O'REILLY®

Database Reliability Engineering

DESIGNING AND OPERATING RESILIENT DATABASE SYSTEMS

Laine Campbell & Charity Majors

Database Reliability Engineering

Building and Operating Resilient Datastores

Laine Campbell and Charity Majors

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Database Reliability Engineering
by Laine Campbell and Charity Majors

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editors: Courtney Allen and Virginia Wilson
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR
Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition
2016-11-17: First Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491925904* for release details.

978-1-491-92590-4

[FILL IN]

Table of Contents

CHAPTER 1

Introducing Database Reliability Engineering

Our goal with this book is to provide the guidance and framework for you, the reader, to grow on the path to being a truly excellent database reliability engineer. When naming the book we chose to use the words *reliability engineer*, rather than administrator.

Ben Treynor, VP of Engineering at Google says the following about reliability engineering:

> fundamentally doing work that has historically been done by an operations team, but using engineers with software expertise, and banking on the fact that these engineers are inherently both predisposed to, and have the ability to, substitute automation for human labor.

Today's database professionals must be engineers, not administrators. We build things. We create things. As engineers practicing devops, we are all in this together, and nothing is someone else's problem. As engineers, we apply repeatable processes, established knowledge and expert judgment to design, build and operate production data stores and the data structures within. As database reliability engineers, we must take the operational principles and the depth of database expertise that we possess one step further.

If you look at the non-storage components of today's infrastructures, you will see systems that are easily built, run and destroyed via programmatic and often automatic means. The lifetimes of these components can be measured in days, and sometimes even hours or

minutes. When one goes away, there is any number of others to step in and keep the quality of service at expected goals.

Guiding Principles of the DBRE

As we sat down to write this book, one of the first questions we asked ourselves was what the principles underlying this new iteration of the database profession were. If we were redefining the way people approached data store design and management, we needed to define the foundations for the behaviors we were espousing.

Protect the Data

Traditionally, this always has been a foundational principle of the database professional, and still is. The generally accepted approach has been attempted via:

- A strict separation of duties between the software and the database engineer
- Rigorous backup and recovery processes, regularly tested
- Well regulated security procedures, regularly audited
- Expensive database software with strong durability guarantees
- Underlying expensive storage with redundancy of all components
- Extensive controls on changes and administrative tasks

In teams with collaborative cultures, the strict separate of duties can become not only burdensome, but restrictive of innovation and velocity. In chapter 14, Schema and Data Management, we will discuss ways to create safety nets and reduce the need for separation of duties. Additionally, these environments focus more on testing, automation and impact mitigation than extensive change controls.

More often than ever, architects and engineers are choosing open source datastores that cannot guarantee durability the way that something like Oracle might have in the past. Sometimes, that relaxed durability gives needed performance benefits to a team looking to scale quickly. Choosing the right datastore, and understanding the impacts of those choices is something we look at Chapter 16, The Datastore Field Guide. Recognizing that there are multiple tools based on the data you are managing, and choosing effectively is rapidly becoming the norm.

Underlying storage has also undergone a significant change as well. In a world where systems are often virtualized, network and ephemeral storage is finding a place in database design. We will discuss this further in chapter 5, Infrastructure Engineering.

Production Datastores on Ephemeral Storage

In 2013, Pinterest moved their MySQL database instances to run on ephemeral storage in Amazon Web Services (AWS). Ephemeral storage effectively means that if the compute instance fails or is shut down, anything stored on disk is lost. The ephemeral storage option was chosen because of consistent throughput and low latency.

Doing this required substantial investment in automated and rock-solid backup and recovery, as well as application engineering to tolerate the disappearance of a cluster while rebuilding nodes. Ephemeral storage did not allow snapshots, which meant that the restore approach was full database copies over the network rather than attaching a snapshot in preparation for rolling forward of the transaction logs.

This shows that you can maintain data safety in ephemeral environments with the right processes and the right tools!

The new approach to data protection might look more like this:

- Responsibility of the data shared by cross-functional teams.
- Standardized and automated backup and recovery processes blessed by DBRE.
- Standardized security policies and procedures blessed by DBRE and Security.
- All policies enforced via automated provisioning and deployment.
- Data requirements dictate the datastore, with implicit understanding of durability needs part of the decision making process.
- Reliance on automated processes, redundancy and well practiced procedures rather than expensive, complicated hardware.
- Changes incorporated into deployment and infrastructure automation, with focus on testing, fallback and impact mitigation.

Self-Service for Scale

A talented DBRE is a rarer commodity than an SRE by far. Most companies cannot afford and retain more than one or two. So, we must create the most value possible, which comes from creating self-service platforms for teams to use. By setting standards, and providing tools, teams are able to deploy new services and make appropriate changes at the required pace, without serializing on an overworked database engineer. Examples of these kinds of self-service methods include:

- Ensure the appropriate metrics are being collected from data stores by providing the write plugins.
- Building backup and recovery utilities that can be deployed for new data stores.
- Defining reference architectures and configurations for data stores that are approved for operations, and can be deployed by teams.
- Working with security to define standards for data store deployments.
- Building safe deployment methods and test scripts for database changesets to be applied.

In other words, the effective DBRE functions by empowering others and guiding them, not functioning as a gatekeeper.

Elimination of Toil

This phrase is used often by the Google SRE teams, and is discussed in Chapter 5 of the Google SRE book. In the book, toil is defined as:

> Toil is the kind of work tied to running a production service that tends to be manual, repetitive, automatable, tactical, devoid of enduring value, and that scales linearly as a service grows.

Effective use of automation and standardization is necessary to ensure that the DBREs are not overburdened by toil. Throughout this book, we will be bringing up examples of DBRE specific toil, and the approaches to mitigation of this.

Manual Database Changes

In many customer environments, database engineers are asked to review and apply DB changes, which can include modifications to tables or indexes, the addition, modification or removal of data, or any other number of tasks. Everyone feels reassured that the DBA is

applying these changes, and monitoring the impact of the changes in real time.

At one customer site, the rate of change was quite high, and those changes were often impactful. We ended up spending about 20 hours a week applying rolling changes throughout the environment. Needless to say, the poor DBA who ended up spending half of their week running these repetitive tasks became jaded and ended up quitting.

Faced with a lack of resources, management finally allowed the DB team to build a rolling schema change automation utility that could be used by software engineers once the changeset had been reviewed and approved by one of the database engineers. Soon, everyone trusted the tool, and monitoring, to introduce change, paving the way for the DBRE team to focus more time on integrating these processes with the deployment stack.

Databases are Not Special Snowflakes

Our systems are no more or less important than any other components serving the needs of the business. We must strive for standardization, automation, and resilience. Critical to this is the idea that the components of database clusters are not sacred. We should be able to lose any component, and efficiently replace it without worry. Fragile data stores in glass rooms are a thing of the past.

The metaphor of pets vs cattle is often used to show the difference between a special snowflake and a commodity service component. Original attribution goes to Bill Baker, Microsoft Distinguished Engineer. A pet server, is one that you feed, care for and nurture back to health when it is sick. It has a name -- at Travelocity in 2000, our servers were Simpsons characters. Our two SGI servers running Oracle were named Patty and Selma. I spent so many hours with those gals on late nights. They were high maintenance!

Cattle servers have numbers, not names. You don't spend time customizing servers, much less logging on to individual hosts. When they show signs of sickness, you cull them from the herd. You should, of course, keep those culled cattle around for forensics, if you are seeing unusual amounts of sickness. But, we'll refrain from mangling this metaphor any further.

Data stores are some of the last hold outs of "pethood". After all, they hold "The Data", and simply cannot be treated as replaceable cattle, with short lifespans and complete standardizations. What about the special replication rules for our reporting replica? What about the different config for the master's redundant standby?

Eliminate the Barriers Between Software and Operations

Your infrastructure, configurations, data models and scripts are all part of software. Study and participate in the software development lifecycle as any engineer would. Code, test, integrate, build, test and deploy. Did we mention test?

This might be the hardest paradigm shift for someone coming from an operations and scripting background. There can be an organizational impedance-mismatch in the way software engineers navigate an organization and the systems and services built to meet that organization's needs. Software engineering organizations have very defined approaches to developing, testing and deploying features and applications.

In a traditional environment, the underlying process of designing, building, testing and pushing infrastructure and related services to production was separate between SWE, SE and DBA. The paradigm shifts discussed previously are pushing for removal of this impedance mismatch, which means DBREs and Systems Engineers find themselves needing to use similar methodologies to do their jobs.

