January 2017 COMPUTER SCIENCE APPLIED Issue # 3

THE MORNING PAPER
QUARTERLY REVIEW

GOODS: ORGANIZING GOOGLE'S DATASETS

FLEXIBLE PAXOS: QUORUM INTERSECTION REVISITED

ON DESIGNING AND DEPLOYING INTERNET SCALE SERVICES

A selection of papers from the world of computer science,
as featured by Adrian Colyer on The Morning Paper blog

InfoQ

CONTENTS

Issue # 2, July 2016

DBSHERLOCK: A PERFORMANCE DIAGNOSTIC TOOL FOR TRANSACTIONAL DATABASES 05

GOODS: ORGANIZING GOOGLE'S DATASETS 11

FLEXIBLE PAXOS: QUORUM INTERSECTION REVISITED 16

ON DESIGNING AND DEPLOYING INTERNET SCALE SERVICES 20

ON THE CRITERIA TO BE USED IN DECOMPOSING SYSTEMS INTO MODULES 26

WELCOME...

It should have been easier to choose the five papers for this edition, on account of the fact that I paused in reviewing papers for a month in the summer and thus had fewer write-ups to choose from. But it was just as hard as ever: there's so much wonderful research out there! With apologies to those I had to leave out, here are my five choices for this edition, in the order in which the posts first appeared on The Morning Paper.

Adrian Colyer

DBSHERLOCK: A PERFORMANCE DIAGNOSTIC TOOL FOR TRANSACTIONAL DATABASES

Yoon et al., SIGMOD 2016.

What I love about this paper is that it introduces a general framework for producing human-understandable explanations of performance anomalies. Lots of anomaly-detection systems will tell you when something doesn't look quite right. DBSherlock can help you figure out why the system exhibited poor performance during that time window. At a basic level, it can show you the set of predicates with the most explanatory power for the observed difference. On top of that, it can learn to provide higher-level explanations such as "the database was slower than normal during this time period due to a rotation of the redo log file." Yoon et al. happen to apply their method in the context of DBMS performance but it has much broader applicability. As an added bonus, DBSherlock is also released as open source as part of the DBSeer database administration tool.

GOODS: ORGANIZING GOOGLE'S DATASETS

Havely et al., SIGMOD 2016.

Where is all the data within your enterprise? I've seen lots of well-intentioned centralised data-management schemes fall by the wayside in the heat of business. Google take a different approach: centralised control doesn't really seem to fit within the Google culture. And then, of course, there's the sheer scale of the problem - we're talking about approximately 26 billion datasets (I had to go back to the original paper and double-check that I got that right!). So what does Google do? It does what Google does best, of course, and crawl and index them! Goods has become an indispensable tool within Google, and a number of other systems are built on top of it. It's a great reminder that data dependencies and provenance are just as important as managing the dependencies between applications and services.

FLEXIBLE PAXOS: QUORUM INTERSECTION REVISITED

Howard et al., 2016.

Consensus algorithms are a hard-core topic within distributed systems, much studied and notoriously difficult to get right (think Paxos, Raft). One of the foundational building blocks is the notion of a majority quorum: if, at a given stage, a majority of the system members agree on a decision, then we will always be able to rely on at least one of those members being present in the majority quorum of a subsequent stage in order to pass that decision on. Howard et al. show that the requirement for a majority quorum in each phase of Paxos is conservative, and by relaxing that requirement, we can build consensus systems with lower latency, higher throughput, and more participants. In an area that's been so well studied, it's a real eye-opener to see something so significant still up for grabs!

ON DESIGNING AND DEPLOYING INTERNET SCALE SERVICES

Hamilton, 2007.

When Netflix first looked for principles on which to build its cloud architecture, this is one of the key papers it turned to. The paper is absolutely packed with advice for building Internet-scale systems (and cloud-native applications), with three major themes running through it: expect failures, keep things simple, and automate everything. What's equally amazing is that all of this knowledge was captured nine years ago! A true goldmine.

ON THE CRITERIA TO BE USED IN DECOMPOSING SYSTEMS INTO MODULES

Parnas, 1971.

