding what it was doing and how it was doing at the time of a failure.

Examples of types of data to keep track of include:

NUMERIC
: The snapshot of some value at any given time. For example, the memory usage, or hard disk usage on a system, or the number of connected clients to a web server. Another example is counters, such as the total number of packets sent or received by a network interface since the server was started.

DISTRIBUTION
: The statistical distribution of some population. When the corpus of data is too large to sample, it can be useful to only keep track of its distribution. For instance, the 50th, 90th, and 99th percentile in a web server's request latency distribution. Where keeping track of every individual request may be too costly, the distribution of request latencies can be used to determine if a server is healthy or not.

TEXT
: Textual data can be anything. It can be logs emitted by the kernel, or stack traces or debug logs emitted by an application.

Of these types there is none that is by itself enough monitor a system. Arguably, all types are needed together to form an impression of how a system is behaving, if it is healthy or not, and lastly, to be able to track down the cause of unexpected or unhealthy state *when* it happens.

2.5 CONTEXT: CLOUD FOUNDRY

The system that sparked the work described in this thesis is Cloud Foundry [6]. It is an open source Platform-as-a-Service (*PaaS*) created and maintained by VMware. Where an Infrastructure-as-a-Service (*IaaS*) allows its users to stop worrying about their infrastructure (e.g. hardware, network, data center, etc.), a PaaS allows its users to stop worrying about their platform (e.g. operating system, language runtime, data services, etc.). Cloud Foundry takes care of managing infrastructure to facilitate running applications in a dynamic mesh of resources. The system is not explicitly tied to one, or any, infrastructure layer. The components that make up Cloud Foundry can all be co-located and run on a single machine, or can be distributed over a vast number of physical machines in a data center. The source code of Cloud Foundry is publicly available to allow people to run their own instance and contribute code back to the source tree. Additionally, VMware runs Cloud Foundry commercially on `http://cloudfoundry.com/`.

Web applications are the unit of trade in Cloud Foundry. A single Cloud Foundry deployment can server multiple users, who in turn can deploy many applications. Every application can optionally be bound to a number of data services. Cloud Foundry can bind applications to a number of data services, including MySQL, MongoDB, and Redis, and can be extended to support more. Provisioning these data services is transparent to the user. For example, if a user wants to bind his application to a MySQL database, he tells the system to do so and his application is provided with credentials to access a MySQL database. The process of binding to other data services is similar. Applications are scaled by increasing or decreasing the number of instances they run. Instances are dynamically distributed over the infrastructure pool. Application traffic is automatically balanced over available instances by the pool of HTTP routers managed by the system. These HTTP routers are the entry point for all traffic to all applications, and make sure application traffic is routed to available instances.

2.5.1 *Components*

To give an impression of the way applications and data flows through Cloud Foundry, we give a short overview of the core components in the system and how they interact.

ROUTER
: Cloud Foundry's routers are HTTP routers that proxy requests to the applications they are intended for, by mapping URLs to applications. If there are multiple available instances for a single application, the router makes an effort to fairly distribute

requests over these instances. As new instances are started, and existing instances are stopped, the routing map is updated such that requests can always be served by live instances.

CLOUD CONTROLLER
Like regular applications, the cloud controller is an HTTP endpoint whose traffic flows through the routers. It exposes the API for users to interact with the system. Applications are uploaded to cloud controllers, application metadata is updated via the cloud controllers, services are bound to applications via the cloud controllers, and so forth. To fulfill API calls the cloud controllers may interact with other components in the system. For example, when an application needs to be started a cloud controller starts a discovery phase to find available resources, and instructs the node with available resources to start the application.

DEA
The *Droplet Execution Agents* (DEAs) together form the resource pool that is used to run application instances. DEAs keep track of their own resource usage and depending on the amount of available resources it may or may not respond to discovery requests from the cloud controller. Upon starting an application it notifies the router to update its routing map, such that the instance can participate in handling the application's load. Similarly, when an instance crashes it notifies the router to remove the instance from its routing map, such that no new requests are forwarded to this instance.

HEALTH MANAGER
To ensure that for any given application the number of instances that are supposed to be running is equal to the number of instances that is actually running, the health manager continuously monitors the system's state. It does so by formulating an expected state from the application metadata stored with the cloud controllers, and comparing this to the actual state observed by heartbeats that are broadcasted by the DEAs. When there are too few instances running for a particular application, the health manager instructs the cloud controller to start extra instances. Similarly, when there are too many instances running for a particular application, the health manager instructs the cloud controller to stop the superfluous instances. Categorizing this component in terms of the different types of monitoring systems discussed earlier in this chapter, the health manager is a *non-persistent and active monitoring system*.

Internal communication happens over a publish/subscribe message bus named NATS [7]. It is both used to issue requests from one

component to another, and to broadcast messages to all components in the system.

2.5.2 *Monitoring Challenges*

While the Health Manager already tracks the state of the system, and performs corrective actions when the observed state is incorrect, this does not solve all problems. For example, only keeping track of current state does not give any insight into how often applications need to be restarted, or how the distribution of load over the DEA pool changes over time. Having access to historical state data creates a new domain for data analysis, and data that can be made visible to users of the system. Following the categorization of monitoring systems discussed earlier in this chapter, the type of monitoring system that provides this is a *persistent and passive monitoring system*.

Conventional monitoring systems of this type often rely on static network topologies. The system is configured once, to fit this topology, and provides access to data based on that topology. Usually, kernel and application metrics, or streams of logs are made available on a per-host basis. Some monitoring systems can aggregate data from multiple hosts into one or more global metrics that give an overview of system-wide state.

This type of monitoring tool is inappropriate for use in Cloud Foundry. While the network topology of the infrastructure that run Cloud Foundry may be static, the nature of applications that Cloud Foundry runs is highly dynamic. Applications can move through the system at a fast pace: they can be updated often and be scaled up and down in a matter of seconds. To get a system-wide overview of the status of a single application may require using data points from every single one of the system's nodes. Data points that give information about state and performance of an application may originate from any node in the system. Additionally, the data points can no longer be identified using the host alone, because every host is equally likely to run every application. Aggregation of data points on a per-host basis is therefore not enough; data points may need to be aggregated on a per-user, per-application, or per-URL basis. The dynamicity of the data points that are used to track state of an application throughout the system asks for a thorough evaluation of the data model, approach to storage, and querying capabilities.

3 RELATED WORK

This chapter gives an overview of a number of commonly applied tools for system monitoring. Because a distinction between collecting and monitoring logs and collecting and monitoring metrics is made, we also make that distinction when discussing different tools.

The protocols and tools that we put forward in this chapter is by no means an exhaustive list. The problem domain concerned with monitoring logs and metrics is so common that many companies build their own tools that exactly fit their needs, instead of adopting existing tooling. Arguably, the work that we put forward in this thesis does the same.

3.1 LOGS

A widely used protocol that is used to deal with logs is the *syslog* protocol. It specifies a format to transport log messages between processes living on the same machine or living on the same network. Processes can use the protocol to send their log messages to one or more syslog daemons, that in turn can act on these messages by writing them to a file, sending them via email, forwarding them to another syslog daemon, etcetera.

Its wide use can be attributed to it being usable through standard libraries such as `glibc` [36], and the general availability of syslog daemons on Unix-like systems. It was first defined in 2001, in the informational RFC 3164 [22].

The attributes of a log message that the protocol defines include the *facility* that generated the message, and the *severity* of the message. These attributes allow system administrators to define policies for log messages depending on their source and importance.

For instance, log messages that are generated by the operating system and are marked with a high severity may require immediate attention. On the other hand, log messages that are generated by some user space process and are marked with informational severity may not even be stored on disk.

The facilities that can be used to specify the source of a message are predefined by the protocol. They include the kernel, the email daemon, and security related processes. Predefined severities vary from debug messages (the least important), to emergency messages (the most important). In between are severities for informational messages, warning messages, alert messages, and others.

The default transport layer for syslog is UDP[1]. This choice contributes to the simplicity of the protocol. Because UDP is stateless, producers of log messages can hand off the messages they generate by sending them to a pre-known address at a pre-known port (typically port 514) without requiring negotiation. This means that messages are lost if no daemon is listening on this port.

Besides the required facility and severity of a log message, the protocol also requires log messages to contain a timestamp and the host name of the originating machine. Because log messages may be forwarded through a number of hosts on a network, this information is required for tracing a message back to its producer and the time it was produced. The remainder of a message is free form data according to the protocol. However, many implementations of the protocol use strict formatting of this free form data to encode more information about the context in which the message was produced. Extra context may be added by means of process IDs, thread IDs, user IDs, or identifiers specific to the process that allow an administrator to retrieve context not accessible through syslog.

There are several implementations of syslog daemons. Examples of widely used implementations are `syslogd`, `syslog-ng`, and `rsyslog`. These implementations allow the creation of complex forwarding topologies, where after logs are created, they are forwarded through a number of intermediate syslog nodes before being written to disk, or discarded.

Only using syslog to capture a system's log data does not result in a data set that allows easy querying. The individual log lines are unstructured strings and are stored in flat files. These problems is what *logstash* [21] attempts to solve. It runs as a daemon that can receive log data from a variety of inputs, including syslog and files. After receiving logs, they can be transformed according to a number of steps, such as serialization and deserialization, regular expression matching, and more. Once transformed, possibly into a more structured representation of the data, logs can be forwarded to variety of outputs. The possible outputs include regular files and a number of databases.

Another example of a log management tool is *Splunk* [35]. This is a commercial product that can index large amounts of log data, and allows querying and analysis.

3.2 METRICS

Next to the domain of tooling that is primarily concerned with moving log data around, there is the domain of tooling that is primarily concerned with sampling, collecting, storing, and aggregating metric data. This domain deserved to be explicitly called out next to the log

1 User Datagram Protocol

data domain, because instead of spuriously receiving log data, metrics sampling can have a predefined cadence. Tools in this domain commonly assume metric data to be numeric, and can provide a number of ways to work with this numeric data. For example, historical numeric data can be used for trend analysis, real-time numeric data collected from multiple sources can be subjected to statistical analysis to detect outliers, and so forth.

An example of a tool that can be used for metrics sampling, collection and visualization is *Ganglia* [14]. It runs a collection agent on every node that monitors a configurable set of metrics over time. It forwards the metric data that it collects to upstream nodes in a hierarchical topology. The root node of this topology receives data from all nodes in the system and provides a single view of this data. Ganglia uses *RRDtool* [31] to store the metrics it collects.

The *RRD* in RRDtool stands for Round Robin Database, which explains how it persist metric data to disk. Data points are stored in round robin archives (commonly abbreviated as RRA's), or *circular buffers*. Because this type of metric data is periodically sampled, the number of data points only grows. As new data points stream in, existing data points are slowly aging. The round robin archives leverage the fact that data points age, and at some point in time are no longer interesting and can be discarded. This enables them to have a constant size, which in turn imposes a fixed upper limit on their file size. Every round robin database can hold multiple round robin archives, where every archive stores a different time interval. For instance: a high resolution archive can store data points for the last 10 minutes, a medium resolution archive can store data point for the last 2 hours, and a low resolution archive can store data points for the last 24 hours. Such a configuration allows people who are interested in both high resolution data and day to day data to use the same graph, while the on disk size of the archive remains constant. RRDtool requires every archive to have an associated function that consolidates multiple high frequency data points into a single lower frequency data point. Common examples of such functions are `MIN`, `MAX` or `AVG`, for minimum, maximum or average respectively.