DBREs might also find themselves embedded directly in a software engineering team, working in the same code base, examining how code is interacting with the data stores, and modifying code for performance, functionality and reliability. The removal of this organizational impedance creates an improvement in reliability, performance and velocity an order of magnitude greater than traditional models, and DBREs must adapt to these new processes, cultures and tooling.

Software Engineers Must Learn Operations!

Too often, operations folks are told to learn to "code or to go home". While I do agree with this, the reverse must be true as well. Software engineers who are not being pushed and led to learn operations and infrastructure principles and practices will create fragile, non-performant and potentially insecure code. The impedance mismatch only goes away if all teams are brought to the same table!

Our goal for you, the reader of this book, is that you gain a framework of principles and practices for the design, building and operating of data stores within the paradigms of reliability engineering and devops cultures. Our hope is that you can take this knowledge, and apply it to any database technology, or environment that you are asked to work in, at any stage in your companies growth.

Operations Core Overview

The next section will be all about operations core competencies. These are the building blocks for designing, testing, building and operating any system with scale and reliability requirements that are not trivial. This means that if you want to be a database engineer, you need to know these things. Each chapter will be fairly generic, but where database specific sections make sense, we dive into these more specialized areas.

Operations at a macro level is not a role. Operations is the combined sum of all of the skills, knowledge and values that your company has built up around the practice of shipping and maintaining quality systems and software. It's your implicit values as well as your explicit values, habits, tribal knowledge, reward systems. Everybody from tech support to product people to CEO participates in your operational outcomes.

Too often, this is not done well. So many companies have an abysmal ops culture that burns out whoever gets close to it. This can give the discipline a bad reputation, which many folks think of when they think of operations jobs, whether in systems, database or network. Despite this, your ops culture is an emergent property of how your org executes on its technical mission. So if you go and tell us that your company doesn't do any ops, we just won't buy it.

Perhaps you are a software engineer, or a proponent of infrastructure and platforms as a service. Perhaps you are dubious that operations is necessity for the intrepid database engineer. The idea that serverless computing models will liberate software engineers from needing to think or care about operational impact is flat out wrong. It is actually the exact opposite. In a brave new world where you have no embedded operations teams —where the people doing operations engineering for you are Google SREs and AWS systems engineers and PagerDuty and DataDog and so on? This is a world where application engineers need to be much better at operations and architecture and performance than they currently are.

Hierarchy of Needs

Some of you will be coming at this book with experience in enterprises, and some in startups. As we approach and consider systems, it is worth thinking about what you would do on day one of taking on the responsibility of operating a database system. Do you have backups? Do they work? Are you sure? Is there a replica you can fail over to? Do you know how to do that? Is it on the same power strip, router, hardware, or availability zone as the primary? Will you know if the backups start failing somehow? How?

In other words, we need to talk about a hierarchy of database needs.

For humans, Maslow's hierarchy of needs is a pyramid of desire that must be satisfied for us to flourish: physiological survival, safety, love and belonging, esteem, and self-actualization. At the base of the pyramid are the most fundamental needs, like survival. Each level roughly proceeds to the next -- survival before safety, safety before love and belonging, and so forth. Once the first four levels are satisfied, we reach self-actualization, which is where we can safely explore and play and create and reach the fullest expression of our unique potential. So that's what it means for humans. Let's apply this as a metaphor for what databases need.

Survival and Safety

There are three primary parts to consider here. Backups, replication and failover. *Do you have a database*? *Is it alive*? *Can you ping it?*, and *is your application responding*? *Does it get backed up?*, *Will restores work?*, and *how will you know if this stops being true*?

Is your data safe? Are there multiple live copies of your data? Do you know how to do a failover? Are your copies distributed across multiple physical availability zones or multiple power strips and racks? Are your backups consistent, can you restore to a point in time, will you know if your data gets corrupted? How? Plan on exploring this much more in the backup and recovery section.

This is also the time when you start preparing for scale. Scaling prematurely is a fool's errand, but you should consider *sharding*, growth and scale now as you determine ids for key data objects, storage systems and architecture.

Scaling Patterns

We will discuss scale quite frequently. Scalability is the capability of a system or service to handle increasing amounts of work. This might be *actual* ability, because everything has been deployed to support the growth, or it might be *potential* ability, in that the building blocks are in place to handle the addition of components and resources required to scale. There is a general consensus that scale has 4 pathways that will be approached.

- Scale vertically, via resource allocation. *aka scale up*
- Scale horizontally, by duplication of the system or service. *aka scale out*
- Separate workloads to smaller sets of functionality, to allow for each to scale independently. *aka functional partitioning*
- Split specific workloads into partitions that are identical, other than the specific set of data that is being worked on. *aka sharding*

The specific aspects of these patterns will be reviewed in Chapter 5, Infrastructure Engineering.

Love and belonging

Love and belonging is about making your data a first-class citizen of your software engineering processes. It's about breaking down silos between your databases and the rest of your systems. This is both technical and cultural, which is why you could also just call this section the "devops section". At a high level, it means that managing your databases should look and feel (as much as possible) like managing the rest of your systems. It means that you culturally encour-

age fluidity and cross-functionality. The love and belonging phase is where you slowly stop logging in and performing cowboy commands as root.

It is here where you begin to use the same code review and deployment practices. Database infrastructure and provisioning should be part of the same process as all other architectural components. Working with data should feel consistent to all other parts of the application, which should encourage anyone to feel they can engage with and support the database environment.

Resist the urge to instill fear in your developers. It's quite easy to do and quite tempting to do it because it feels better to feel like you have control. It's not — and you don't. It's much better for everyone if you invest that energy into building guard rails so it's harder for anyone to accidentally destroy things. Educate and empower everyone to own their own changes. Don't even talk about preventing failure, as such is impossible. In other words, create resilient systems and encourage everyone to work with the datastore as much as possible.

Guardrails at Etsy

Etsy introduced a tool called *Schemanator* to apply database changes, otherwise known as change-sets, safely to their production environments. Multiple guardrails were put in place to empower software engineers to apply these changes. These guardrails included:

- Change-set heuristic reviews, validating standards had been followed in schema definitions.
- Change-set testing, to validate the scripts run successfully.
- Preflight checks, to show the engineer the current cluster status.
- Rolling changes, to run impactful changes on "out of service" databases.
- Breaking workflows into subtasks, to allow for cancelling out when problems occur that can not be predicted.

You can read more about this at Etsy's Blog: [1]

Esteem

Esteem is the last of the deficiency needs. For humans, this means respect and mastery. For databases, this means things like observability, debuggability, introspection and instrumentation. It's about being able to understand your storage systems themselves, but also being able to correlate events across the stack. Again there are two aspects to this stage: one of them is about how your production services evolve through this phase, and the other is about your humans.

Your services should tell you if they're up or down or experiencing error rates. You should never have to look at a graph to find this out. As your services mature, the pace of change slows down a bit and your trajectory is more predictable. You're running in production so you're learning more every day about your storage system's weaknesses, behaviors,failure conditions. This can be compared to teenager years for data infrastructure. What you need more than anything is visibility into what is going on. The more complex your product is, the more moving pieces there are and the more engineering cycles you need to allocate into developing the tools you need to figure out what's happening.

You also need knobs. You need the ability to selectively degrade quality of service instead of going completely down. e.g.:

- Flags where you can set the site into read-only mode
- Disabling certain features
- Queueing writes to be applied later
- The ability to blacklist bad actors or certain endpoints

Your humans have similar but not completely overlapping needs. A common pattern here is that teams will overreact once they get into production. They don't have enough visibility, so they compensate by monitoring everything and paging themselves too often. It is easy to go from zero graphs to literally hundreds of thousands of graphs, 99% of which are completely meaningless. This is not better. It can actually be worse. If it generates so much noise that your humans

1 https://codeascraft.com/2013/01/11/schemanator-love-child-of-deployinator-and-schema-changes/

can't find the signal, so they are reduced to tailing log files and guessing again, it's as bad or worse than not having the graphs.

This is where you can start to burn out your humans by interrupting them, waking them up and training them not to care or act on alerts they do receive. In the early stages, if you're expecting everyone to be on call, you need to document things. When you're bootstrapping and you have shared on call and you're pushing people outside of their comfort zones, give them a little help. Write minimally effective documentation and procedures.

Self-actualization Just like every person's best possible self is unique, every organization's self-actualized storage layer is unique. The platonic ideal of a storage system for Facebook doesn't look like the perfect system for Pinterest, or Github, let alone a tiny startup. But just like there are patterns for healthy, self-actualized humans (doesn't throw tantrums in the grocery store, they eat well and exercise), there are patterns for what we can think of as healthy, self-actualized storage systems.