A lot of attention has recently been given to the idea of splitting a system into a number of smaller modules (microservices). But much less has been said about the criteria by which we decide how to divide the system up. This classic paper is a timely reminder that it's not simply about having lots of small modules: a large part of the success or otherwise of your system depends on how you undertake the partitioning. If you're hoping that adopting microservices will help your development team move faster, it's well worth a read.

DBSHERLOCK:
A PERFORMANCE DIAGNOSTIC TOOL FOR TRANSACTIONAL DATABASES

What I love about this paper is that it introduces a general framework for producing human-understandable explanations of performance anomalies.

Yoon et al. 2016

...Tens of thousands of concurrent transactions competing for the same resources (e.g. CPU, disk I/O, memory) can create highly non-linear and counter-intuitive effects on database performance. (All quotes from Yoon et al. 2016.)

If you're a DBA responsible for figuring out what's going on, this presents quite a challenge. You might be awash in stats and graphs (MySQL maintains 260 statistics and variables, for example), but still sorely lack the big picture. Yoon et al. sum it up in "DBSherlock: A performance diagnostic tool for transactional databases" (*SIGMOD 2016*):

As a consequence, highly-skilled and highly-paid DBAs (a scarce resource themselves) spend many hours diagnosing performance problems through different conjectures and manually inspecting various queries and log files, until the root cause is found.

DBSherlock (available at http://dbseer.org) is a *performance-explanation framework* that helps DBAs diagnose performance problems in a more principled manner. The core of the idea is to compare a region of interest (an anomaly) with normal behaviour by analysing past statistics to try and find the most likely causes of the anomaly. The result of this analysis is one or both of the following:

- A set of concise predicates describes the combination of system configurations or workload characteristics causing the performance anomaly. For example, to explain an anomaly caused by a network slowdown:

DBSherlock is a performance-explanation framework that helps DBAs diagnose performance problems in a more principled manner. The core of the idea is to compare a region of interest (an anomaly) with normal behaviour by analysing past statistics to try and find the most likely causes of the anomaly.

```
network_send < 10KB
? network_recv < 10KB
? client_wait_times >
100ms
? cpu_usage < 5
```

- A high-level diagnosis is based on existing causal models in the system. A sample cause presented here might be "Rotation of the redo log file."

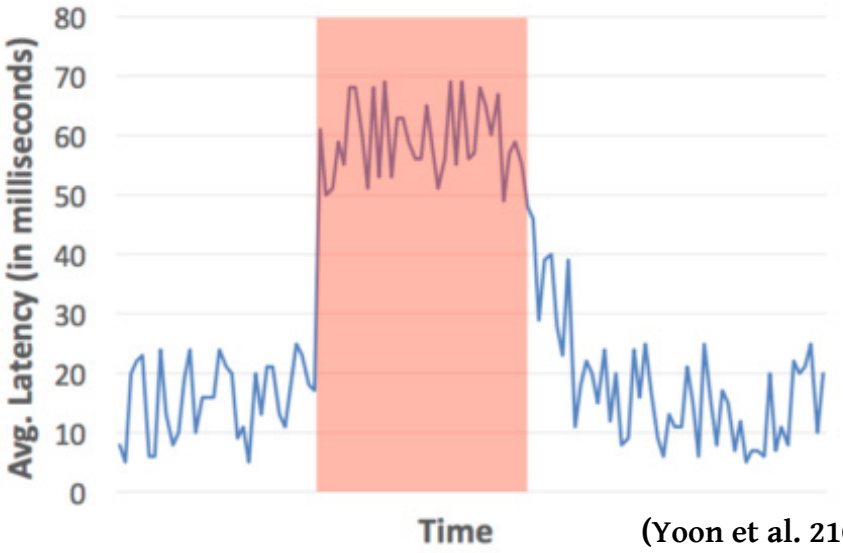
Suppose that in the first of these two cases, the predicates are presented to the user, who diagnoses the root cause as a network slowdown based on these hints. This information is fed back to DBSherlock by accepting the predicates and labelling them with the actual cause.