A similar approach to storing metric data – or *time series* data – is used by *Graphite* [15]. Unlike Ganglia, Graphite does have separate metric collection agents and requires its user to provide the metric data himself. By doing this, it moves complexity to its user (the data needs to be massaged such that Graphite can interpret it), but at the same time is more flexible. Graphite is not distributed itself. It runs on a single node and acts as a sink for metric data; all its users send their data and Graphite stores and visualizes it.

These and other metric storage and visualization systems generally use simple string identifiers to tell different time series apart. Such identifiers are usually made up from the host name of the node they

are sampled, and some string containing what kind of data they hold (e.g. `server1:memory` or `total:memory`). In a dynamic environment such as Cloud Foundry there may be a large number of time series that are not necessarily can be pinned to a single node or single instance. Rather a time series for an individual instance may contribute to aggregate time series for the node it is running on, for the application it is a part of, for the user that owns it, etcetera. To our knowledge, efficiently representing time series data in a way that it can be aggregated along multiple axes is not possible using existing open source log storage or metrics storage software.

4 ANALYSIS & DESIGN

4.1 DATA MODEL

Existing tooling for logging is often opaque in terms of resources being logged and monitored, and the data that is involved. Logs are per-host streams of plain text, and derived metrics are string-identified time series. Because of the lack of semantical information, it is rarely possible to mix and match data from different sources without first manually massaging it. This section discusses positive and negative traits of this lack of semantical information, proposes alternatives and discusses their trade-offs.

4.1.1 *Logs or metrics?*

Collecting plain text logs and collecting metrics is often treated as two separate problems. We have seen this in the chapter discussing related work; some tooling explicitly collects logs over time, and some tooling explicitly collects metrics over time. Rather than treating the two as different problems, we like to see them as one and the same, being *system visibility*.

If only tooling to collect metrics is used, valuable context about the nature of those metrics may be lost. Consider a web shop that keeps track of the number of purchases per minute. If this number suddenly decreases, it is very valuable to be able to quickly find out why this is happening. The number of purchases per minute doesn't tell a lot in isolation; it is merely an indicator that something is wrong. Having access to plain text logs can be instrumental in finding out the cause of the sudden decrease in purchases. Maybe a third-party payment provider is suffering from an outage that makes purchases fail. Maybe the application running the web shop has crashed, logging a stack trace of the crash. Maybe the database backing the web shop is no longer available. While these examples can all be translated into a specific metric, the probability that application developers included a number of `print` statements in the application code is higher than the probability of a specific metric being instrumented for every edge case. Metrics that could have quickly identified the cause of a failure are often added *after* the failure happened. Having access to more rudimentary system visibility facilities such as plain text logs can greatly assist in quickly finding the cause of a failure.

Conversely, if only tooling to collect plain text logs is used, a brief summary of the status of a system is hard to create and maintain.

In the same example of the web shop, only keeping an eye on the plain text logs and not the metrics that can be derived might make a sudden decrease in the number of purchases go unnoticed.

We believe that both approaches to achieving visibility into a system are complementary, and that the value of a combination of the two is greater than the sum of its parts.

4.1.2 *Dimensionality*

Traditionally, logging and monitoring systems capture data over time. The data being logged or monitored can be a simple numeric or string value, or a complex composite type with nested numeric and string values. At its core, though, the semantics of this data is opaque to the logging system; it is merely a chunk of data where interpretetation is left to the consumer of the data. Logging and monitoring systems can keep many of these opaque *data over time* streams, where these streams can be identified by some string identifier. Individual streams can tranditionally be traced back to a single source that emitted or produced the data that it contains. For example, a syslog implementation can write logging data to a file identified by the originating *hostname* and *logging facility*. Or, Graphite can keep the number of requests per second for a specific HTTP frontend on a specific machine. These are examples where opaque streams of *data over time* are identified by some arbitrary string.

This string can contain important additional data that can help interpreting the data that is stored in the stream. In the syslogd example, this string contains a hostname and a logging facility. In the Graphite example, this string contains an identifier of the HTTP frontend, and the fact that the data it identifies is the number of requests per second.

However, the additional data that the string holds must be valid for the entire lifetime of the stream. If the hostname in the syslogd example changes, the string identifier of the stream changes. Similarly, if the identifier of the HTTP frontend in the Graphite example changes, the string identifier of the stream also changes. In other words: the string identifying a stream changes when metadata that uniquely identifies that stream changes. If this happens, the streams appear to be different without any relation to each other, unless an explicit effort is made to preserve this relationship.

This is not necessarily a bad thing. Because these identifiers can be any string, and therefore can contain a serialization of any metadata, the string can be made arbitrary rich. Besides the properties mentioned before, it can contain more information such as the kernel version of originating machine, the process ID of the originating process, and so forth. All properties that do not change over time can be embedded in this identifier, since only properties that not change over

time can uniquely identify a single stream of logging or monitoring data.

In this model, all data can be said to be 3-dimensional. First, there are the actual data points. These can be arbitrary data: simple numeric values or complex composite types. Whatever they are, they make up the first dimension. These data points are captured over time, which makes up the second dimension. Every 2-dimensional plane of data over time is then identified by a string, which makes up the third dimension.

4.1.3 *Encoding more dimensions into one*

On a per-system basis, the string identifier of a stream can possibly be deconstructed into more dimensions. If the identifier encodes the originating hostname, for example, this could be called out as separate dimension. Or, the logging facility could be called out as separate dimension. The problem with this is that this can only be done by convention, and on a per-system basis. If syslog decides to encode this data by prefixing it on every log line, in pre-specified order, separated by whitespace, the consumer of the data needs to know this in order to extract it. A different system may use a different convention to encode extra dimensions, and the consumer needs to adapt.

What follows from this, is that it is not trivial to combine data logged or monitored by different systems. Because the way that the source hostname is encoded in the string identifier can differ from system to system, correlating data logged or monitored by different systems requires a manual step to align these. For example, the syslog protocol explicitly calls out the hostname of the machine that produced the data, whereas Graphite uses an arbitrary point-delimited naming scheme where this is up to the user. Correlating data from these systems cannot be done automatically, without adding explicit semantics to the Graphite naming scheme.

4.1.4 *Adding more context*

The problem of being restricted to only two data dimensions (recall that the third dimension is time) becomes worse when an attempt is made to capture more context for every log line or sample.

4.1.4.1 *More information in the data points*

Going back to the syslog example, consider it being used to capture log messages from an HTTP server. A widely used logging format for HTTP servers is the Common Log Format [8]. This log format specified a sequencing and formatting style for various properties involving an HTTP, resulting in a human readable line of text. See

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] \
  "GET /apache_pb.gif HTTP/1.0" 200 2326
```

Figure 1: Log entry formatted using the Common Log Format. Example taken from [20]. *Note: the \ denotes line continuation.*

Figure 1 for an example. The properties that it encodes include the client IP address, the date and time, the HTTP request method, the requested path, the response status code, and the number of bytes that make up the response. While this format may encode enough properties for some, it lacks for others. Examples of properties that are not included are the user agent [1], the hostname that the client makes its request for, or the referring URL [2]. Because the Common Log Format cannot be used to log additional properties, it needs different adaptations or versions to make this happen. The more versions of a log format there are, where different versions have only subtle differences, the more difficult it becomes to tell one format from another. Interpretation of the data they encode becomes harder, as ambiguity increases.

However additional context for an HTTP request is encoded, this approach adds data to every point in the *data dimension*. The stream dimension and time dimension remain unaltered, while the amount of information per data point increases. This additional context may be perfectly decomposable into more extra dimensions, but the model that I have described so far only allows to store more information per data point. Note that the number of data points does not change, only the information per data point. The interpretation of the extra information per data point is left to the consumer of the data. A `syslogd` process cannot derive any information from the data points, without explicitly being instructed to do so with rules to extract this information.

4.1.4.2 *More information in the string identifier*

Another approach to add data is to include it in the dimension of the string identifier. Going back to the example of Ganglia being used to monitor the number of requests per second, the data dimension cannot be used to store more information; it must be a numeric value. This means that adding extra context for every data point is only possible by extending the string identifier. Because the string identifier identifying a stream of log entries or samples uniquely identifies a *single* stream, regardless of the time the log entries or samples were added, changing it means increasing the number of streams. The properties that are called out in the example above, talking about the

1 Client identification. This usually contains the name and version of the web browser, and the name and version of the operating system.

2 Uniform Resource Locator

Example identifier	Worst-case number of streams
`rps-srv1`	1
`rps-srv1-ip:192.168.1.1`	#A
`rps-srv1-path:/home`	#B
`rps-srv1-ip:192.168.1.1-path:/home`	$\#(A \times B)$

Table 1: Example identifiers and worst-case number of streams when the substring representing the client IP address or requested path is extrapolated.

Common Log Format, also are interesting in this example. If, for instance, there is interest in knowing the number of requests per second *per client IP address*, this is only possible by adding an extra stream *per client IP address*. These extra streams can be constructed by including the client IP address in their indentifier. This means that the worst case growth factor for the number of monitoring streams in this system is equal to the number of IP addressess: 2^{32} (or 2^{128} for IPv6). While it is unlikely that every possible IP address will make at least one request, it is perfectly possible. The total number of requests per second can be derived from this potentially large number of individual streams. The total number of requests per second is equal to the sum of the number of requests per second per IP address.

Consider that there is interest in knowing the number of requests per second per requested path, and there are 1MM possible paths for the website that is served by the HTTP server. In the worst case, every path is requested at least once, and because there is a single monitoring stream per path, this requires 1MM monitoring streams. When these two interests are combined – interest in the number of requests per second *per client IP address per requested path* – the worst case number of monitoring streams is a multiplication of the two worst case numbers.

This is expected, because every dimension that is added needs to be encoded in the stream's identifier. Suppose the set of possible client IP addresses is set A and the set of possible request paths is set B, the worst case number of monitoring streams is the cardinality of the product set $A \times B$. An example is given in Table 1. Every additional discrete dimension – for discriminating requests per second in this example – is another input for the product set, and thereby causes the worst case number of monitoring streams to grow by its cardinality.

Adding more information to the data points themselves does not increase the cardinality of any of the dimensions. Adding more information to the stream identifier does increase the cardinality of that dimension. In practice, the individual monitoring streams in this model may map to individual files, while their contents may map to a series

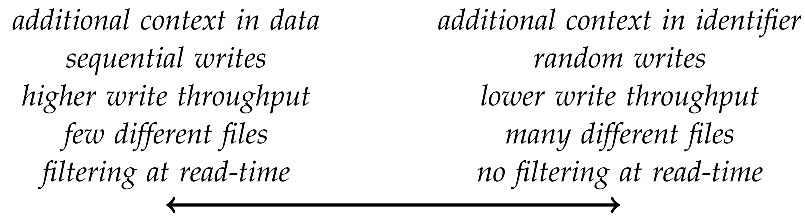


Figure 2: Trade-off between read-time and write-time cost

of (time, data) tuples, where time is monotonically increasing. This approach is very common for logging systems, such as `syslogd`.

Applying the previously discussed example to this materialized version of the model results in the following. When more information is added to the data points, the number of streams – thus the number of files – remains constant, yet the size of their contents will increase. When more information is added to the identifiers, the number of streams – thus the number of files – grows with respect to the cardinality of the dimensions they encode, while the size of their contents will decrease.

4.1.4.3 *Consequences and trade-offs*

The consequence of the first option is that the complexity of extracting data from the stream is moved to read-time. In the example, figuring out how many requests in a certain time frame came from a specific client IP address will require scanning the entire file, and filtering relevant entries. The cost at read-time is therefore potentially large, while the cost at write-time is kept low because the logging system sequentially appends new data to a single file.

The consequence of the second option is that the complexity is moved to write-time. In the example, data points are appended to their respective streams. Because many data points may end up in many different files, this puts a lot of pressure on the file system and the backing disk. Concurrently writing to many different files is likely to result in many random disk writes, which is considerably more costly than sequential disk writes. At read-time, however, the consumer of the data only needs to locate the file of interest to extract data. No extra filtering is needed because this has already been done by writing data to different files.

This is not a binary trade-off; some information may be encoded in the stream identifiers, while other information may be encoded in the data points themselves. The extremes of this trade-off are illustrated in Figure 2.

Where on the trade-off scale this solution will be depends on how it will be queried, and how it will storage and retrieve data. Only

after these aspects are discussed can we determine what place on this scale the solution will take.

4.2 QUERYING

This solution should support two modes of operation.

- It should allow data to be extracted after-the-fact. This means that after log data has been received by a logging agent, it should be possible for an external client to retrieve a subset of the the log data after some amount of time.
- It should allow data to be extracted in near real-time. This means that while log data is being received by a logging agent, it should be possible for an external client to subscribe to a subset of the log data. The logging agent should then forward all data is receives, that is contained in the subset the external client it interested in.

These two modes of querying log data should not only be usable in isolation, but also in combination. For instance, an external client should be able to subscribe to a subset of the log data that has been received since 10 minutes. The logging agent (or multiple logging agents) should then serve up the data that has already been written to disk, and switch to a near real-time subscription when this completes.

4.2.1 *What about* `grep` *and* `tail`*?*

Querying log data after it has been stored can be done with the `grep` utility. It allows its user to specify a regular expression and one or more file names, and prints the lines in the files that match the regular expression. This is a well understood approach, because of the ubiquity of the utility, and the wide use of plain text files to store log data. It can also be easily combined with a near real-time subscription to log data as it is generated or received by using the `tail` utility. This utility takes one or more file names as input and prints lines that are added to those files as they are added. The output of `tail` can be sent to `grep` by means of a pipe to get a near real-time subscription to log lines that match the specified regular expression. An example is given in Figure 3. While this can be a very powerful technique for extracting data, it doesn't take into account the semantics of the log lines that are scanned.

First, these tools work on inputs with line-delimited content. Every log line is delimited with a newline character (`LF` or `\n`) to separate it from following lines. This means that the data itself cannot contain this character, or it will be misinterpreted. If log data contains newline characters, they should to be escaped to ensure correct interpretation.

log/httpd.log:

```
127.0.0.1 - - [13:55:36] "GET /p1 HTTP/1.1" 200 100
10.0.11.1 - - [13:55:37] "GET /p2 HTTP/1.1" 200 100
127.0.0.1 - - [13:55:38] "GET /p3 HTTP/1.1" 404 100
10.0.11.2 - - [13:55:39] "GET /p4 HTTP/1.1" 404 100
127.0.0.1 - - [13:55:40] "GET /p5 HTTP/1.1" 500 100
10.0.11.3 - - [13:55:41] "GET /p6 HTTP/1.1" 500 100

$ grep " 127.0.0.1 " log/httpd.log
127.0.0.1 - - [13:55:36] "GET /p1 HTTP/1.1" 200 100
127.0.0.1 - - [13:55:38] "GET /p3 HTTP/1.1" 404 100
127.0.0.1 - - [13:55:40] "GET /p5 HTTP/1.1" 500 100

$ tail -4 -f log/httpd.log
127.0.0.1 - - [13:55:38] "GET /p3 HTTP/1.1" 404 100
10.0.11.2 - - [13:55:39] "GET /p4 HTTP/1.1" 404 100
127.0.0.1 - - [13:55:40] "GET /p5 HTTP/1.1" 500 100
10.0.11.3 - - [13:55:41] "GET /p6 HTTP/1.1" 500 100
...

$ tail -4 -f log/httpd.log | grep " 127.0.0.1 "
127.0.0.1 - - [13:55:38] "GET /p3 HTTP/1.1" 404 100
127.0.0.1 - - [13:55:40] "GET /p5 HTTP/1.1" 500 100
...
```

Figure 3: Example use of `grep` and `tail`.
(these lines use an abbreviated version of the Common Log Format)

After extracting the data, the escaped newline characters should again be unescaped again to preserve the original data.

Second, while every log line encodes a number of different dimensions, these tools are completely oblivious to this. Every line is just a series of characters, followed by a newline. All information that is encoded in these lines needs to be decoded by the consumer of the data. This also means that filtering the lines can be hard when the tools used for filtered are not able to decode the data themselves. For instance, if in the example showed in Figure 3 the requested path contains the string `" 127.0.0.1 "`, `grep` will match those lines as well as the lines containing that IP address. Or, when the lines are filtered for the status code 200, unintentional matches may include lines with a response body size of 200 bytes, or lines where an IP address octet equals 200. There are many more ambiguities when properties of a log line are encoded without context.

Related to this concern is that time is not explicitly called out. The inputs to these tools are opaque streams where any filtering – including filtering on time – needs to be specified by the consumer of the data. A widely applied technique for being able to filter log lines in a certain interval is to include the offset of that interval in the file name storing the log lines. Then, for every interval that passes, the old log file is closed, and a new one is opened for writing (known as *log file rotation*). How the interval is chosen, and how time is encoded in the file name varies between logging systems and configurations. It may

use an interval equal to a full day, and encode only the date in the file name (e.g. `log/httpd.2000-10-10.log`). Or, it may use an interval equal to 5 minutes, and encode the offset as integer timestamp (e.g. `log/httpd.971211300.log`, when the integer timestamp is the number of second since the Unix epoch – January 1, 1970 at midnight – in the UTC timezone). Either way, the task of scoping a query to a certain time interval is left as an excersise for the consumer of the data.

4.2.2 *Goals*

The problems that exist for querying log data without contextual information should be addressed in this solution.

First, the problem that line delimiting illustrates – required encoding and decoding of "special" characters – should be addressed. Log data should be allowed to be arbitrary binary data, without requiring an encoding and decoding step when the data is produced and consumed respectively. Log data being arbitrary of size and contents should be transparent to both the producer and the consumer. As a consequence, it may no longer appropriate to talk about log *lines*. For the remainder of this chapter they will be referred to as log *records*.

Second, the ambiguity problem should be addressed. If a consumer of the log data wants to apply a filter, it should be possible to unambiguously specify conditions for any of the properties in any log record. This not only has consequences for the interface towards the consumer of the data, but also for the interface towarda the producer of the data. The producer of log records alone is able to attach semantical information to the log records it creates.

In the example of the Common Log Format, this semantical information is only available implicitly. It depends on the contract between the producer and the consumer that the log *lines* that are produced follow the predefined formatting rules. Only when this contract is strictly followed, the consumer is able to extract the semantical information it holds. The Common Log Format is specific to HTTP servers, so producers of log data that does not originate from HTTP requests will need their own formatting standard to transfer semantical information to the consumer. If executed, this is not only a daunting task, but all producers and consumer must be in perfect alignment in the formatting and interpretation of the log data. Also, it does not solve the ambiguity problem on the level of the logging system. As long as the logging system is unaware of a distinction between different properties of a log record, it cannot make filtering decisions.

These considerations lead to the need for explicit inclusion of semantical information. Only when semantical information is included in the log records can the consumer direct the logging system to filter certain records. For instance, in the example of an HTTP server, the

```
$ query --time="-5s.." --filter="client_ip=127.0.0.1"

t ----------------------> 971211339.000000
client_ip --------------> 127.0.0.1
http_request_method ----> GET
http_request_path ------> /p4
http_request_version ---> 1.1
http_response_status ---> 404
http_response_body_size > 100