It means that your data infrastructure helps you get where you're trying to go. Your database workflows are not an obstacle to progress. Rather, they empower your developers to get work done and help save them from making unnecessary mistakes. Common operational pains and boring failures should remediate themselves and keep the system in a healthy state without needing humans to help. It means you have a scaling story that works for your needs. Whether that means 10x'ing every few months or just being rock solid and stable and dumb for three years before you need to worry about capacity. Frankly, you have a mature data infrastructure when you can spend most of your time thinking about other things. Fun things. Like building new products, or anticipating future problems instead of reacting to current ones.

It's okay to float back and forth between levels over time. The levels are mostly there as a framework to help you think about relative priorities, like making sure you have working backups is WAY more important than writing a script to dynamically re-shard and add more capacity. Or if you're still at the point where you have one copy of your data online, or you don't know how to fail over when your primary dies, you should probably stop whatever you're doing and figure that out first.

Operational Core Competencies

Each chapter in the next section will focus on a specific core competency that any operations or reliability engineer should practice as part of their craft. A quick enumeration here is on order, to allow you to jump around as necessary. If you have been doing ops for awhile, these sections might be old hat. If so, feel free to jump to section 2, Data stores and Data. It won't hurt our feelings!

The operational core can be broken into the following list:

- Chapter 2: Service Level Management
- Chapter 3: Risk Management
- Chapter 4: Operational Visibility
- Chapter 5: Infrastructure Engineering
- Chapter 6: Release and Change Management
- Chapter 7: Backup and Recovery
- Chapter 8: Security
- Chapter 9: Incident Management
- Chapter 10: Performance Management
- Chapter 11: Distributed Systems

With that being said, let's move bravely forward, intrepid engineer!

CHAPTER 2
Operational Visibility

Visibility (often referred to as monitoring) is the cornerstone of the craft of database reliability engineering. Operational visibility means that we have awareness of the working characteristics of a database service due to the regular measuring and collection of data points about the various components. Why is this important? Why do we need operational visibility?

Break/Fix and Alerting

We need to know when things break, or about to break, so that we can fix them before violating our service level objectives.

Performance and Behavior Analysis

We need to understand the latency distribution, including outliers, in our applications, and we need to know the trends over time. This data is critical to understand the impact of new features, of experiments and of optimization.

Capacity Planning

Being able to correlate user behavior and application efficiency to real resources (CPU, network, storage, throughput, memory, etc...) is critical to making sure you never get hit by a lack of capacity in a critical business moment.

Debugging and Post-Mortems

Moving fast means things do break. Good operational visibility means you are able to rapidly identify failure points and optimiza-

tion points to mitigate future risk. Human error is never a root-cause, but systems can always be improved upon and made to be more resilient.

Business Analysis

Understanding how your business functionality is being utilized can be a leading indicator of issues, but is also critical for everyone to see how people are using your features, and how much value vs. cost is being driven.

Situational Awareness

By having events in the infrastructure and application register themselves in your operational visibility stack, you can rapidly correlate changes in workload, behavior and availability. Examples of these events are application deployments, infrastructure changes and database schema changes.

As you can see, pretty much every facet of your organization requires true operational visibility (*opviz)*. Our goal in this chapter is to help you understand observability in the architectures you will be working with. While there is no one set of tools we will be espousing, there are principles, a general taxonomy and usage patterns to learn. We will do this with numerous case studies and example approaches. First, let's consider the evolution of opviz from traditional approaches to those utilized today.

Traditional Monitoring

Traditional monitoring systems can be characterized as follows:

- Hosts are generally servers, rather than virtual instances or containers, and they have a long lifetime, measured in months, sometimes years.
- Network and hardware addresses are stable.
- There is a focus on systems rather than services.
- Utilization, and static thresholds (aka symptoms) are monitored more than customer facing metrics (Service level indicators).
- Different systems for different silos (network, DB etc...).
- Low granularity (1 or 5 minute intervals).
- Focus on collection and presentation, not analysis.

- High administration overhead and often more fragile than the services they monitor.

et's summarize this into a "traditional monitoring" mindset. In traditional environments, database administrators focused on questions such as "is my database up", and "is my utilization sustainable". They were not considering how the behavior of the database was impacting latency to users by looking at histograms of the latency metrics to understand distribution and potential outliers. They often wanted to, but they didn't have tools to make this happen.

So, operational visibility is a big deal! If such is the case, we need some rules on how we design, build and utilize this critical process.

The New Rules of Operational Visibility

Modern operational visibility assumes that data stores are distributed, often massively. It recognizes that collection, and even presentation of data, is not as crucial as the analysis. It always asks, and hopefully leads to rapid answering, of two questions: *how is this impacting my service level objectives*? and *how is this broken, and why?* In other words, rather than treating your opviz stack as a set of utilities to be relegated to the ops team, it must be designed, built and maintained as a business intelligence platform. This means it must be treated the same way a data warehouse, or big data platform would be. The rules of the game have changed to reflect this.

Treat your opviz system like a business intelligence system: When designing a BI system, you begin by thinking about the kinds of questions your users will be asking, and building out from there. Consider your users needs for data latency (*how quickly is data available?),* data resolution (*how deep down can the user drill?)* and data availability. In other words, you are defining SLOs for your opviz service. Refer to Chapter 2.

The hallmark of a mature opviz platform is that it can provide not only the state of the infrastructure running the application, but also the behavior of the applications running on that infrastructure. Ultimately, this should also be able to show anyone how the business is doing, and how that is being impacted by the infrastructure and applications that business is relying on. With that in mind, the opviz platform must support operations and database engineers, software engineers, business analysts and executives.

Distributed and ephemeral environments are trending towards the norm: We've already discussed the fact that our database instance lifecycles are trending down with the adoption of virtual infrastructures. While they are still much longer-lived than other infrastructure components, we still must be able to gather metrics for services consisting of short lived components that are aggregated rather than individual database hosts.

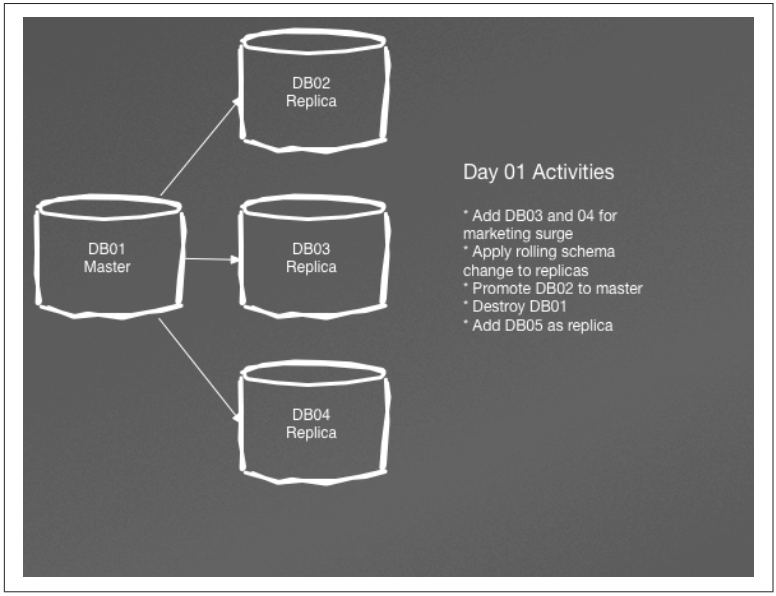

Figure 2-1. Typical Master/Replica Setup

Even in figure X above, which is a fairly stable master/replica set up for a relational datastore, numerous activities can occur in one day. By the end, we can see a completely new setup.

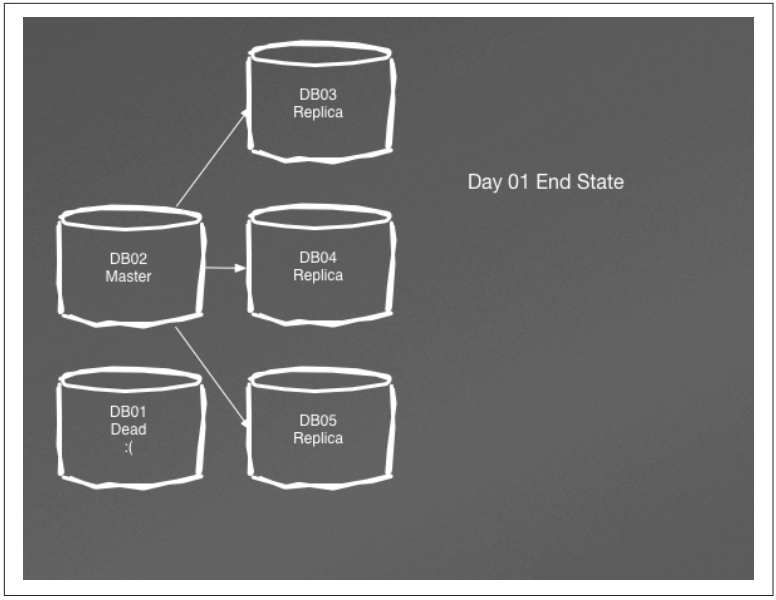

Figure 2-2. End of Day 1

This kind of dynamic infrastructure requires us to store metrics based on roles, rather than hostnames or IPs. So instead of storing a set of metrics as DB01, we would add metrics to the "master" role, allowing us to see all master behavior even after switching to a new master.