This "cause" and its corresponding predicates comprise a causal model, which will be utilized by Sherlock for future diagnoses.

There are a number of similarities here to other work on time-series discretization, anomaly detection, and correlation, but Yoon et al. emphasise *explaining* anomalies (largely assumed to be already detected by the user because they're of the obvious "performance is terrible" kind, though automated anomaly detection can be integrated with the system) rather than *detecting* them.

The starting point is a graph like the following, where the user can highlight a region and ask what's going on.

There are three main layers to answering that question inside DBSherlock: firstly, predicate generation seeks to find a small number of predicates that have high separation power to divide the anomalous and normal region behaviours; secondly, the set of predicates can be further pruned using simple rules that encode limited domain knowledge; and finally the causal model layer seeks to learn and map sets of predicates to higher-level causes, capturing the knowledge and experience of the DBA over time.



(Yoon et al. 2106)

DBSherlock is integrated as a module in DBSeer, an open-source suite of database administration tools for monitoring and predicting database performance. This means DBSherlock can rely on DBSeer's API for collecting and visualising performance statistics. It collects the following data at one-second intervals:

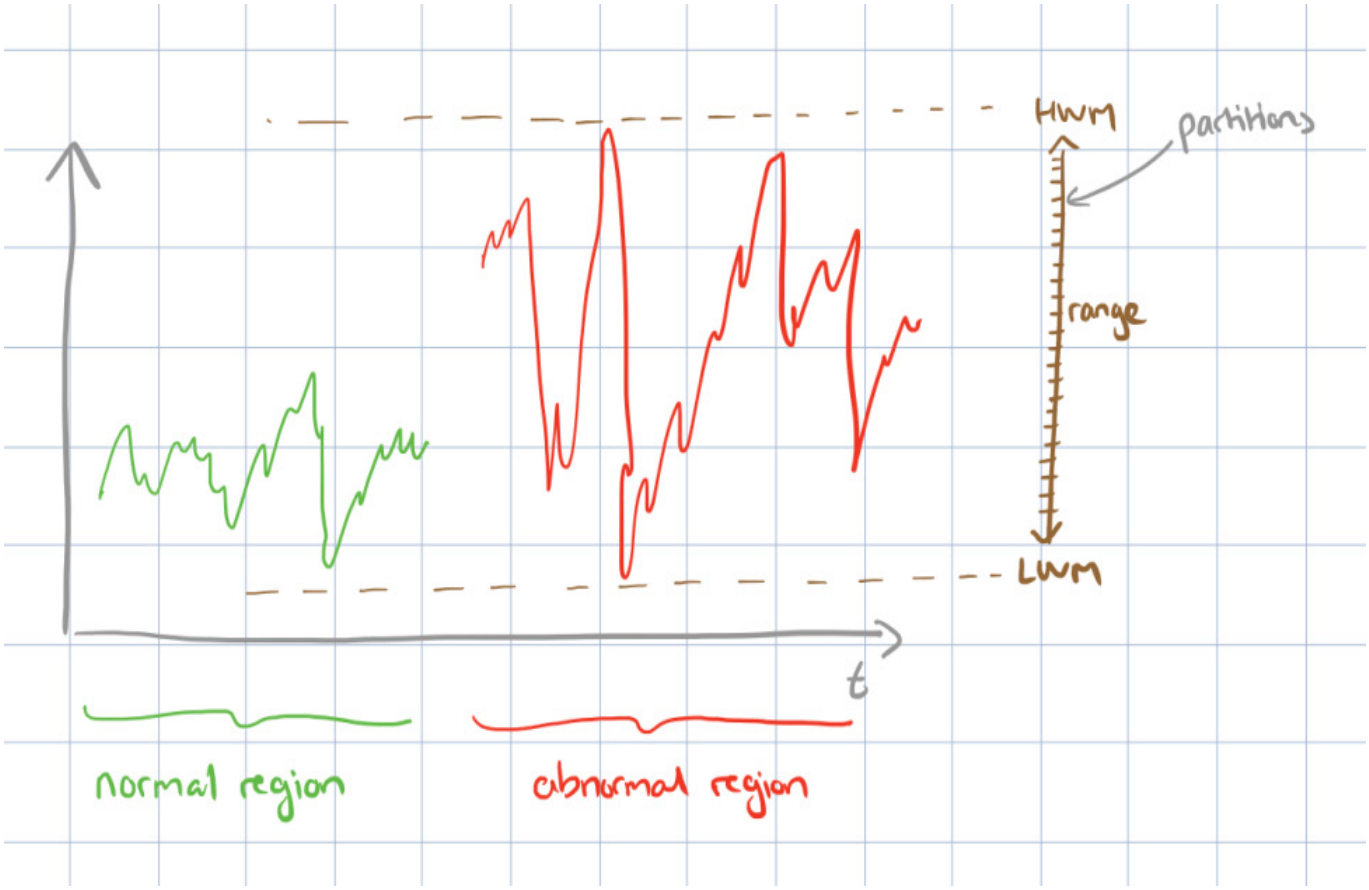
- OS resource-consumption statistics,
- DBMS workload statistics,
- time-stamped query logs, and
- configuration parameters from the OS and DBMS.