t ----------------------> 971211341.000000
client_ip --------------> 127.0.0.1
http_request_method ----> GET
http_request_path ------> /p6
http_request_version ---> 1.1
http_response_status ---> 500
http_response_body_size > 100
```

Figure 4: What a querying the described system could look like.

consumer should be able to filter based on the client IP address, HTTP request method, requested resource, response status code, etcetera. When these properties are explicitly called out in every log record, the logging system is able to make these filtering decisions.

Last, the one property that is shared among all log records is time, and should be treated as first-class. The consumer of the data should not be concerned with selecting which file contains the time interval of interest, or where this file is located. Also, the consumer of the data should not be concerned with how time is stored in the log records, or how to address a certain time interval. Time is an important dimension when talking about log data. The importance of log data often decays over time; the information stored in records that were created *2 weeks ago* is usually less important than the information stored in records that were created *5 seconds ago*. Consumers of the data should be able to easily navigate through log records over time. To enable this, addressing log records by a specific point in time, or by time interval should be possible.

Putting these goals together, invoking a tool that executes a query against log data could look like Figure 4. To illustrate that log lines no longer use naive record delimiters such as newlines, the records are shown as distinct objects. While in practice the text-only output of a command line tool will need to use some kind of delimiting scheme, this is intentionally omitted in this example by using a graphical representation.

The different properties are explicitly called out for every record. This means that it is no longer necessary to use a static formatting

scheme to encode semantical information implicitly. Rather, it is possible to include arbitrary properties and their values just by adding them to the record. This is enough information for the logging system to make decisions regarding filtering or aggregation of log records, as illustrated in the figure.

Time looks like just another property in every log record. The difference with the other properties is that time is mandatory. If it is not specified by the producer of the log record, the logging agent that receives it can choose to automatically add it, or to reject the log record altogether.

It is the primary source for determining causality between records. Other means for determining causality between log records can be added by their producers, leaving interpretation for the consumers, when it is not possible to rely on monotonicity of clocks. For instance, clocks in a distributed system can never be in perfect synchronization. Determining causality in a distributed system by using numeric time alone is therefore not possible.

However, the causality relationship does exist for log records that were created with their time being read from the same monotonically increasing clock.

4.3 storage and retrieval

There are many approaches to storing and retrieving log records. A number of key characteristics that inherently apply to log records need to be kept in mind when discussing the possible approaches, and when the choice for a particular approach is made.

The first characteristic of the data can be found in the property that applies to all log records: time. Log records can be traced back to a single process that was responsible for creating them. They are a reaction to events taking place, and being noticed by this process. The log records that are created following an event can be seen as a materialization of that event. Besides information about the event itself, it can contain additional context that captures the process's state when the event was noticed. After they are created, they cannot change. They provide a point-in-time snapshot, and are therefore immutable. Events can be caused by entities external to some system, such as clients requesting a web page, or entities internal to some system, such as a timer firing every second. Either way, when a process has expressed its interest in being notified when a particular event happens, and it does happen, there will be a point in time where the event is noticed and is picked up by the process. When an event is noticed, it is not possible for this event to be un-noticed at some later point in time. The moment it is noticed, it affects the state of the process that is interested in it, and this cannot be undone. The corpus of events that are noticed over time – and therefore the log records

that they may create – can only grow. Old log records can be deleted after some amount of time, but that doesn't mean they never existed. It only means they are no longer available. The data set containing all log records that were captured for some system can therefore be characterized as a **monotonically growing set of immutable elements**.

Another characteristic can be identified by looking at the form of the data. As is discussed in the previous section about querying, the data that makes up a log record is arbitrary. A record may contain any number of properties, as long as it includes the time the record was created. Properties are identified by some string identifier, and associate to values that can be arbitrary binary data. This may be seen as a 1-to-1 mapping, but there not really is a reason not to allow a property mapping to multiple values. Therefore, instead of seeing it as a 1-to-1 mapping, it can be seen as a set of tuples $(\mathtt{property}, \mathtt{value})$. The only restriction on this set is that it includes a single tuple that holds the time the log record was created. This restriction can be ignored for the sake of simplicity when talking about the structure of the log records. Individual log records can be characterized as an **immutable set of 2-element tuples**.

The next characteristic can be identified by looking at the relation between log records. It can be argued that it is more likely for two log records to be part of the same query result when they are temporally close, than when they are temporally distant. For example, a query more likely asks for *all* records that were created in the last 5 minutes, than for a *random sample* of records that were created in the last 5 minutes. Also, it can be argued that it is more likely for two log records to be part of the same query result when they are spatially close, than when they are spatially distant. Where the temporal relation comes natural for data that is ordered in time, the spatial relation is less natural. The spatial relation bwteen log records (or, recall, sets of 2-element tuples) can be viewed as a measure for similarity. Log records that have more tuples in common can be said to be spatially closer than log records that have fewer tuples in common. This means that it is more likely for a query to ask for records that contain tuples $t_1, \ldots, t_n$, than to ask for records without restriction. An example of this in the context of the HTTP server is a query that asks for all records that contain $(\mathtt{client_ip}, \mathtt{127.0.0.1})$ and $(\mathtt{http_request_method}, \mathtt{GET})$. Or, a query that asks for all records that were created by the same producer. These examples of how log records – or sets of 2-element tuples – can be related show that there exists **temporal and spatial affinity between sets**.

These characteristics should drive the design of the storage and retrieval layer of this solution. The described data can be mapped to a variety of storage techniques and databases, such as flat files, relational databases and key/value databases. Some techniques will be a better fit for the described characteristics than others. The question is:

which technique is the *best fit* for the described characteristics? Which technique can best handle monotonic growth? Which technique can best store and retrieve records without predefined schema? Which technique can best optimize for spatial and temporal affinity? And lastly, which technique is the *simplest*?

4.3.1 *Flat Files*

The first storage technique is straightforward. The use of flat files is the default way to store log data for `syslog`-like systems. While it may seem a contrived example to use as a storage technique for the type of data that is being discussed, it is a surprisingly good fit. For storing log records, the only task of a logging agent that uses flat files to store the records it receives is to serialize them to a format that can be written to disk, and write it to disk. Because log records are immutable, every record is written once and only once. This means that the write operations that the logging agent performs can be sequential and append-only. Once data has been written, it will never be modified or rewritten. The fact that log records that are temporally close together will also be stored close together is an implicit side-effect of writing them sequentially.

Just like `syslogd`-like systems, log rotation can be used to partition files over time. This gives the ability for files with older log records to be moved to cheaper storage for archiving, or to be removed altogether when there is no need to keep them around.

Another aspect that was already discussed in the section about querying, is that partitioning files by time allows log record retrieval to quickly find the files of interest when querying for a certain time interval. The larger the partitions, the more log records need to be scanned to find the start of a particular time interval. Conversely, the smaller the paritions, the fewer log records need to be scanned to find the start of a particular time interval.

A downside to using flat files is that is does not provide a way to index log records out of the box. The only primitive index that is implicitly present is that of time. It is implicitly present because it is facilitated by partitioning files with log records over time. Queries that besides time put extra constaints on the log records that should be extracted require scanning *all* log records. Every log record is then tested against the set of constraints, and included in the result if it matches. This may put an unacceptable load on the system at read-time, if the ratio of log records included in query results to all available log records is low. Conversely, if this ratio is high (when – for instance – a log record is included in the result of 50% of the queries that scan a particular time partition) this may be acceptable.

4.3.2 *Relational Database*

Log records can be transformed to fit the relational model. Whether this is an idea to pursue depends on how well the characteristics that describe log records map to this model.

To begin, there are different interpretations of the relational model, and different implementations of relational databases, where each has its upsides and downsides. Because of this diversity, it is not feasible to address the subtleties of every different interpretation and implementation. The following discussion assumes the widely known relational model as made popular by the SQL[3] standard.

The basic building blocks of the relation model are sets of n-ary tuples. In more widely known terms these sets are called *tables*, the n-ary tuples are called *rows*, and the rows contain n *columns*. Tables must be defined before they can be used. Their definition includes the columns every row can use, and the data types they can contain. This means that storing free-form data is not inherently possible when applying this model.

Intuitively, every log record in this model should map to a single row in a table. However, since log records are arbitrarily sized sets themselves, it is not possible to use a single table alone when every log record should map to a single row with a predefined and fixed number of columns. Rather, every 2-element tuple that defines a property in a log records needs to be represented in a separate table, where it references the log record it belongs to. For this to work, every log record, and all the properties that it contains need to share a common identifier. Without a common identifier, it is not possible to talk about the same log record – or *entity* – in multiple tables. This is unfortunate, because it increases the amount of data required to store a single log record, while only artificial data is added. Another problem with requiring a separate table that stores every 2-element tuple is that it prevents creating indexes on certain tuples. Either all tuples are indexed by their property name, by their value, or both. This inflexibility can be very costly. Storing and maintaining indexes comes at a cost, so when an index contains more information than is required by the queries that it serves, this is a waste. Not only does it waste storage space, it also wastes processing power, because indexes need to be updated whenever data is added or modified.

This requirement of a split log record and log record properties table can be avoided when the set of properties that log records may contain is restricted to be a fixed set. The set of properties then needs to be defined before any log record is added to the database, and properties that are included in log records but not defined as part of the database schema need to be discarded. Every predefined property will then map to its own column. Because SQL databases allow

3 Structured Query Language

indexing of individual columns, extracting for subsets of log records matching certain criteria is as easy as formulating a SQL query. While this approach allows for querying the data as the relational model prescribes, it ignores the key characteristic that log records can contain an arbitrary number of tuples, and can therefore not be used.

Regardless of the use of a split-table approach or a single-table approach, the number of log records can only increase. Unlike the case with flat files, it is not possible to partition SQL database tables over time, without resorting to techniques that are not inherently part of the SQL standard. An example of such a technique is partitioning over time by using separate tables. If applied, this makes the consumer of the data responsible for choosing the right tables to run a query on, and combine the results. While technically possible, the framework that SQL provides in this case needs to be circumvented to solve this practical problem.

Of the explored approaches to storing log records in a relational database, none appear to be a good fit. The idea to use the relational model to store log records with the described characteristics is therefore be rejected.

4.3.3 *Key/Value Database*