Store at high resolutions for key metrics: As reviewed in Chapter 2, high resolution is critical for understanding busy application workloads. At a minimum, anything related to your SLOs should be kept at 1 second or lower sampling rates to ensure you understand what is going on in the system. A good rule of thumb is to consider if the metric has enough variability to impact your SLOs in the span of 1-10 seconds, and to base granularity on that.

For instance, if you are monitoring a constrained resource, such as CPU, you would want to collect this data at 1s or smaller sample, as CPU queues can build up and die down quite quickly. With latency SLOs in the milliseconds, this data must be good enough to see if CPU saturation is the reason your application latency is being impacted. Database connection queues is another area that can be missed without very frequent sampling.

Conversely, for low changing items such as disk space, service up/down etc... you can measure these in the 1 minute or higher sampling rates without losing data. High sample rates consume a lot of resources, and you should be judicious in using them. Similarly, you should probably keep less than 5 different sampling rates to maintain simplicity and structure in your opviz platform.

For an example of the impacts of a sampling rate that is too long, let's consider the following graph.

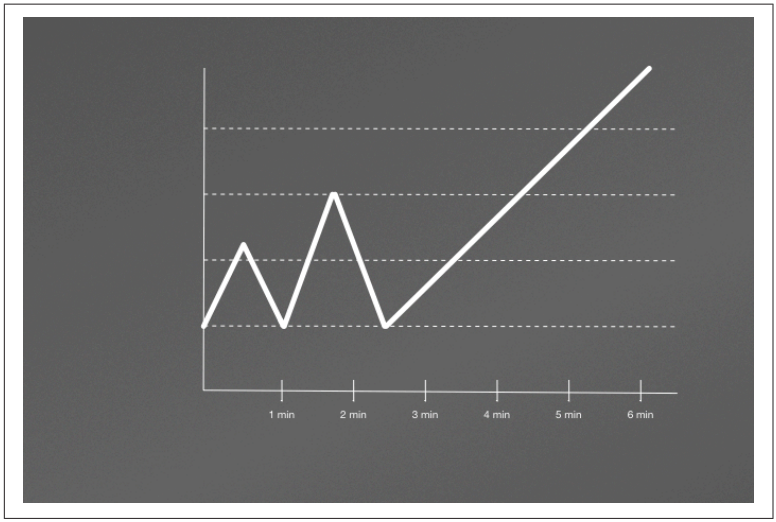

Figure 2-3. Real workload showing spikes

In figure X, we see two spikes followed by a long ascension. Now, if we are sampling this metric at one minute intervals, the graph would look like the following:

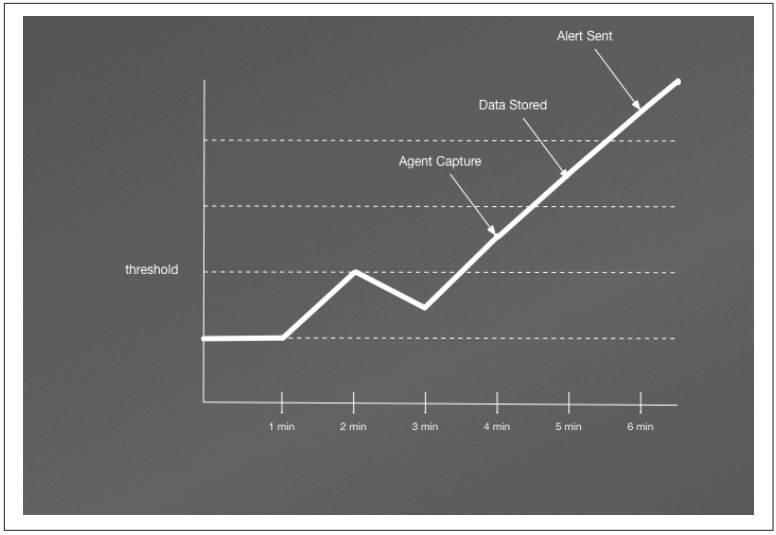

Figure 2-4. Workload visualized via one minute sampling.

You'll notice that we don't even see one spike, and the second looks much less impactful. In fact, our alerting threshold is not even exceeded until minute three. Assuming one minute schedule for storage and for alert rules checking, we wouldn't even send an alert to an operator until 7.5 minutes after the rule was violated!

Keep your architecture simple: It is not unusual for a growing platform to have 10,000 or more metrics being checked at various levels of granularity for any number of instances/servers that are going in and out of service in the infrastructure at any time. Your goal is to be able to rapidly answer the questions discussed above, which means you must continually push for reducing the ratio of signal to noise. This means being ruthless in the amount of data you allow into your system, particularly at the human interaction points, such as presentation and alerting.

Monitor all the things has been the watch cry for quite awhile, and was a reaction to environments where monitoring was sparse and ad hoc. The truth though, is that distributed systems and multi-service applications create too many metrics. Early stage organizations do not have the money or time to manage this amount of monitoring

data. Larger organizations should have the knowledge to focus on what is critical to their systems.

Focusing your metrics

Focus initially on metrics directly related to your SLOs. This can also be called the critical path. Based on Chapter 2, what are the metrics that should be a priority for getting into your opviz platform?

- **Latency**: Client calls to your service. How long do they take?
- **Availability:** How many of your calls result in errors?
- **Call Rates:** How often are calls sent to your service?
- **Utilization:** Looking at a service, you should know how critical resources are being utilized to ensure quality of service and capacity.

Of course, as the DBRE, you will want to immediately start breaking out these metrics into the datastore subsystems. That only makes sense, and its a natural evolution that will be discussed in the "what to measure" section of this chapter.

Simplicity and signal amplification also includes standardization. This means standardizing templates, resolutions, retentions and any other knobs and features presented to engineers. By doing so, you ensure that your system is easy to understand, and thus user friendly for answering questions and identifying issues.

Remembering these four rules will help keep you on track with designing and building incredibly valuable and useful monitoring systems. If you find yourself violating them, ask yourself why. Unless you have a really good answer, please consider going back to the foundation.

An Opviz Framework

Let's be clear, we could right a whole book on this stuff. But, we don't have time. As you begin to gather and prepare the right data to go into the opviz platform for you to do your job, you should be able to recognize a good platform, and advocate for a better platform. This is our goal for this section.

Let's think of our opviz platform as a great big, distributed I/O device. Data is sent in, routed, structured and eventually comes out the other side in useful ways that help you to understand your systems better, to identify behaviors caused by broken or soon to be broken components and to meet your SLOs.

- Data is generated by agents on clients, and sent to a central collector for that datatype. (i.e. metrics to *Sensu* or *CollectD*, events to *Graphite*, logs to *Logstash* or *Splunk*)
  - Occasionally a centralized monitoring system (like *Sensu* or *Nagios)* will execute checks via a pull method in addition to the push method above.
- The collectors store data (in systems like *Graphite*, *InfluxDB* or *ElasticSearch*) or forward to event routers/processors (such as *Riemann).*
- Event routers uses heuristics to send data to the right places.
- Data output includes long-term storage, graphs and dashboards, alerts, tickets, external calls etc...

These outputs are where we get the true value of our opviz stack. And, it is where we will focus as we go through this chapter.

Data In

In order to create outputs, we require good inputs. Wherever possible, use data already generated by your environments rather than artificial probes. When you simulate a user by sending a request into the system, it is called blackbox monitoring. Blackbox monitoring can be effective if you have low traffic periods, or items that just don't run frequently enough for you to monitor. But, if you are generating sufficient data, getting real metrics, aka whitebox monitoring, is infinitely more appealing.

One benefit of this approach, is that anything creating data becomes an agent. A centralized, monolithic solution that is generating checks and probes will have challenges scaling as you grow. Instead, you've distributed this job across your entire architecture. This kind of architecture also allows new services, components etc... to easily register and deregister with your collection layer, which is a good thing based on our opviz rules. That being said, there are still times when having a monitoring system that can perform remote executions as a pull can be valuable, such as checking to see if a service is

up, monitoring to see if replication is running or checking to see if an important parameter is enabled on your database hosts.