From the raw data, DBSeer computes aggregate statistics about transactions executed during each time interval, and aligns them with the OS and DBMS statistics according to their timestamps. It is these resulting time series that DBSherlock uses as the starting point for diagnosis.

FINDING PREDICATES OF INTEREST

Starting with a given abnormal region and a normal region, DBSherlock aims to find predicates that can separate the two regions as cleanly as possible. The *separation power* of a predicate is defined as the difference between percentage of abnormal tuples that satisfy it and the percentage of normal tuples that satisfy it.

Identifying predicates with high separation power is challenging. First, one cannot find a predicate of high separation power by simply comparing the values of an attribute in the raw dataset. This is

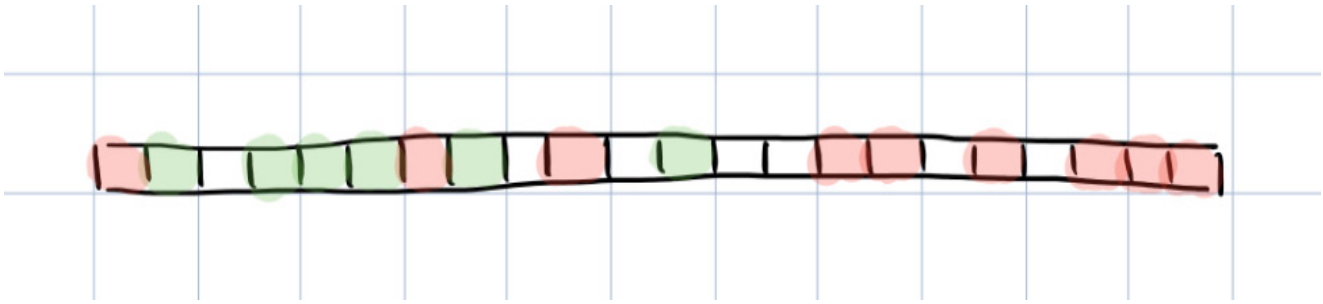


because real-world datasets and OS logs are noisy and attribute values often fluctuate regardless of the anomaly. Second, due to human error, users may not specify the boundaries of the abnormal regions with perfect precision. The user may also overlook smaller areas of anomaly, misleading DBSherlock to treat them as normal regions. These sources of error compound the problem of noisy datasets. Third, one cannot easily conclude that predicates with high separation power are the actual cause of an anomaly. They may simply be correlated with, or be symptoms themselves of the anomaly, and hence, lead to incorrect diagnoses.