To determine if the key/value model is a good fit for storing log records with the described characteristics, we need to take a look at how storage and retrieval would be implemented. There is a wide range of use cases where the key/value model is applied in practice, ranging from ephemeral in-memory caching to durable storage. Examples of databases that use the key/value model somewhere along this range include: Memcached [23] for in-memory caching, Redis [30] for durable in-memory caching and storage of rich values, and Riak [5] for fault-tolerant, distributed and durable object storage. Depending on the intended use, every product has its advantages and disadvantages. For example: Memcached is not durable, where Redis and Riak are. Memcached and Redis require the data set to fit in main memory, where Riak doesn't have this requirement. Redis can store rich values such as lists and sets, where Memcached and Riak require values to be arbitrary blobs. Riak is distributed out of the box, where Memcached and Redis are not. Regardless of their differences, they all use the key/value model. In this model, there is a 1-to-1 mapping between keys and values. Both keys and values can be arbitrary sized data. To extract values from a key/value database, the consumer needs to know the keys that identify them. Some key/-value databases support range queries over the set of all keys, or support secondary indexes that can be used to annotate and extract subsets of keys, where others require the user to create these indexes themselves.

In the absence of secondary indexes or range queries, it is wise to use keys that can be computed or derived from data related to the value they store. For example, if the value stores log records that originate from the HTTP server previously talked about in this chapter, the key could contain the fact that the log records come from a HTTP server, the time offset for the log records, and the client IP address shared by all log records in that value. Then, if a consumer has the intent to query log records generated by a HTTP server, in a particular time interval, for a specified client IP address, he can generate the keys that are expected to store these records and retrieve them.

However, if the consumer wants to query the same set of log records, *for all* client IP addresses, this presents a problem. It is not feasible for the consumer to compute a key for every possible client IP address and retrieve every key, because the total number of keys will simply be too large. This is better feasible if the database supports range queries over all keys, although this also has restrictions. Range queries typically use lexicographical key ordering. Then, if keys are structured like this: `httpd:<time offset>:<client ip>`, all keys that store log records for a particular time offset regardless of client IP address can be extracted following the lexicographical order between keys. The inclusive start of this range will be `httpd:<time offset>`, and the exclusive stop of this range will be `httpd:<time offset+1>`. Clearly, the lexicographical order between keys only helps when the dimension that needs to be iterated over is the last part of the key. For example, if the `syslogd` file naming scheme as previously discussed is adopted, and the the request path is added as suffix to the current format, it is no longer possible to use lexicographical order to iterate over the dimension storing storing the client IP address, without also iterating over the request paths. This means that either only one dimension can be encoded in the key when lexicographical key ranges are used, or that we need to resort to a different approach for iterating over keys where one or more dimensions are not specified.

Going back to the definitions of the characteristics of the log records, they can be seen as sets of 2-element tuples. Every tuple contains an identifier of a particular dimension, and the value in that dimension. Instead of sets of 2-element tuples, they can therefore also be seen as points in a space with arbitrary dimensionality. In the example talking about log records generated by an HTTP server, the different dimensions include time, the client IP address, the HTTP request method, the HTTP request path, and the HTTP status code. The cardinality of these dimensions widely varies: a dimension storing the HTTP status code has a low cardinality (there are only a few dozen defined HTTP status codes [12]), where a dimension storing an IP address has a significantly higher cardinality (2^{32} for IPv4 [28], 2^{128} for IPv6 [10]). For every dimension that we want to allow querying on, we can maintain a separate index. In this index, every possible value that is an element

of the indexed dimension points to a list of log records that contain that value. In the case of the client IP address, the index contains an entry for every IP address that the system has seen, that maps to the list of log records that contain the tuple for a particular client IP address. Now, if a consumer wants to query log records generated by an HTTP server, in a particular time interval, for a particular client IP address, the system can use the indexes to determine the matching subset per dimension, and use the intersection between these subsets to get the final result.

To be able to refer to individual log records in the key/value model, every log record needs to be stored under its own key. For non-distributed key/value stores, this problem can be solved by using an incrementing integer. For distributed key/value stores, however, an incrementing integer cannot be used because its value can only be guaranteed to be unique locally, not globally. This can in turn be solved by using a consistent distributed store for locking (e.g. ZooKeeper [4], Doozer [11]), or by applying theory for parallel generation of rougly sequential integers [1]. Another approach to creating unique identifiers for log records is to compute a hash for every log record and use that as its key. The only requirement for the hash function is that its output domain needs to be large enough such that the probability of hash collisions is very low. An example of such a hash function is SHA-1 [34], with an output domain of 160 bits. This key can be used to store the log record itself (in serialized form), and as key identifying the log record in the indexes.

This approach appears to be a solution to any possible query that can be formulated for the data. Queries can be transformed into constraints on a per-dimension basis, that in turn are combined by one or more set operations. The per-dimension results not necessarily have to be combined by set intersection; they can be combined by any set operation.

Unfortunately, the approach also has downsides. Listed at key characteristic of log record data, is that the number of log records monotonically grows. For only storing the log records themselves, this is not a problem. While there may be spatial and temporal affinity between log records, they are not inseperable. Values can be distributed over multiple nodes if the capacity of a single node is lacking, using well known distribution techniques such as consistent hashing. Storing the indexes, however, this is a problem. Indexes refer to the log records by their keys, so a growing number of log records means a growing index. The larger the number of log records that are associated with a single entry in an index, the most costly querying it becomes. With an increasing number of keys returned by per-dimension queries, the final set operations combining the per-dimension results into the final result become more costly over time.

To alleviate from this increasing cost over time, the indexes need to be partitioned over time. By partitioning over time, the size upper bound of an index is determined by the upper bound of the number of log records that in a time window as large as the time window the indexes are partitioned by. Only then is it possible to keep the cost of a query for a particular time interval relatively constant, if no other parameters change. Complexity for executing a query slightly increases, because a query now potentially needs to be executed across multiple time partitions, after which the results need to be combined.

If we consider the growing indexes as a solved problem, there is another – more practical – downside to this model. It can be found in the fact that log records may have temporal and spatial affinity. In the application of the key/value model so far, this has been ignored. Using the hash of a log record as its key means all affinity information is lost (for example: log records that were created after each other, or log records that are created by the same producer). This means that log records that are likely to be part of the same query result (when they are temporally close, spatially close, or both) are likely stored on different nodes in the distributed system. If they are stored on a single node, it is highly likely they are stored on different disk blocks. In other words, there is no data locality whatsoever. The lack of data locality is bad for performance when the throughput for looking up log records on disk is bound by disk I/O throughput. Namely, performing a single disk seek and read is cheaper than performing mulitple.

Looking back at the characteristics of the data, the key/value model appears to be well-suited with respect to storing log records without a predefined schema. It is also well-suited with respect to monotonic growth, albeit with the requirement that indexes are partitioned over time. It is less suited with respect to the spatial and temporal affinity of log records. This is not so much an issue at write-time, but can severly hurt performance at read-time.

4.3.4 *Evaluation*

Now we have discussed different models for storage and retrieval of log records, we can evaluate how they perform with respect to the different characteristics of log data.

First, handling monotonic growth can be problematic in both the relational model as in the key/value model. Intrinsic properties of both models need to be circumvented to make sure the query cost remains constant under growth. To partition log records over time in the relational model we need to use a different set of tables for every partition, losing the expressiveness of SQL when a query spans multiple partitions. Partitioning over time is easier in the key/value model. There, the only requirement is that indexes are partitioned

over time. However, it does not provide a vehicle for moving older data to cheaper storage. Using a hash to identify log records removes the information required to determine if log records can be moved to alternative storage, or be removed altogether. It can therefore only be done by manually iterating over log records to determine if they can be moved or removed. Such workarounds are not needed when log records are stored in flat files. Here, partitioning over time is easily achieved by creating a different file for every time partition. As time progresses, new files are created, and older files become immutable. Moving older log records to cheaper storage can be done by moving files. Removing older log records can be done by removing files. When we take a look at the simplicity of every solution to cope with the growing nature of the data set, it is clear that storing log records in flat files is the easiest in terms of operational ease of use.

Second is the fitness of the model to store log records without predefined schema. This only appears to be a problem in the relational model. If log records are stored following this model, every log record needs to be stored in two tables: one to store an identifier for the log record and one to store every 2-element tuple. With every tuple stored as a separate row, it is difficult to maintain indexes over a subset of dimensions; either every tuple is indexed, or no tuple is indexed. Storing log records without a predefined schema is not a problem in the key/value model or when using flat files. In either case the log records need to be serialized and are treated as arbitrary blob of data after that, and indexing needs to be handled separately. Ignoring the fact that log records need to be indexed, we find treating log records as arbitrary blobs the simplest, and preferable, solution.

Lastly, there is the spatial and temporal affinity between log records. In the relational model there is little control over *where* the data is stored. This largely depends on whether or not the database implements key clustering, which indexes are being maintained, and log record insertion order. Affinity between log records – be it spatial or temporal – does not necessarily result in improved locality. In the key/value model there is no control whatsoever. Because log records are identified by hashes as keys, all affinity information is lost when the log records are added to the database. Consequently, the described approach for storing log records in a key/value model does not result in a locality advantage. In the case of flat files, temporal affinity directly results in temporal locality because the files can be written sequentially. This means that log records that are created after each other, by the same producer, are very likely to be stored after each other on disk. Not only does sequentially writing log records to disk result in high throughput at write-time, allowing the log records to be sequentially read also results in high throughput at read-time.

The approach that appears to work best on all fronts is the use of flat files. After files are created, populated and closed, they become

immutable. Monotonic growth of the log data can be handled by moving files with older log records to cheaper storage, or by removing them. Log records can contain arbitrary data because no schema is imposed. The temporal affinity between log records can directly result in a locality advantage, if they are stored sequentially. The only disadvantage with the use of flat files is that there needs to be an additional indexing step to make sure querying doesn't require a full scan of all log records stored in a file. Because this needs to be done out-of-band, there is complete flexibility in choice of indexing algorithm and storage, as well as complete flexibility in choosing which dimensions to index.

4.4 DISTRIBUTION

After establishing an approach for storing and retrieving log data, we need to take a look at how this approach fits in a distributed environment. The logging system needs to cater to many producers of log data, distributed over a vast number of (virtual) machines. Not only needs the system to be ready for a significant number of producers, it needs to be ready for significant growth. Linear scalability in this type of environment is not just a desirable property, it is a requirment. This section describes various techniques that can be used to achieve scalability, and how they can be applied for the problem at hand.

4.4.1 *Topology*

To understand the scope of the problem of distribution, we must first establish *how* the problem is distributed. We do this by discussing possible network topologies for the logging system, with an increasing degree of distribution.

The simplest network topology for the logging system is one without a network. In this contrived topology the various processes and tasks that collectively make up this logging system are all co-located on a single machine. These processes are:

THE PRODUCING PROCESS

The source of log records.

Every process in the system that is connected to the logging system may be capturing events, transforming them into log records and sending them to the storing process. An increase in the number of producing processes, or the number of log records they emit likely requires an increase in the number of storage processes as well, to handle the increasing production load. The more log records are sent to the storage processes, the more resources are needed to receive and store them. In this

basic topology, the producing process, or multiple producing processes, are co-located with the other processes.

THE STORAGE PROCESS
The sink for log records.