Signal to Noise

More and more, we rely on larger datasets to manage distributed systems. It has gotten to the point where we are creating big data systems in order to manage the data we collect on our applications and the infrastructures supporting them. As discussed earlier in this chapter, the use of data science and advanced math is one of the greatest shortcomings to today's observability stacks. To effectively identify the signal from all of this noise, we must rely on machines to sort the signal from the noise.

This area is still very theoretical, as most attempts at good anomaly detection have proven unusable with multi-modal workloads and continuous changes to workloads due to rapid feature development and user population changes. A good anomaly detection system will help identify activity that does not fit the norm, thus directing people immediately to problems, which can reduce MTTR by getting a higher amount of signal to noise.

Some systems worth evaluating are listed below:

- Reimann
- Anomaly.io
- VividCortex
- Dynatrace
- Circonus

OK, so we are looking to send all of this valuable data to our opviz platform. What kind of data are we talking about anyway?

Telemetry/Metrics

Ah metrics! So diverse, so ubiquitous. A metric is the measurement of a property that an application or a component of your infrastructure possesses. Metrics are observed periodically, to create a time series that contains the property or properties, the timestamp and the value. Some properties that might apply include the host, service, datacenter etc... The true value of this data comes in observing it over time through visualizations such as graphs.

Metrics often have mathematical functions applied to them in order to assist humans in deriving value from their visualizations. Some of these functions are listed below. These functions do create more value, but it is important to remember that they are derived data, and that the raw data is just as critical. If you're tracking means per minute, but do not have the underlying data, you won't be able to create means on larger windows such as hours or days.

- Count
- Sum
- Average
- Median
- Percentiles
- Standard Deviation
- Rates of Change
- Distributions

NOTE

Visualizing Distributions

Visualizing a distribution is an very valuable to look at the kind of data that is often generated in web architectures. This data is rarely distributed normally, and often has long tails. It can be challenging to see this in normal graphs. But, with the ability to generate distribution maps over buckets of time, you enable new styles of visualization, such as histograms over time and flame graphs that can really help a human operator visualize the workloads that are occurring in your systems.

Metrics are the source for identifying symptoms of underlying issues, and thus are crucial to early identification and rapid resolution of any number of issues that might impact your SLOs.

Events

An event is a discrete action that has occurred in the environment. A change to a config is an event. A code deployment is an event. A database master failover is an event. Each of these can be signals that are used to correlate symptoms to causes.

Logs

A log is created for an event. So, you can consider log events to be a subset of an event. Operating systems, databases and applications all create logs during certain events. Unlike metrics, logs can provide additional data and context to something that has occurred. For instance, a database query log can tell you when a query was executed, important metrics about that query and even the database user who executed it.

Data Out

So, data is flowing into our systems, which is nice and all, but doesn't help us answer our questions or meet our SLOs, now does it? What should we be looking to create in this opviz framework?

Alerts

An alert is an interrupt to a human that tells them to drop what they are doing and investigate a rules violation that has caused the alert to be sent. We'll be digging much more deeply into this in Chapter 9, Incident Management. This is an expensive operation, and should only be utilized when SLOs are in imminent danger of violation.

Tickets/Tasks

When work must be done, but there is not an imminent disaster, the output of monitoring should be tickets/tasks that go in engineer queues for work.

Notifications

Sometimes you just want to record that an event has occurred to help create context for folks. For instance, when code deploy events are registered. Notifications will often go to a chat room, or a wiki or collaboration tool to make it visible without interrupting workflow.

Automation

There are times when data, particularly utilization data, advises of the need for more or less capacity. Auto-scaling groups can be called to modify resource pools in such cases. This is but one example of automation as an output of monitoring.

Visualization

Graphs are one of the most common outputs of opviz. These are collected into dashboards that suit the needs of a particular user

community, and are a key tool for humans to perform pattern recognition on.

Bootstrapping your Monitoring

If you are like most rational people, you might be starting to feel overwhelmed by all of these things that should be happening. That is normal! This is a good time to remind you that everything we build here is part of an iterative process. Start small, let things evolve and add in more as you need it. Nowhere is this more true than in a startup environment.

As a brand new startup, you start with zero. Zero metrics, zero alerting, zero visibility, just a bunch of engineers cranking out overly optimistic code. Many startups somehow end up with an instance somewhere in a public cloud that was a prototype or testbed and then it somehow turned into their master production database. Head meet desk!

Maybe you just got hired in as the first ops/database engineer at a young startup and you're taking stock of what the software engineers have built around monitoring or visibility, and it's effectively … zero.

Sound familiar? If you have any experience with startups, it should. It's nothing to be ashamed of! This is how startup sausage gets made. A startup that started out by building out an elaborate operational visibility ecosystem in advance of their actual needs would be a stupid startup. Startups succeed by focusing hard on their core product, iterating rapidly, aggressively seeking out customers, responding to customer feedback and production realities, and making hard decisions about where to spend their precious engineering resources. Startups succeed by instrumenting elaborate performance visibility systems as soon as they need them, not before. Startups fail all the time, but usually not because the engineers failed to anticipate and measure every conceivable storage metric in

advance. What we need to start with is a *Minimum Viable Monitoring Set*.

Enumerating Moving Parts of the Database

You can think of your data as a stream from clients to databases. At the highest of levels, the database exists to take data in, to hold data and to serve data back.

- Data in client memory.
- Data across the wire between client and datastore.
- Data in your databases memory structures.
- Data in your OS and disk memory structures.
- Data on your disks.
- Data in backups and archival.

Everything we seek to measure about our databases can boil down to:

- How long does it take to get data out, and why does it take that long?
- How long does it take to put data in, and why does it take that long?
- Is the data safely stored, and how is it stored?
- Is the data available in redundant locations in case primary retrieval fails?

Of course this is quite simplistic, but it is a good top level structure to think about while we dig into the following sections.

There are an infinite number of metrics you can monitor, between the db, system, storage, and various application layers. In the physiological needs state, you should be able to tell if your database is up or down. As you work towards fulfilling the "esteem" state, you start by monitoring other symptoms that you have identified that correlate with real problems, such as connection counts, or lock percentages. One common progression looks like:

- Monitor if your databases are up or down (pull checks)
- Monitor overall latency/error metrics and end-to-end health checks (push checks)
- Instrument the application layer to measure latency/errors for every database call (push checks)

- Gather as many metrics as possible about the system, storage, db and app layers, whether you think they will be useful or not. Most operating systems, services and databases will have plugins that are fairly comprehensive.
- Create specific checks for known problems e.g. have lost x% of database nodes, global lock % is high (do this iteratively as well as proactively, see Chapter 3, Risk Management)

Sometimes you can shortcut to the "esteem" level by plugging in third-party monitoring services like VividCortex, Circonus, HoneyComb or NewRelic. But it's kind of a hack if you're storing these db metrics in a system separate from the rest of your monitoring. Storing in disparate systems makes it more challenging to correlate symptoms across multiple monitoring platforms. We're not saying this is bad or you shouldn't do this; elegant hacks can take you a really long way! But the "self-actualization" phase generally includes consolidating all monitoring feeds into a single source of truth.

Okay. Now that you've made sure that your company won't go out of business when you lose a disk or an engineer makes a typo, you can start asking yourself questions about the health of your service. As a startup, the key questions to ask yourself are: is my data safe, is my service up, are my customers experiencing pain? This is your minimum viable product, from a monitoring perspective.

Is the data safe?

For any mission-critical data that you truly care about, you should avoid running with less than three live copies. That's one primary and 2+ secondaries for leader-follower data stores like MySQL or MongoDB, or a replication factor of 3 for distributed data stores like Cassandra or Hadoop. Because you never, ever want to find yourself in a situation where you have a single copy of any data you care about, ever. Which means you need to be able to lose one instance while still maintaining redundancy. Even when you are penny-pinching and worrying about your run rate every month as a baby startup. Mission-critical data is not the right place to cut those costs. We will discuss availability architecture in Chapter 5, Infrastructure Engineering.

But not all data is equally precious! If you can afford to lose some data, or if you could reconstruct the data from immutable logs if necessary, then running with n+1 copies is perfectly ok. This is a

judgment call -- only you can know how critical and how irreplaceable each data set is for your company, and how tight your financial resources are. You also need backups, and you need to regularly validate that the backups are restorable and that the backup process is completing successfully. If you aren't monitoring that your backups are good, you cannot assume that your data is safe.