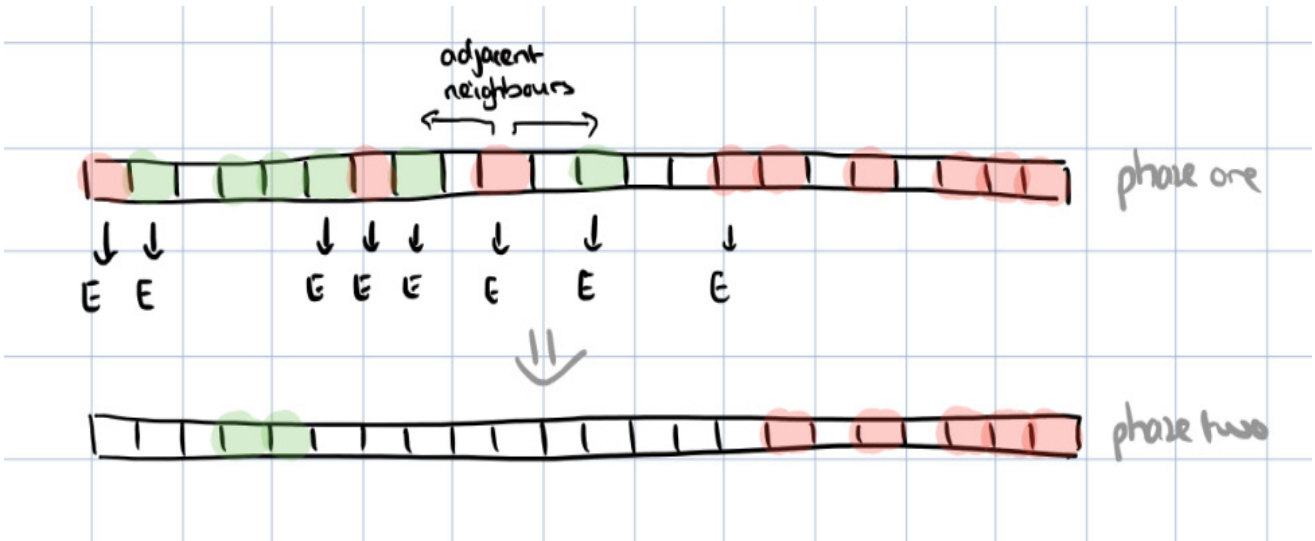
The first step is to discretize the time series (or each attribute within a multidimensional time series, if you prefer to think of it that way) within the ranges of interest. For numeric attributes, the maximum and minimum values within the ranges of interest are found, and the resulting range is divided into *R* equal-sized buckets or *partitions*. For example, with a range of 0 to 100 and R of 5, these would be [0,20), [20,40), [40,60), [60,80), and [80,100). By default, DBSherlock uses R of 1,000. For categorical attributes, it creates one partition for each unique attribute value. The result of this step is a *partition space*. (*Why not use a standard time-series algorithm such* as SAX for *this step?*)

The next step is to *label* the partitions. DBSherlock examines all the tuples that fall within each partition. If all of these tuples are from the abnormal region, the partition is labelled "abnormal". If all of the tuples are instead from the normal region, then the partition is labelled "normal". Otherwise, DBSherlock leaves a partition empty (unlabelled).

At this stage, due to the noise in the data, there will probably be a mixed set of abnormal, normal, and empty partitions such as this:



To cope with this noise, a *filtering* and *filling* process now takes place. Filtering helps to find a sharper edge separating normal and abnormal. This is a two-phase process. In the first phase, DBSherlock examines each non-empty (i.e. abnormal or normal) partition. If either neighbouring non-empty partition is not of the same type (e.g. an abnormal partition has a normal partition as a non-empty neighbour) then that partition is marked. In the second phase, all marked partitions are switched to empty.



Our filtering strategy aims to separate the Normal and Abnormal partitions that are originally mixed across the partition space (e.g., due to the noise in the data or user errors). If there is a predicate on attribute Attr, that has high separation power, the Normal and Abnormal partitions are very likely to form well-separated clusters after the filtering step. This step mitigates some of the negative effects of noisy data or user's error, which could otherwise exclude a predicate with high separation power from the output.

After filtering, we will be left with larger blocks of consecutive normal and abnormal partitions, separated by empty partitions. DBSherlock now fills the empty partitions by marking them as either normal or abnormal according to these criteria:

- If the nearest adjacent partitions to an empty partition both have the same type (e.g. abnormal), then the partition is assigned that type.