The storage of log records is a composite task. First, there is the intake of log records sent by producing processes. After receiving them, they may be initially kept in an in-memory buffer before finally being written to disk. The storage process is also responsible for indexing the log records it stores, and serving querying processes. Querying processes can connect to the storage process and make queries on the set of log records that it stores. The required scale of the storage process is influenced by both the load generated by producing processes, as well as the load generated by the querying processes. In this basic topology, the only storage process is co-located with the other processes.

THE QUERYING PROCESS
A filter for log records.

Querying can be done by any process connected to the logging system. After a query is formulated, it is sent to one or more storage processes. These storage processes are responsible for extracting matching log records from the set that they manage and send them back to the querying process. An increase in the number of querying process, or the complexity of the queries they want to execute likely requires an increase in the number of storage processes as well, to handle the increasing query load. In this basic topology, the querying process, or multiple querying processes, are co-located with the other processes.

This contrived topology is shown in Figure 5. The topology figures use the letters P, S, and Q to denote the different processes. White boxes are nodes in the system. Lines between nodes are communication paths.

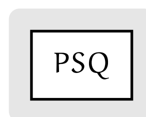


Figure 5: Topology where all processes run on a single node

In the stand-alone topology, there is no flow of information outside the single node that hosts every process. In reality however, it is likely that other nodes besides the one producing and storing the log records is interested in that data. As an example, imagine administrators who want to know how their system is doing. Likely the nodes they use to retrieve the query result are different from the nodes that

produce and store the log records. It is possible that a node that produces and stores log records has zero or more incoming connections from nodes that want to execute queries against its data set. While this does not mean that the node can no longer query itself, Q is omitted from the node that executes PS to prevent ambiguity. The corresponding topology is shown in Figure 6.

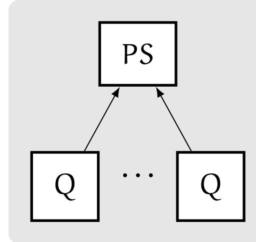


Figure 6: Topology where querying processes are distributed

Only systems that are run on a single node also produce log records on a single node. When a system grows beyond the capacity of a single node and requires more than one node to execute, there will be more than one node producing log records. If we take the topology from Figure 6 as a basis, the difference is that we get multiple PS nodes, shown in Figure 7. Instead of one node producing and storing log records, there are now multiple. The nodes that coordinate query execution now have to connect to every PS node to get a complete view of the system. If they do not do this, they may miss log records that should have been part of the query result. Because the tuples that form a log records are completely arbitrary, any PS node may be responsible for creating a log record that is of interest to a particular query.

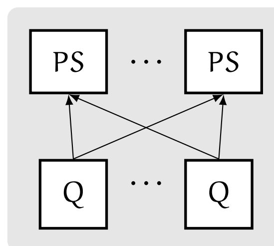


Figure 7: Topology where log record production is distributed

This can be illustrated by the following example. Imagine a cluster of HTTP servers that serves traffic for a number of different web sites, where load-balancing is achieved through round-robin DNS. The responsibility of picking a server to talk to is moved to the client when this technique is applied. This means that the cluster itself doesn't have control over where traffic is routed; every server is equally likely to receive a request for any of the web sites. Now, assume every HTTP server creates a log record for every request it receives, and that this

log record includes the requests HTTP headers. Included in a requests HTTP headers is the hostname of the web site that the request is trying to talk to. Now, if a querying process wants to know how many requests a particular web site has received, it needs to query every single HTTP server, because every HTTP server is equally likely to hold log records that are relevant for the outcome of the query.

There are two problems with this topology. First, the performance characteristics of the core tasks that these nodes execute may be impacted when the logging system starts claiming (potentially sparse) resources. How significant these resource claims are depends on how many queries are being executed, and the complexity of these queries. The higher this resource claim by the logging system, the fewer resources that can be used to execute the tasks these nodes are meant to execute. Second, the number of connections every querying process needs to make is equal to the number of nodes that produce log records. This imposes an inherent maximum to the number of queries that can be concurrently executed. If all queries that can possibly be executed follow a request-response flow, this is not a significant problem because queries can be executed serially. However, this system is also required to execute streaming queries, where log records are forwarded to the querying process immediately after being received by the storage process. The number of concurrently running queries can therefore be large, and not serializable.

These concerns form the motivation for the topology shown in Figure 8. Here, the production and storage of log records is decoupled. The nodes that are responsable for the production of log records hand them off to nodes that are responsible for storing them. Querying nodes only need to communicate with the storage nodes, and therefore no longer have a direct resource impact on the producing nodes. Also, because the nodes hosting the storage processes have a focused and dedicated responsibility there can be fewer nodes in the storage tier than in the production tier. With $\#S < \#P$ the querying processes are also alleviated in terms of the number of connections they have to make. Instead of having to connect to #P different nodes, they now have to connect to #S different nodes to execute a query.

This topology also raises a number of questions. Are log records pushed to the storage tier by the producing tier, or pulled from the producing tier to the storage tier? If they are pushed, how do producing nodes know to which storing node they should push their log records? If they are pulled, how do storing nodes know which producing nodes to pull log records from? What happens when a storing node becomes unavailable? Are log records dropped on the floor? Are log records stored in some transient store, that can be sourced when the storing node becomes available again?

Some of these problems can be addressed by treating the communication between the producing tier and the storing tier as querying,

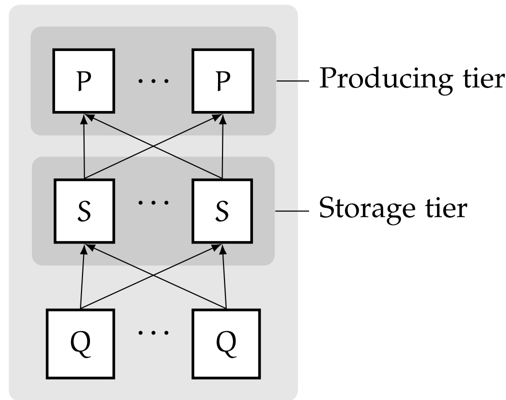


Figure 8: Topology with a separate tier for storing processes

as in the more simple topology shown in Figure 7. This is achieved by adding S to the producing tier, and adding Q to the storage tier. As a result, the storage tier is required to actively query the producing tier for its log records. Note that this does not mean that the storage tier needs to poll the producing tier for new records; it can use a streaming query and the producing tier will stream new log records directly to the storing tier.

The fact that every node in the producing tier now has its own local storage process has positive side-effects. For example, it protects log records from being lost in the event of storage node unavailability. When this happens, a different storage node can pick up streaming log records where the unavailable storage node left off, without loss of information. Similarly, multiple nodes in the storage tier can stream log records from a single node in the producing tier (e.g. for stand-by redundancy). Another positive side-effect is that it re-enables administrators to execute a query directly against a node in the producing tier. In the case of an outage in the storage tier, this may be crucial in still being able to extract information from the system.

For a node in the storage tier to be able to pick up where a node that became unavailable left off, it needs to know *where* it left off. It it not possible to simply run a streaming query for all new log records, as that may result in the loss of log records that were sent to the unavailable node and were not acknowledged. This leads to questioning acknowledgement.

So far, we have only talked about storage as writing to flat files and indexes, partitioned by time. For every node in the storage tier, this results in an immutable set of files per time interval. What happens to these immutable sets of files once they become immutable is not yet defined. If nothing happens, and the storage node is itself responsible for maintaining the sets of files it created, a failure of this node means that a subset of all log records is immeditaly unavailable. This is not acceptable if the failure results in permanent data loss. The options

for protecting against data loss in the case of a node failure are to either build replication into the storage tier, or to decouple it from the storage tier and use an off-the-shelf solution for reliable storage. Key in choosing between one or the other is that the sets of files become immutable over time. This property means that there is a small, fixed set of files that is being written to by the storage tier, and there is a large, growing set of files that is read-only. We can therefore choose to use an off-the-shelf solution for reliable storage, while not impacting write performance of the storage tier, as long as the sets of files are copied to the reliable storage once they become immutable. It does not matter which specific solution is chosen, as long as it provides full, shared access for all nodes in the storage tier. Because of the shared access, nodes can now determine the most recent set of files stored by sibling nodes, and thereby determine to what point in time sibling nodes have *acknowledged* the log records they have received. The modified topology including the reliable storage is shown in Figure 9.

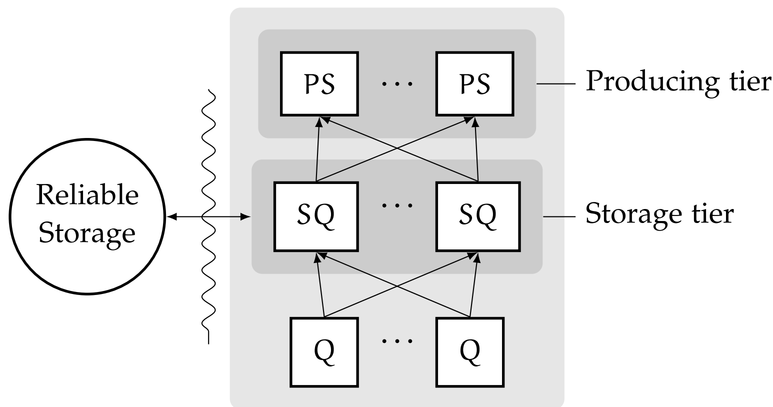


Figure 9: Topology with external reliable storage

The reliable storage can be implemented using a wide variety of products and techniques. For example, it can be implemented using the Hadoop Distributed File System [16]. This not only serves as a reliable storage backend, but it also allows the use of Hadoop MapReduce [17] on the immutable sets of files it stores, for offline processing. Reliable storage can also be implemented using NFS[4]. The file system that is served over NFS can in turn be hosted on a cheap hard disk, or on an expensive SAN[5]. The choice for one or the other depends on the reliability requirements of the logging system, but does not matter for the logging system as long as it fulfills the shared access requirements.

4 Network File System
5 Storage Area Network

A node in the storage tier can now use the reliable store as the primary source in determining where it should pick up work where a node that became unavailable left off. How much work is duplicated in doing so depends on the length of the time intervals that are used to partition the sets of files. Choosing the length of the interval that is partitioned by significantly impacts other aspects of the system. For example, for any node in the storage tier to recreate a set of files otherwise created by a sibling node, nodes in the producing tier should keep *at least* one interval worth of log records around. To keep disk usage in the producing tier under control, the interval can be chosen to be short. In turn, a short interval means more partitions, thus more sets of files, thus more pressure on the storage tier in executing queries. These effects should influence choosing the length of the interval to partition by.

4.4.2 *Partitioning*

In the topologies shown so far, the storage tier receives streams of log records from the producing tier by querying it. This bypasses the fact that the nodes in the storage tier need to be directed *which* nodes in the production tier to query. Because $\#S < \#P$ every node in the storage tier queries at least 1 node in the producing tier. In reality the number of streams received per node in the storage tier will be higher, because the nodes in the storage tier are meant to be dedicated to this task and therefore should be able to handle multiple streams. The problem that needs to be solved is to determine which nodes in the storage tier are responsible for querying which nodes in the producing tier.

Every node in P should be queried by at least one node in S. It is possible that a node in P is queried by more than 1 node in S for availability reasons, but for a large S it should not be queried by *all* nodes in S for scalability reasons. Namely, if every node in S would query every node in P, the system cannot scale P beyond the capacity of a single node in S.

The mapping between nodes in S and nodes in P can be made using hashing. Here, every node in P has a unique identifier. This identifier can be a random string, or something inherently unique to the node such as its IP address, as long as it is unique. To determine which node $s \in S$ is responsible for querying a node $p \in P$, we compute $i = (hash(p) \bmod \#S)$ and use i to index an ordered version S' of S. This gives us a way to map every $p \in P$ to a single $s \in S$, but it breaks down when $\#S$ changes, and therefore the range of i changes. When this happens, almost every mapping of $p \in P$ to $s \in S$ changes, and almost every node in the storage tier will need to start querying a different node in the producing tier. This is not catastrophic for our use case, though. Since every node in the storage tier needs to be

queried to get a complete query result, it should not matter *where* log records are stored, as long as they are stored *somewhere*. It does mean, however, that log records for two different producing nodes that were once colocated, may no longer be after #S changes. Given an identifier of a node p, it is not possible to determine which set of files its log records is stored in without knowing #S *at the time the set of files was created*.