Sample Data Safety Monitors

- 3 data nodes up
- Replication threads running
- Replication on at least one node < 1 second behind
- Last Backup success
- Last Automated replica rebuild from backups successful

Is the service up?

End-to-end checks are the most powerful tool in your arsenal, because they most closely reflect your customer experience. You should have a top-level health check that exercises not just the aliveness of the web tier or application tier, but all the db connections in the critical path. If your data is partitioned across multiple hosts, the check should fetch an object on each of the partitions, and it should automatically detect the full list of partitions or shards so you do not need to manually add new checks any time you add more capacity.

However -- and this is important -- you should have a simpler aliveness check for your load balancers to use that does not exercise all your database connections! Otherwise you can easily end up health-checking yourself to death.

Excessive Health Checking

Charity once worked on a system where a haproxy health check endpoint did a simple SELECT LIMIT 1 from a mysql table. One day, they doubled the capacity of some stateless services and accidentally took the whole site down. Over 95% of all db queries were those stupid health checks. Don't do that!

Speaking of lessons learned the hard way, you should always have some off-premise monitoring -- if nothing else, an offsite health check for your monitoring service itself. It doesn't matter how amazing and robust your on-premise monitoring ecosystem is if your data center or cloud region goes down and takes your whole monitoring apparatus with it. Setting up an external check for each major product or service, as well as a health check on the monitoring service itself is a good best practice.

NOTE

Sample Database Availability Monitors

- Health check at the application level that queries all front end datastores.
- Query run against each partition in each datastore member, for each datastore.
- Imminent capacity issues
    - Disk Capacity
    - Database Connections
- Error log scraping
    - DB Restarts (faster than your monitor!)
    - Corruption

Are the consumers in pain?

Ok, you are monitoring that your service is alive. The patient has a heartbeat. Good job!

But what if your latency subtly doubles or triples, or what if 10% of your requests are erroring in a way that cleverly avoids triggering your health check? What if your database is not writable, but can be read from, or the replicas are lagging which is causing your majority write concern to hang, or your RAID array has lost a volume and is running in degraded mode, or you have an index building, or you are experiencing hot spotting of updates to a single row?

Well, this is why systems engineering and databases in particular are so much fun. There are infinite ways your systems can fail and you can probably only guess about five percent of them in advance. Yay!

This is why you should gradually develop a library of comprehensive high-level metrics about the health of the service. Health checks, error rates, latency. Anything that materially impacts and

disrupts your customer experience. And then? Go work on something else for a while and see what breaks.

We are almost entirely serious. As discussed in Chapter 3, Risk Management, there is only so much to be gained by sitting around trying to guess how your service is going to break. You just don't have the data yet. You may as well go build more things and wait for things to break, and then pay a lot of attention when things actually start failing.

Now that we've provided a bootstrapping method, and an evolution method. Let's breakdown what you should be measuring, with a focus on what you as the DBRE need.

Instrumenting the Application

Your application is the first place to start. While we can measure most things at the datastore layer, the first leading indicators of problems should be changes in user and application behavior. Between application instrumentation by your engineers, and application performance management solutions (APM) such as New Relic and AppDynamics, you can get a tremendous amount of data for everyone in the organization.

- You should already be measuring and logging all requests and responses to pages or API endpoints.
- You should also be doing this to all external services, which includes databases, search indexes and caches!
- Any jobs, independent workflows, etc.. should be similarly monitored.
- Any independent, reusable code like a method or function that interacts with databases, caches etc... should be similarly instrumented.
- Monitor how many database calls are executed by each endpoint, page or function/method.

Tracking the data access code (such as SQL calls) called by each operation to allow for rapid cross-referencing to more detailed query logs within the database. This can prove challenging with ORMs, where SQL is dynamically generated.

NOTE

SQL Comments

When doing SQL tuning, a big challenge is mapping SQL running in the database to the specific place in the codebase it is being called. In many database engines, you can add comments for information. These comments will show up in the database query logs. This is a great place to insert the codebase location.

Distributed Tracing

Tracing performance at all stages from the application to the datastore is critical for optimizing long-tail latency issues that can be hard to capture. Systems like New Relic or Zipkin (open source) allow for distributed traces from application calls to the external services such as your databases. A full transaction trace from the application to datastore should ideally give timing for all external service calls, not just the database query.

Tracing with full visibility through to the database can become a powerful arsenal in educating your SWE teams and creating autonomy and self-reliance. Rather than needing you to tell them where to focus, they are able to get the information themselves. As Aaron Morton at the Last Pickle says in his talk, "Replacing Cassandra's Tracing with Zipkin":

> Knowing in advance which tools create such positive cultural shifts is basically impossible to foretell, but I've seen it with Git and its practice of pull requests and stable master branches, and I've seen it with Grafana, Kibana, and Zipkin.

You can read more about this on The Last Pickle's blog. [1]

There are many components of an end to end call that may occur and be of interest to the DBRE. These include, but are not limited to:

- Establishing a connection to a database or a database proxy
- Queuing for a connection in a database connection pool
- Logging a metric or event to a queuing or message service
- Creating a user id from a centralized UUID service
- Selecting a shard based on a variable (such as user id)

1 http://thelastpickle.com/blog/2015/12/07/using-zipkin-for-full-stack-tracing-including-cassandra.html

- Searching, invalidating or caching at a cache tier
- Application layer compression or encryption
- Querying a search layer

Traditional SQL Analysis

Laine here. In my consulting days, I can't tell you the amount of times I'd come into a shop that had no monitoring that mapped application performance monitoring to database monitoring. I'd invariably have to do tcp or log based SQL gathering to create a view from the database. Then, I'd go back to the SWEs with my prioritized list of SQL to optimize and they'd have no idea where to go to fix that code. Searching code bases and ORM mappings could take a week or more of precious time.

As a DBRE you have an amazing opportunity to work side by side with SWEs to ensure that every class, method, function and job has direct mappings to SQL that is being called. When SWEs and DBREs use the same tools, DBREs can teach at key inflection points, and soon you'll find SWEs doing your job for you!

If a transaction has a performance "budget", and the latency requirements are known, then the staff responsible for every component are incentivized to work as a team to identify the most expensive aspects and make the appropriate investments and compromises to get there.

Events and Logs

It goes without saying that all application logs should be collected and stored. This includes stack traces! Additionally, there are numerous events that will occur that are incredibly useful to register with opviz:

- Code deployments
- Deployment time
- Deployment errors

Application monitoring is a crucial first step, providing realistic looks at behavior from the user perspective, and is directly related to latency SLOs. These are the symptoms providing clues into faults and degradations within the environment. Now, let's look at the sup-

porting data that can help with root cause analysis and provisioning: host data.

Instrumenting the Server or Instance

Next is the individual host, real or virtual, that the database instance resides on. It is here we can get all of the data regarding the operating system and physical resources devoted to running our databases. While this data is not specifically application/service related, it is valuable to use when you've seen symptoms such as latency or errors in the application tier.

When using this data to identify causes for application anomalies, the goal is to find resources that are over or under-utilized, saturated or throwing errors. (USE, as Brendan Gregg defined in his methodology. [2] This data is also crucial for capacity planning for growth, and performance optimization. Recognizing a bottleneck, or constraint, allows you to prioritize your optimization efforts to maximize value.

2 http://www.brendangregg.com/usemethod.html

NOTE

Distributed Systems Aggregation

Remember that individual host data is not especially useful, other than for indicating that a host is unhealthy and should be culled from the herd. Rather, think about your utilization, saturation and errors from an aggregate perspective for the pool of hosts performing the same function. In other words, if you have 20 Cassandra hosts, you are mostly interested in the overall utilization of the pool, the amount of waiting (saturation) that is going on, and any errors faults that are occurring. If errors are isolated to one host, then it is time to remove that one from the ring and replace it with a new host.

In a linux system, a good starting place for resources to monitoring in a linux environment includes:

- CPU
- Memory
- Network Interfaces
- Storage I/O
- Storage Capacity
- Storage Controllers
- Network Controllers
- CPU Interconnect
- Memory Interconnect
- Storage Interconnect

In addition to hardware resource monitoring, operating system software has a few items to track:

- Kernel Mutex
- User Mutex
- Task Capacity
- File Descriptors

If this is new to you, I'd suggest going to Brendan Gregg's USE page for Linux [3], because it is incredibly detailed in regards to how to monitor this data. Its obvious that a significant amount of time and effort went into the data he presents.

3 http://www.brendangregg.com/usemethod.html

Events and Logs

In addition to metrics, you should be sending all logs to an appropriate event processing system such as *RSyslog* or *Logstash*. This includes kernel, cron, authentication, mail and general messages logs, as well as process or application specific log to ingest as well, such as mysqld, or nginx.