- If the nearest adjacent partitions have different types, then the partition is assigned the label of the closer partition. The distance calculation includes an *anomaly-distance* factor δ by which the distance to an *abnormal* adjacent neighbour is multiplied before doing the comparison. If δ is greater than 1, this favours marking a partition as normal. DBSherlock uses δ of 10 by default.

Finally, it comes time to extract a candidate predicate for the attribute. A candidate predicate is generated for a numeric attribute if the following two conditions hold:

1. There is a single block of abnormal partitions.

2. After normalizing the attribute values so that they all fall on a spectrum from 0..1, the average value of the attribute in the normal range and the average value of the attribute in the abnormal range are separated by at least θ, where θ is a configurable *normalized difference threshold*.

If a numeric attribute passes these two tests, a predicate with one of the following forms will be generated:

- `attr < x`
- `attr > x`
- `x < attr < y`

Categorical attribute predicates are simpler, and DBSherlock simply generates the predicate *Attr ∈ {p | p.label = abnormal}*.

INCORPORATING DOMAIN KNOWLEDGE

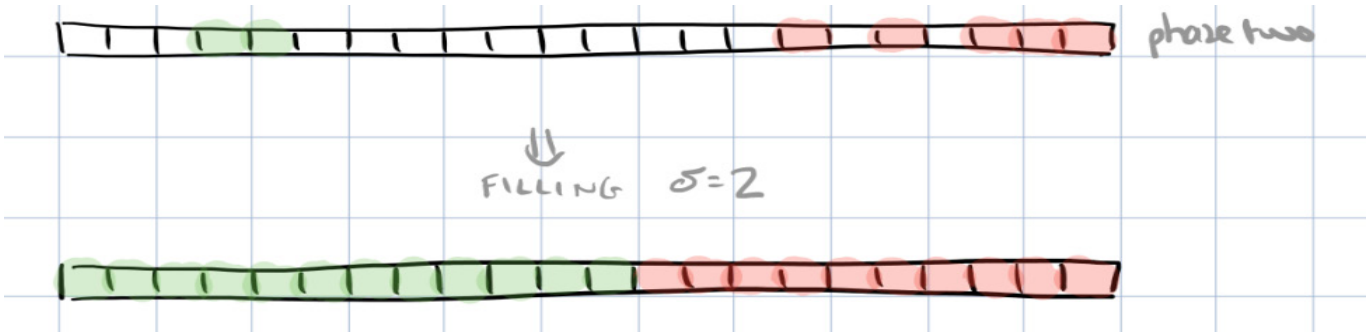
Our algorithm extracts predicates that have a high diagnostic power. However, some of these predicates may be secondary symptoms of the root cause, which if removed can make the diagnosis even easier.

The secondary symptom predicates can be pruned with a small amount of domain knowledge. This knowledge is expressed as a set of rules of the form *Attri → Attrj*, meaning that if predicates for both *Attri* and *Attrj* are generated then the predicate for *Attrj* is likely to be a secondary symptom. Cycles are not allowed in the rule set.

As an example, for MySQL on Linux, there are just four rules:

- DBMS CPU usage → OS CPU usage (If DBMS CPU usage is high, we probably don't need to also flag OS CPU usage.)

- OS allocated pages → OS free pages; OS used swap space →



OS free swap space; OS CPU usage → OS CPU idle (For these three, one attribute is always completely determined by the other and so is uninteresting.)

CAUSAL MODELS

Previous work on performance explanation has only focused on generating explanations in the form of predicates. DBSherlock improves on this functionality by generating substantially more accurate predicates (20-55% higher F1-measure;). However, a primary objective of DBSherlock is to go beyond raw predicates, and offer explanations that are more human-readable and descriptive.

Suppose DBSherlock finds three predicates with high predictive power for an anomaly, and after investigation the user confirms with the help of these clues that the true cause was rotation of the redo log file. "Rotation of the redo log file" is then added to the system as a causal model and linked with these three predicates.