This problem can be solved using consistent hashing [18, 9]. Instead of bounding the range of the hashing function by the variable #S, it uses a very large constant to bound its range. The nodes $s \in S$ to map nodes $p \in P$ to then randomly pick one or more values $v_1, \ldots, v_n$ that lie in the same range. Then, the value i for some node p maps to the node with the lowest v with $v > i$, or $v > 0$ if none matches. If the range of the hash function is visualized as a ring, the node is found by starting at position i and walking clockwise until a node is found for which $v = i$. Because the range of the hash function is constant, every p consistently maps to the same i. When nodes are added to S, they will take over part of the range of their neighbors, and when nodes are removed from S, neighbors will take over their range. Instead of changing almost every mapping from nodes in the producing tier to nodes in the storage tier, only few mappings change. Because the values v are chosen randomly, given an identifier of a node p, it is only possible to determine which set of files its log records is stored in when a different set of files is used for every p. This is infeasible when P is large; ideally we want to coalesce log records from multiple producers in a single set of files per node in the storage tier. Another downside to randomly choosing values v is that no guarantees about load distribution can be made. A slight variation of the algorithm partitions the range of the hash function in a predefined number of partitions, or virtual nodes [9]. Every virtual node is responsible for a predefined and equal range of the hash range. The different nodes in the storage tier then each become responsible for multiple of these virtual nodes, in an approximate uniform distribution over the range. Now, when nodes are added to S they will take over virtual nodes from other nodes, and when they are removed other nodes will take over their virtual nodes. Given an identifier of a node p, it can now be consistently mapped to a single virtual node. The index of a virtual node can be used to identify the set of files that contains all the log records for that virtual node, solving the problem of determining in which set of files the log records produced by a specific node should be stored.

5 IMPLEMENTATION

In this chapter we discuss the trade-offs and decisions made in implementing a prototype of the system discussed in Chapter 4. Recall that the log record data that the system is concerned with is immutable; it doesn't change once it has been created. This leads us to think that is it possible to sequentially write every log record that is received to disk, and be done. There are a number of subtleties that need to be reviewed before we can judge if this is indeed the case.

5.1 TIME

Observations of discrete time by different entities are likely to be different. With respect to log records, there are multiple points in time, observed by multiple entities. First, there is the point in time when the log record is created. Next, there are points in time when the log record is received by a storage node, when it is queried by a querying node, when it is stored again, queried again, etcetera. Every time an interaction with a log record takes place, its "time" needs to be reinterpreted. In Chapter 4, we claimed that time should be treated specially. It is the dimension that is required to be specified in every query, that is used to group data as time progresses, and that can be used to determine causality. Yet, interpreting it suffers from ambiguity. With multiple entities operating on log records, there are multiple clocks involved. With multiple clocks, there are different versions of time.

For every individual node, we assume its clock is monotonically increasing. That means that the clock can only move forward in time, not backward. This is an issue when the clock needs to move backwards in time (e.g. a leap second is inserted, and *happens twice*). This can be solved by stopping the clock or slowing it down and distributing a single leap second over multiple *slower* seconds [24]. Forward jumps in time do not violate the constraint that time only moves forward, so are not a problem.

This assumption alone is not enough, because it doesn't put constraints on the synchronization of different clocks in the system. Therefore, we also assume that every individual node goes through a phase of clock synchronization when it starts. This can be executed using NTP[1]. This protocol, and the associated tooling, uses a single authoritative server as its synchronization target, whose clock is assumed to be correct. Not only should synchronization be executed when a

1 Network Time Protocol

node is started, it should also run periodically to compensate for clock drift.

With these assumptions in place, we can consequently dare to assume that the clocks participating in the system are approximately synchronized. This is a dangerous assumption to make, because it is impossible to fully prevent out of sync clocks. We therefore need to put additional protective measures in place to prevent clocks that have drifted too far from corrupting the system. This can be done by by formulating an upper bound for the tolerable difference in time. When the time encoded in a log record fails to meet this bound, it can for example be discarded. The fact that a log record is discarded in itself deserves a log record to be created to capture that event, because it is not something that should remain unnoticed as it can possibly indicate a clock de-synchronization in the system.

It is required to include a time interval in queries for the system. When events take place that trigger the creation of a log record, time may be the property that can correlate them. For instance, if an event takes place on a node that results in sharing the local time on that node with an external entity, *that* time will be used by that external entity to try and find related events. Now, if instead of using the local time for log records, the time on the node that stores the log records is used, these two may be out of sync causing the external entity not being able to find related events. Because this behavior is not intuitive, we choose to always capture time the moment a log record is created. This means that if local time on a node is externalized, it will always be possible for the entities that captured that time to find events that took place on that node around that same time. For example, a HTTP server encodes time in its responses in the `Date` header field. The client making the request can observe that time, and want to query the logging system for events that happened during the execution of its request. A difference between the local time that the log records were created and the remote time that they were stored and they are associated with makes it impossible to get the intended results. Then, the difference between the time that is encoded in log records and the local time on the nodes that eventually store those log records is effectively the latency between creating and storing them. The distribution of this latency can then in itself be captured, resulting in a measure of timeliness (again, assuming the clocks of the producing and storing nodes are approximately synchronized).

Storing processes may receive multiple streams of log records originating from multiple different remote sources. This means that at time of arrival, the log records from these different remote sources may encode slightly differing time. If the different streams are multiplexed to a single stream, the time encoded by the log records in that stream may no longer be monotonically increasing. Monotonically increasing time can be used as the primary way to determine causality

(albeit primitively). When the multiplexed stream is directly streamed to a file, and the monotonicity property no longer applies, it is only possible to determine if other log records happened before a specific log record by scanning the entire file, because they may appear anywhere in the stream. All but one stream in the multiplexed stream can be paused for some amount of time. This causes log records from the stream that is not paused to appear at a position in the file before log records from the temporarily paused streams, while the time encoded by the latter log records happened before the time encoded in the former log records. Conversely, one stream can be paused for some amount of time, which causes its log records to appear at a position in the file after log records from the other streams.

This is not necessarily a bad property, because the monotonicity property still holds for log records that are created by the same producer, even in a file with interleaved streams. Interleaving streams on a per-record basis does, however, negatively impact locality.

5.2 LOCALITY

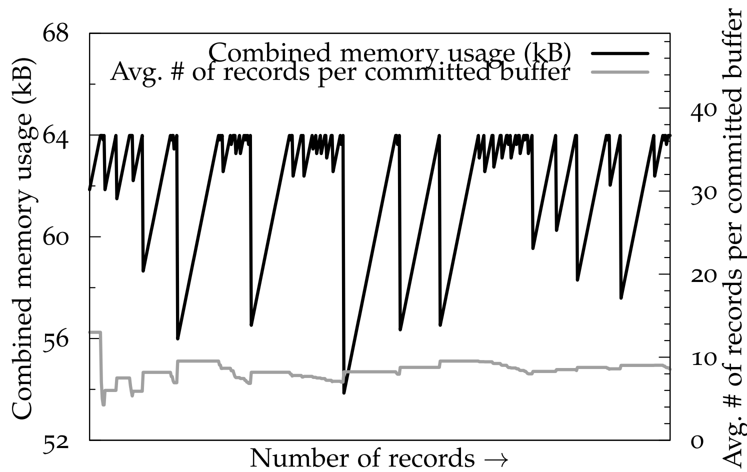
When streams from multiple sources are multiplexed into a single stream, extracting log records from one of those sources can be costly. If every stream equally contributes to the multiplexed stream, every log record in the file storing the stream needs to be examined and filtered to reconstruct a single source stream. This is not a problem in itself, but we can think of common query workloads where this is suboptimal. Namely, queries will likely be interested in the log records that are created by one particular producer. An example of such a producer/query pattern is where a single producer emits log records capturing the resource usage on the node where it is running. A consumer of this data is more likely to be interested in a series of these samples rather than a single sample in particular. Having a some degree of data locality in the files produced by multiplexing multiple streams helps in keeping the cost of answering such queries relatively low.

To achieve data locality on a per-stream basis in the file produced by multiplexing multiple streams, the storing process needs to keep a per-stream buffer that is periodically flushed to that file. The size of this buffer depends on a number of aspects. As every stream uses its own buffer, the more streams are multiplexed, the more buffers need to be allocated. If the ideal time interval to coalesce log records on disk is a minute, the buffer size also depends on the rate at which new log records are received. If the ideal number of log records to coalesce on disk is 50, the buffer size also depends on the byte size of the serialized representation of the log records. Clearly, the optimal size of this buffer depends on many aspects of the workload, and needs to be set accordingly.

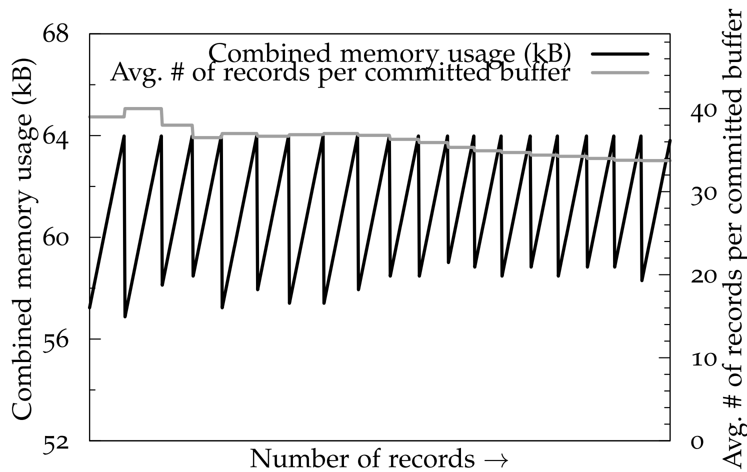
Instead of choosing a fixed buffer size, we choose to make it dynamic. Picking a fixed buffer size can result in unpredictable memory usage for the storage process. For example, the number of streams may suddenly spike, the rate at which they are received may spike, the size of individual log records may spike, etcetera). We set the target memory usage to be constant, thus predictable, and make buffering dynamic. Now, it is still possible to flush a buffer based on its serialized byte size, the number of log records it contains, or the time interval the log records it contains spans, but it may also need to be flushed when the total target memory usage is exceeded. This happens when none of the per-buffer limits is reached, but the memory usage of all buffers combined is higher than the target memory usage. When this happens, one or more buffers need to be flushed to disk to disk such that the combined memory usage is reduced until it is lower than the target memory usage. Choosing which buffers are flushed can be done in a number of ways. One way to choose buffers is by picking them at random. A buffer is chosen from the set of available buffers, it is written to disk and its memory is deallocated. This process is repeated until the combined memory usage no longer exceeds the target memory usage. When there are many small buffers and only a few large buffer this strategy is suboptimal, because flushing fewer buffers keeps the total cost of flushing low, and results in better locality for the other buffers. Instead of choosing buffers randomly, another way to choose buffers from the set of available buffers is to use order between the buffers. The buffers are ordered by their serialized byte size and are stored in a priority queue. Then, when the combined memory usage exceeds the target memory usage, the largest buffer is removed from the priority queue, is written to disk, and its memory is deallocated. This process is also repeated until the combined memory usage no longer exceeds the target memory usage. With this strategy, fewer buffers need to be flushed every time the combined memory usage exceeds the target memory usage. It gives us the ability to both keep the memory usage of the storage process under control and store log records with good locality.

The effects of both of these approaches are presented in Figure 10. The data in these figures is generated by executing an experiment where log records of constant size are generated and are tied to a series of streams. Log records are distributed over these streams following an exponential distribution with $\lambda = 1$. This is done to simulate the varying rate at which log records are added to different streams. The artificial memory limit is set to 64 kilobytes. Whenever this limit is reached or exceeded, one buffer is chosen following either approach, and is committed to disk. Because the serialized byte size of the log records is constant, and the rate at which they are added is constant, this results in a sawtooth pattern. The regularity of the sawtooth depends on which approach is used to choose a buffer to

flush. To evaluate the locality fitness of both approached, both graphs include the average number of log records contained in every individual commit operation. The more log records are contained in an individual commit, the better the locality for its stream, because more log records can be read from disk in a single sequential read operation. Clearly, the locality is better when the largest buffer is committed to disk every time the combined memory usage exceeds the target memory usage.



(a) Committing a random buffer



(b) Committing the largest buffer

Figure 10: Memory usage and a locality measure for storing log records with constant size from multiple streams in a single file. Streams are buffered, and buffers are flushed when the cumulative memory usage exceeds 64 kB.

5.3 FORMAT

With a strategy in place to preserve some data locality while multiplexing multiple streams to a single file, we now discuss the format

of this file is. Recall from Chapter 4 that every log record is an immutable set of 2-element tuples. In every tuple, both the dimension identifier and the value in that dimension can contain arbitrary binary data. This requires the backing storage format to be binary safe. Also, log records need to be individually addressable for indexing purposes. There are a number of existing serialization technologies that can be used for storing this type of data. Examples are Thrift [3], Protocol Buffers [29], and MessagePack [13]. Where both Thrift and Protocol Buffers require a schema and a compiler to generate code for encoding and decoding, MessagePack does not require a schema. Because it does not require a schema, MessagePack is said to be *self-describing*, that is: given a encoded record, a decoder can figure out what the structure of the record is and what the data types of its fields are. This property is useful when the producer of the data doesn't know what the consumer is. The consumer can decode the data it received entirely by itself. In this case, the producer and the consumer are the same process, which means that a self-describing serialization technology is not an absolute necessity. However, when code upgrades are performed, the producer and consumer may no longer be the same process, and a self-describing serialization technique can be beneficial for extensibility. When requirements change over time and additional data needs to be added to the log records, this can be done without breaking backwards compatibility (e.g. older code can read data generated by newer code). The same can be possible for serialization techniques that require a schema to be defined, but is less light-weight. We therefore choose to use a self-describing serialization technology to serialize *individual log records*.