Your configuration management and provisioning processes should also be registering critical events to your opviz stack. Here is a decent starting point:

- A host being brought into our out of service.
- Configuration changes.
- Host Restarts
- Service Restarts
- Host crashes
- Service crashes

Cloud and Virtualized Systems

There are a few extra items to consider in these environments.

Cost! You are spending money on-demand in these environments, rather than up-front spend that you might be used to in datacenter environments. Being cost effective and efficient is crucial.

When monitoring CPU, monitor steal time. This is time the virtual CPU is waiting on real CPU which is being used elsewhere. High steal times (10% or more over sustained periods) are indicators that there is a noisy neighbor in your environment! If steal time is the same across all of your hosts, this probably means that you are the culprit and you may need to add more capacity and/or rebalance.

If steal time is on one or a few hosts, that means some other tenant is stealing your time! Its best to kill that host and launch a new one. The new one will hopefully be deployed somewhere else, and will perform much better.

If you can get the above into your opviz stack, you will be in great shape for understanding what's going on at the host and operating system levels of the stack. Now, let's look at the databases themselves.

Instrumenting the Datastore

What do we monitor and track in our databases, and why? Some of this will depend on the kind of datastore. Our goal is to give goals that are generic enough to be universal, but specific enough to help you track to your own databases. We can break this down into four areas:

- Datastore Connection Layer
- Internal Database Visibility
- Database Objects
- Database Calls/Queries

Each of these will get its own section, starting with the datastore connection layer.

Datastore Connection Layer

We have discussed the importance of tracking the time it takes to connect to the backend datastore as part of the overall transaction. A tracing system should also be able to break out time talking to a proxy, and time from the proxy to the backend as well. This can also be captured via tcpdump and *Tshark/Wireshark* for ad hoc sampling if something like *Zipkin* is not available. This can be automated for occasional sampling, or run ad hoc.

If you are seeing latency and/or errors between the application and the database connection, you will require additional metrics to help identify causes. Taking the USE method we recommended above, let's see what other metrics can assist us.

Utilization

Databases can support only a finite number of connections. The maximum amount of connections is constrained in multiple locations. Database configuration parameters will tell the database to accept only a certain amount of connections, setting an artificial top boundary to minimize overwhelming the host. Tracking this maximum, as well as the actual number of connections is crucial, as it might be set arbitrarily low by a default configuration.

Connections also open resources at the operating system level. For instance, Postgres uses one unix process per connection. MySQL, Cassandra and MongoDB use a thread per connection. All of them

use memory and file descriptors. So, there are multiple places we want to look at in order to understand connection behaviors.

- Connection upper bound and connection count
- Connection states (working, sleeping, aborted etc...)
- Kernel level Open file utilization
- Kernel level max processes utilization
- Memory utilization
- Thread pool metrics, such as MySQL table cache or MongoDB thread pool utilization
- Network throughput utilization

This should tell you if you have a capacity/utilization bottleneck somewhere in the connection layer. If you are seeing 100% utilization, and saturation is also high, this is a good indicator. But, low utilization combined by saturation is also an indicator of a bottleneck somewhere. High, but not full, utilization of resources is also often quite impactful to latency and could be causing latency as well.

Saturation

Saturation is often most useful when paired with utilization. If you are seeing a lot of waits for resources that are also showing 100% utilization, you are seeing a pretty clear capacity issue. However, if you are seeing waits/saturation without full utilization, there might be a bottleneck elsewhere that is causing the stack up. Saturation can be measured at these inflection points:

- TCP connection backlog
- DB specific connection queuing, such as MySQL back_log
- Connection timeout errors
- Waiting on threads in the connection pools
- Memory swapping
- Database processes that are locked

Queue length and wait timeouts are crucial for understanding saturation. Any time you find connections or processes waiting, you have an indicator of a potential bottleneck.

Errors

With utilization and saturation, you can find out if capacity constraints and bottlenecks are impacting the latency of your database connection layer. This is great information for deciding if you need to increase resources, remove artificial configuration constraints or

make some architectural changes. Errors should also be monitored and used to help eliminate or identify faults and/or configuration problems. Errors can be captured as follows:

- Database logs will provide error codes when database level failures occur. Sometimes you have configurations with various degrees of verbosity. Make sure you have logging verbose enough to identify connection errors, but do be careful about overhead, particularly if your logs are sharing storage and IO resources with your database.
- Application and proxy logs will also provide rich sources of errors.
- Host errors discussed in the previous section should also be utilized here.

Errors will include network errors, connection timeouts, authentication errors, connection terminations and much more. These can point to issues as varied as corrupt tables, reliance on DNS, deadlocks, auth changes and much more.

By utilizing application latency/error metrics, tracing and appropriate telemetry on utilization, saturation and specific error states you should have the information you need to identify degraded and broken states at the database connection layer. Next, we will look at what to measure inside of the connections.

Troubleshooting Connection Speeds, PostgreSQL

Instagram happens to be one of the properties that chose PostgreSQL to be their relational database. They chose to use a connection pooler, PGBouncer, to increase the number of application connections that could connect to their databases. This is a proven scaling mechanism for increasing the number of connections to a datastore, and considering that PostgreSQL must spawn a new UNIX process for every connection, new connections are slow and expensive.

Using the psycopg2 python driver, they were working with the default of autocommit=FALSE. This means that even for read only queries, explicit BEGINS and COMMITS were being issued. By changing autocommit to TRUE, they reduced their query latency, which also reduced queuing for connections in the pool.

This would initially show up as increased latency in the application as the pool was 100% utilized, causing queues to increase. By looking at the connection layer metrics, and monitoring pgbouncer pools, you would see that the *waiting* pool was increasing due to saturation, and that *active* was fully utilized most of the time. With no other metrics showing significant utilization/saturation, and errors clear, it would be time to look at what is going on inside of the connection that was slowing down queries. We will look into that in the next section.

Internal Database Visibility

Once we are looking inside of the database, there is a substantial increase in the number of moving parts, number of metrics and overall complexity. In other words, this is where things start to get real! Again, let's keep in mind USE. Our goal is to understand bottlenecks that might be impacting latency, constraining requests or causing errors.

It is important to be able to look at this from an individual host perspective, and in aggregate by role. Some databases, like MySQL, PostgreSQL, ElasticSearch and MongoDB have master and replica roles. Cassandra and Riak have no specific roles, but they are often distributed by region or zone, and that too is important to aggregate by.

Throughput and Latency Metrics

How many and what kind of operations are occurring in the datastores? This data is a very good high level view of database activity. As SWEs put in new features, these workloads will shift and provide good indicators of how the workload is shifting.

- Reads
- Writes
  - Inserts
  - Updates

  - Deletes
- Other Operations
  - Commits
  - Rollbacks
  - DDL Statements
  - Other Administrative Tasks

When we discuss latency here, we are talking in the aggregate only, and thus averages. We will discuss granular and more informative query monitoring further in this section. Thus, you are getting no outliers in this kind of data, only very basic workload information.

Commits, Redo and Journaling

While the specific implementations will depend on the datastore, there are almost always a set of I/O operations involved in flushing data to disk. In MySQL's InnoDB storage engine and in PostgreSQL, writes are changed in the buffer pool (memory), and operations are recorded in a redo log (or write-ahead log in PostgreSQL). Background processes will then flush this to disk while maintaining checkpoints for recovery. In Cassandra, data is stored in a memtable (memory), while a commit log is appended to. Memtables are flushed periodically to an SSTable. SSTables are periodically compacted as well. Some metrics you might monitor for this include:

- Dirty Buffers (MySQL)
- Checkpoint Age (MySQL)
- Pending and Completed Compaction Tasks (Cassandra)
- Tracked Dirty Bytes (MongoDB)
- (Un)Modified Pages Evicted (MongoDB)
- log_checkpoints config (PostgreSQL)
- pg_stat_bgwriter view (PostgreSQL)

All checkpointing, flushing and compaction are all operations that have significant performance impacts on activity in the database. Sometimes the impact is increased I/O, and sometimes the impact can be a full stop of all write operations while a major operation occurs. Gathering metrics here allows you to tune specific configurables to minimize the impacts that will occur during such operations. So in this case, when we see latency increasing and see metrics related to flushing showing excessive background activity, we will be pointed towards tuning operations related to these processes.

Replication State

Replication is the copying of data across multiple nodes so that the data on one node is identical to another. It is a cornerstone of availability and read scaling, as well as a part of disaster recovery and data safety. There are three replication states that can occur, however, that are not healthy and can lead to big problems if they are not monitored and caught. Replication is discussed in detail in Chapter 15, Data Replication.

Replication latency is the first of the fault states. Sometimes the application of changes to other nodes can slow down. This may be because of network saturation, single threaded applies that cannot keep up, or any number of other reasons. Sometimes replication will never catch up during peak activity, causing the data to be hours old on the replicas. This is dangerous, as stale data can be served, and if you are using this replica as a failover, you can lose data.

Most database systems have easily tracked replication latency metrics, as it will generally show the difference between the timestamp on the master, and the timestamp on the replica. In systems like Cassandra, with eventually consistent models, you are looking for backlogs of operations used to bring replicas into sync after unavailability. For instance, in Cassandra, this is hinted handoffs.

Broken replication is the second of the fault states. In this case, the processes required to maintain data replication simply break due to any number of errors. Resolution requires rapid response thanks to appropriate monitoring, followed by repair of the cause of the errors, and replication allowed to resume and catch up. In this case, you can monitor the state of replication threads.

The last error state is the most insidious -- replication drift. In this case, data has silently gone out of sync, causing replication to be useless, and potentially dangerous. Identifying replication drift for large data sets can be challenging, and depends on the workloads and kind of data that you are storing.

For instance, if your data is relatively immutable and insert/read operations are the norm, you can run checksums on data ranges across replicas, then compare checksums to see if they are identical. This can be done in a rolling method behind replication, allowing for an easy safety check at the cost of extra CPU utilization on the database hosts. If you are doing a lot of mutations however, this proves more challenging as you either have to repeatedly run check-

sums on data that has already been reviewed, or just do occasional samples.

Memory Structures

Data stores will maintain numerous memory structures in their regular operation. One of the most ubiquitous in databases is a data cache. While it may have many names, the goal of this is to maintain frequently accessed data in memory, rather than from disk. Other caches like this can exist, including caches for parsed SQL, connection caches, query result caches and many more.

The typical metrics we use when monitoring these structures are:

- Utilization: The overall amount of allocated space that is in use over time.
- Churn: The frequency that cached objects are removed to make room for other objects, or because the underlying data has been invalidated.
- Hit Ratios: The frequency with which cached data is used rather than uncached data. This can help with performance optimization exercises.
- Concurrency: Often these structures have their own serialization methods, such as mutexes, that can become bottlenecks. Understanding saturation of these components can help with optimization as well.

Some systems, like Cassandra, use Java Virtual Machines (JVMs) for managing memory, and thus expose whole new areas to monitor. Garbage collection, and usage of the various object heap spaces are also critical in such environments.

Locking and Concurrency

Relational databases in particular, utilize locks to maintain concurrent access between sessions. Locking allows mutations and reads to occur while guaranteeing that nothing might be changed by other processes. While this is incredibly useful, it can lead to latency issues as processes stack up, waiting for their turn. In some cases, you can have processes timing out due to deadlocks, where there is simply no resolution for the locks that have been put in place, but to roll back. The details of locking implementations will be reviewed in Chapter 12, Datastore Attributes.

Monitoring locks includes monitoring the amount of time spent waiting on locks in the datastore. This can be considered a saturation metric, and longer queues can indicate application and concurrency issues, or underlying issues that impact latency, with sessions holding locks taking longer to complete. Monitoring rollbacks and deadlocks is also crucial, as it is another indicator that applications are not releasing locks cleanly, causing waiting sessions to timeout and rollback. Rollbacks can be part of a normal, well behaved transaction but they often are a leading indicator that some underlying action is impacting transactions.

As discussed in the memory structures section above, there are also numerous points in the database that function as synchronization primitives, designed safely manage concurrency. These are generally either mutexes, or semaphores. A mutex is locking mechanism used to synchronize access to a resource, such as a cache entry. Only one task can acquire the mutex. It means there is ownership associated with mutex, and only the owner can release the lock (mutex). This protects from corruption.

A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number. Threads can request access to the resource (decrementing the semaphore), and can signal that they have finished using the resource (incrementing the semaphore. An example of the mutexes/semaphores to monitor in MySQL's InnoDB storage engine is below:

InnoDB Semaphore Activity Metrics

Name	Description
Mutex Os Waits (Delta)	The number of InnoDB semaphore/mutex waits yielded to the OS.
Mutex Rounds (Delta)	The number of InnoDB semaphore/mutex spin rounds for the internal sync array.
Mutex Spin Waits (Delta)	The number of InnoDB semaphore/mutex spin waits for the internal sync array.
Os Reservation Count (Delta)	The number of times an InnoDB semaphore/mutex wait was added to the internal sync array.
Os Signal Count (Delta)	The number of times an InnoDB thread was signaled using the internal sync array.
Rw Excl Os Waits (Delta)	The number of exclusive (write) semaphore waits yielded to the OS by InnoDB.

Name	Description
Rw Excl Rounds (Delta)	The number of exclusive (write) semaphore spin rounds within the InnoDB sync array.
Rw Excl Spins (Delta)	The number of exclusive (write) semaphore spin waits within the InnoDB sync array.
Rw Shared Os Waits (Delta)	The number of shared (read) semaphore waits yielded to the OS by InnoDB.
RW Shared Rounds (Delta)	The number of shared (read) semaphore spin rounds within the InnoDB sync array.
RW Shared Spins (Delta)	The number of shared (read) semaphore spin waits within the InnoDB sync array.
Spins Per Wait Mutex (Delta)	The ratio of InnoDB semaphore/mutex spin rounds to mutex spin waits for the internal sync array.
Spins Per Wait RW Excl (Delta)	The ratio of InnoDB exclusive (write) semaphore/mutex spin rounds to spin waits within the internal sync array.
Spins Per Wait RW Shared (Delta)	The ratio of InnoDB shared (read) semaphore/mutex spin rounds to spin waits within the internal sync array.

Increasing values in these can indicate that your datastores are reaching concurrency limits on specific areas in the code base. This can be resolved via tuning configurables and/or by scaling out in order to maintain sustainable concurrency on a datastore to satisfy traffic requirements.

Locking and concurrency can truly kill even the most performant of queries once you start experiencing a tipping point in scale. By tracking and monitoring these metrics during load tests and in production environments, you can understand the limits of your database software, as well as identifying how your own applications must be optimized in order to scale up to large numbers of concurrent users.

Database Objects

It is crucial to understand what your database looks like and how it is stored. At the simplest level, this is an understanding of how much storage each database object and its associated keys/indexes takes. Just like filesystem storage, understanding the rate of growth and the time to reaching the upper boundary is as crucial, if not more, than the current storage usage.

In addition to understand the storage and growth, monitoring the distribution of critical data is helpful. For instance, understanding the high and low bounds, means and cardinality of data is helpful to understand index and scan performance. This is particularly important for integer datatypes, and low cardinality character based datatypes. Having this data at your SWE fingertips allows you and them to recognize optimizations on datatypes and indexing.

If you have sharded your dataset using key ranges or lists, then understanding the distribution across shards can help make sure you are maximizing output on each node. These sharding methodologies allow for hot spots, as they are not even distributions using a hash or modulus approach. Recognizing this will advise you and your team on needs to rebalance or reapproach your sharding models.

Database Queries

Depending on the database system you are working with, the actual data access and manipulation activity may prove to be highly instrumented, or not at all. Trying to drink at the firehose of data that results in logging queries in a busy system can cause critical latency and availability issues to your system and users. Still, there is no more valuable data than this. Some solutions, such as Vivid Cortex and Circonus, have focused on TCP and wire protocols for getting the data they need, which dramatically reduces performance impact of query logging. Other methods include sampling on a less loaded replica, only turning logging on for fixed periods of time, or only logging statements that execute slowly.

Regardless of the above, you want to store as much as possible about the performance and utilization of your database activity. This will include the consumption of CPU and IO, number of rows read or written, detailed execution times and wait times and execution counts. Understanding optimizer paths, indexes used and statistics around joining, sorting and aggregating is also critical for optimization.

Database Asserts and Events

Database and client logs are a rich source of information, particularly for asserts and errors. These logs can give you crucial data that can't be monitored any other way.

- Connection attempts and failures
- Corruption warnings and errors
- Database restarts
- Configuration changes
- Deadlocks
- Core dumps and stack traces

Some of this data can be aggregated and pushed to your metrics systems. Others should be treated as events, to be tracked and used for correlations.

Wrapping Up

Well, after all of that, I think we all need a break! Hopefully you've come out of this chapter with a solid understanding of the importance of operational visibility, how to start an opviz program, and how to build and evolve an opviz architecture. You can never have enough information about the systems you are building and running. You can also quickly find the systems built to observe the services become a large part of your operational responsibilities! They deserve just as much attention as every other component of the infrastructure.