This takes care of serializing individual log records, but does not give a solution for storing a large number of log records in a single file. Because every log record is an individual unit, we may choose to not add additional data to the file have simply be a continuous stream of these log records. There are two problems with this approach.

First, log records are written in groups at a time. Because every group of log records originates from the same source, it is useful to know where the group starts and stops. This allows the representation of data in memory and in the file to be the same, reducing complexity. Without having this information, the data in log records needs to be inspected to find out to what group it belongs (e.g. by comparing the values in their `source` dimension). We will see in the next section that having log records be addressable by groups is also advantageous when indexing the data.

Second, every log record needs to be fully interpreted before being able to move to the next log record. This is not a problem of semantics, but a problem of optimization. When a consumer of the data is iterating over log records to seek to a certain point in time, it is more optimal if only the time property needs to be extracted than if the

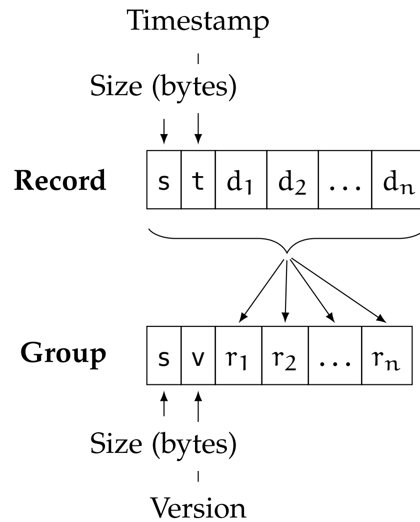


Figure 11: Serialized format of groups of log records and log records themselves.

entire log record needs to be decoded, to be able to figure out if the log record should be used or not. To avoid having to decode an entire log record, and being able to skip over it in reading log records from a stream, its byte size needs to be known. When the byte size of a log record is known, iterating over encoded log records becomes an operation that is not constrained by the number of dimensions encoded in individual log records.

These two problems need to be addressed in the format of the storage files. The first problem is addressed by prefixing every group of log records with the total byte size of the group. Similarly to knowing the byte size of individual log records, this allows a consumer of the data to quickly iterate over groups of log records, without having to iterate over the individual log records. For robustness, we also prefix every group with a version byte for robustness. With the version byte in the header, future modifications can be made while maintaining backwards compatibility. The second problem is addressed by prefixing every log record with its byte size. The format with these modifications is shown in Figure 11. The tuples in individual log records are denoted as d_1 through d_n. The log records in individual groups are denoted as r_1 through r_n. A file storing log records consists of consecutive blocks of groups. With the prefixed size in bytes of both groups of log records and log records themselves, a consumer of the data can easily seek over a large file without having to fully interpret all data.

5.4 INDEXING

To be able to quickly retrieve a subset of log records, the files storing log records need to be indexed. Not doing so requires a scan of all data, and testing every log record for a match, to compute query results.

Indexing is done per dimension. Not necessarily all dimensions need to be indexed. For example, consider log records produced by a HTTP server that include the `User-Agent` header as a separate dimension. Only if queries are executed that filter specifically on this dimension, it should be indexed. If such queries are never executed, that dimension does not necessarily need to be indexed, and doing so regardless incurs unnecessary cost. Indexing is therefore be controlled on a per-dimension basis.

While it is possible to create indexes that contain pointers to individual log records, doing so is not absolutely necessary. Because log records are grouped in blocks that are produced by a single producer, it is likely that there exists a relation between them. In the extreme case, when a producer always includes the same value for a particular dimension, it is enough to only index the block instead of indexing every log record individually. On the other hand, if a value only occurs for a single log record in the entire block, the entire block is indexed and all its sibling log records need to be tested for a match when executing a query. The overhead that is incurred when only a single log record out of an entire block matches is limited by the maximum number of log records that is stored in every group. This number can depend on the dynamic buffering parameters as discussed in the section about locality, or it can also be artificially limited. The extra overhead is also limited by the fact that the log records need to be tested for a match have good locality with at least a single matching log record. If a group of log records can be read from file in a single disk read, testing log records for matches is only bounded by CPU and not I/O.

Because of this, we choose to point to groups instead of individual log records. If the average number of log records per group is 50, this choice *increases* the *worst case* number of log records that need to be tested and *are not included in the result* by a factor of 50, *but reduces* the size of every index entry by a factor of 50, because it only contains a pointer to a group instead of 50 pointers to log records.

The pointers to groups are only known when they are flushed to disk. When this happens we capture the offset in the file where it is written, and use that as pointer to the group. Because a group of log records in entirely self-contained, this is all the information that is needed to point to a group of log records. A problem that this presents it that it is not possible to index log records as they are received. The offset in the file is only known when the group is

flushed to disk. We therefore keep two separate indexes, an index on disk I_D and an index in memory I_M. The in-memory index consists of a hash table per indexed dimension, with values pointing to a set of buffer objects. Every time a log record is added to a buffer, it is immediately indexed in I_M. Then, when a buffer is flushed to disk, its contribution to I_M is also "finalized" and is merged into I_D.

As time progresses and more records are added, eventually the set of files that is used to store log records for a particular time slot will roll over and can be finalized. When this happens, pending buffers are flushed to disk, and their respective contributions to I_M are merged into I_D. After this, the file containing all log records for that time slot will be accompanied by a set of indexes I_D that contain pointers to all stored groups of log records.

Note that while indexes are instrumental for speeding up queries, they are not required. Without indexes, it is still possible to execute a query, though performance may be far worse. It is possible to add indexes after the file for a particular time slot has been finalized, and can be created outside of the storage process, or even by third party tooling.

5.5 QUERYING

Executing queries against the storage format and its indexes follows naturally. A query consists of a set of constraints for zero or more dimensions, and always includes a time interval to execute against. Using the interval the storage nodes determine which files they need to answer the query. If some of these files have already been moved to long-term storage, they can be retrieved when needed. Because every file and its indexes contains an absolute and non-overlapping time interval, executing a query spanning a longer interval means sequentially executing it against every individual file. Executing a query against a single file is harder. While there is temporal ordering inside every group, there is no temporal ordering between groups in the file. Temporal ordering is therefore done at query time.

Every query potentially scans every group in the file. The constraints of a query are used to eliminate groups from the sets of groups to scan. Because groups of log records are identified identically across dimensions, this set operations can be used to reduce the size of the set of groups to scan. For example, a query can limit d_1 equal v_1, and d_2 to v_2. Then, the set of groups to scan is equal to $I_{d_1}[v_1] \cap I_{d_2}[v_2]$, where I_{d_n} is the index for dimension d_n. If constraints are put on dimensions that are not indexed, any log record might match and the constraint is translated to the set of all groups.

The set of groups to scan is then translated into a set of cursors. Because we have the guarantee that log records from a single source have monotonically increasing time, we only need a single cursor

per log record source. This is achieved by enumerating the groups in the set of groups to scan. Every individual source that is seen when enumerating is assigned its own cursor. Every cursor is associated with a chronologically ordered list of groups for its source. Every cursor maintains a pointer to its first log record. The cursors are then inserted in a priority queue, and use the time encoded in their first log record for ordering. The result of the query is then computed by iteratively removing the cursor with the lowest timestamp from the queue, returning it to the client, advancing its pointer, and re-inserting it into the priority queue if it has a next log record. If the next log record has a timestamp lower than the log record pointed to by the cursor that is next in the priority queue, we can take a shortcut and skip re-inserting and removing from the priority queue. These steps are executed for every file that overlaps with the time interval that the query limited to.

When the system consist of multiple storage nodes, queries effectively become a map/reduce job over these nodes. The query is executed on every individual storage node, and the client executing the query is responsible for merging the results together.

6 CONCLUSION

While the trend that applications no longer depend on underlying infrastructure brings advantages and flexibility in terms of cost and scalability, it also presents a number of interesting problems. With highly ephemeral application instances, how can the distributed state of that application be monitored? Are existing monitoring systems adequate to do so, or does the approach towards monitoring applications in such highly ephemeral environments need to be rethought? These are questions we had going starting with the work presented in this thesis. Concepts used in existing monitoring solutions appeared to be unfit to represent the highly dynamic nature of this type of application. Where conventional monitoring solutions often focus on either monitoring log data or monitoring metrics data, we argue that it is possible to derive metrics from log data when not embedded in log data, and there proposed a single model that would be able to satisfy both types of monitoring.

From the analysis, we concluded that modeling log records produced by these applications as sets of tuples gives the best expressiveness and flexibility. Based on the nature of the data, we formulated an approach towards storing this type of log data. With the data model and storage approach in mind, we evaluated different configurations of processes producing log records, storing log records and querying for log records. This lead to an approach for distribution of the problem that allows the intake capacity of the storage processes to be dynamically scaled up.

By storing log records in fixed interval files, and deferring fault-tolerant storage to existing solutions, we explicitly allow the files to be usable by third party software. These files with log records are not tied to a particular database, and can be handled as any other file. They can therefore be processed with frameworks like Hadoop Map/Reduce, they can be copied to increase capacity for answering queries, and they can be removed when they are no longer of use.

In future work, the storage format can be made more efficient. There are various techniques that can be applied to remove duplication in the files storing log records, and to optimize the execution of set operations against the indexes (e.g. using bitmap indices). Also, indexes that support operations other than string equality should be explored, to increase richness of queries.

BIBLIOGRAPHY

[1] Tom Altman and Yoshihide Igarashi. Roughly Sorting: Sequential and Parallel Approach. *Journal of information processing*, 12(2):154–158, 1989-08-30.

[2] Amazon S3 Availability Event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html.

[3] Apache™ Thrift™. http://thrift.apache.org/.

[4] Apache™ zookeeper™. http://zookeeper.apache.org/.

[5] Basho | makers of the Riak distributed database. http://basho.com/.

[6] Cloud Foundry. http://cloudfoundry.org/.

[7] Derek Collison. NATS. https://github.com/derekcollison/nats.

[8] Common Log Format. http://en.wikipedia.org/wiki/Common_Log_Format.

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[10] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), December 1998. Updated by RFCs 5095, 5722, 5871, 6437, 6564.

[11] Doozer. https://github.com/ha/doozer.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266, 6585.

[13] Sadayuki Furuhashi. MessagePack: It's like JSON, but fast and small. http://msgpack.org/.

[14] Ganglia Monitoring System. http://ganglia.sourceforge.net/.

[15] Graphite - Scalable Realtime Graphing. http://graphite.wikidot.com/.

[16] Hadoop™ Distributed File System. http://hadoop.apache.org/hdfs/.

[17] Hadoop™ MapReduce. http://hadoop.apache.org/mapreduce/.

[18] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[19] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[20] Log Files - Apache HTTP Server. http://httpd.apache.org/docs/2.2/logs.html#accesslog.

[21] logstash - open source log management. http://logstash.net/.

[22] C. Lonvick. The BSD Syslog Protocol. RFC 3164 (Informational), August 2001. Obsoleted by RFC 5424.

[23] memcached - a distributed memory object caching system. http://memcached.org/.

[24] David L. Mills. The NTP Timescale and Leap Seconds. http://www.eecis.udel.edu/~mills/leap.html.

[25] C. Mohan, R. Strong, and S. Finkelstein. Method for distributed transaction commit and recovery using Byzantine Agreement within clusters of processors. *SIGOPS Oper. Syst. Rev.*, 19(3):29–43, July 1985.

[26] Nagios - The Industry Standard in IT Infrastructure Monitoring. http://www.nagios.org/.

[27] OODA loop. http://en.wikipedia.org/wiki/OODA_loop.

[28] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFCs 1349, 2474.

[29] Protocol Buffers. http://code.google.com/p/protobuf/.

[30] Redis. http://redis.io/.

[31] RRDtool. http://oss.oetiker.ch/rrdtool/.

[32] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: a large-scale field study. In *Proceedings*

of the eleventh international joint conference on Measurement and modeling of computer systems, SIGMETRICS '09, pages 193–204, New York, NY, USA, 2009. ACM.

[33] Baron Schwartz, Peter Zaitsev, Vadim Tkachenko, Jeremy D. Zawodny, Arjen Lentz, and Derek J. Balling. *High Performance MySQL: Optimization, Backups, Replication, and Load-Balancing*. O'Reilly Media, 2.a. edition, 2008.

[34] SHA-1. http://en.wikipedia.org/wiki/SHA-1.

[35] Splunk - Operational Intelligence, Log Management, Application Management, Enterprise Security and Compliance. http://www.splunk.com/.

[36] Syslog – The GNU C Library. http://www.gnu.org/software/libc/manual/html_node/Syslog.html.