After that, for every diagnosis inquiry in the future, DBSherlock calculates the confidence of this causal model and "Log Rotation" is reported as a possible cause if its confidence is higher than a threshold λ. By default, DBSherlock displays only those causes whose confidence is higher than λ=20%. However, the user can modify λ as a knob (i.e., using a sliding bar) to interactively view fewer or more causes.

When multiple causal models are created for the same root cause while analysing different anomalies over time, DBSherlock can merge these into one by keeping only the predicates for attributes that appear in both models, and by combining the boundary conditions (or categories) on attributed-matched predicates. In evaluation, it turns out that merged causal models are on average 30% more accurate than the original models.

DBSHERLOCK IN ACTION

Section 8 of the paper contains an evaluation of DBSherlock in action, looking for these 10 common types of database performance anomalies: (Table 1)

For details, I refer you to the full paper. The short version is: (see diagram below)

Our extensive experiments show that our algorithm is highly effective in identifying the correct explanations and is more accurate than the state-of-the art algorithm. As a much-needed tool for coping with the increasing complexity of today's DBMS, DBSherlock is released as an open-source module in our workload management toolkit.

Note that the principles used to build DBSherlock are not tied to the domain of database performance explanation in any deep way, so it should be entirely possible to take these ideas and apply them in other contexts — for example, "why has the latency just shot up on requests to this microservice?"

Type of anomaly	Description
Poorly Written Query	Execute a poorly written JOIN query, which would run efficiently if written properly.
Poor Physical Design	Create an unnecessary index on tables where mostly INSERT statements are executed.
Workload Spike	Greatly increase the rate of transactions and the number of clients simulated by OLTPBenchmark (128 additional terminals with transaction rate of 50,000).
I/O Saturation	Invoke stress-ng, which spawns multiple processes that spin on write()/unlink()/sync() system calls.
Database Backup	Run *mysqldump* on the TPC-C database instance to dump the table to the client machine over the network.
Table Restore	Dump the pre-dumped *history* table back into the database instance.
CPU Saturation	Invoke stress-ng, which spawns multiple processes calling poll() system calls to stress CPU resources.
Flush Log/Table	Flush all tables and logs by invoking *mysqladmin* commands ('flush-logs' and 'refresh').
Network Congestion	Simulate network congestion by adding an artificial 300-milliseconds delay to every traffic over the network via Linux's *tc* (Traffic Control) command.
Lock Contention	Change the transaction mix to execute *NewOrder* transactions only on a single warehouse and district.

Table 1: Ten types of performance anomalies used in our experiments.

(Yoon et al. 2106)

GOODS: ORGANIZING GOOGLE'S DATASETS

You can (try to) build a data cathedral. Or you can build a data bazaar. By "data cathedral", I'm referring to a centralised enterprise data-management (EDM) solution that everyone in the company buys into and pays homage to, making a pilgrimage to the EDM every time they want to publish or retrieve a dataset.

Havely et al. 2016

A data bazaar such as Halevy et al. describe in "Goods: organizing Google's datasets" (SIGMOD 2016), on the other hand, abandons premeditated centralised control:

> ***An alternative approach is to enable complete freedom within the enterprise to access and generate datasets and to solve the problem of finding the right data in a post-hoc manner... In this paper, we describe Google Dataset Search (Goods), such a post-hoc system that we built in order to organize the datasets that are generated and used within Google. (All quotes from Halevy et al. 2016.)***

Everyone within Google carries on creating and consuming datasets using whatever means they prefer, and Goods works in the background to figure out what datasets exist and to gather metadata about them. In other words, (surprise!) Google built a crawling engine.... It turns out that this is far from an easy problem to work on (to start with, the current catalogue indexes over *26 billion datasets* — and this counts only those whose access permissions make them readable by *all* Google engineers). As an approach to the problem of understanding what data exists, where it is, and its provenance, I find Goods hugely appealing versus the reality of the ever-losing battle for strict central control within an enterprise.

Let's first examine the benefits of building a data bazaar, then dive into some of the details of how you go about it.

WHY BUILD A DATA BAZAAR?

Here's the big picture of the Goods system:

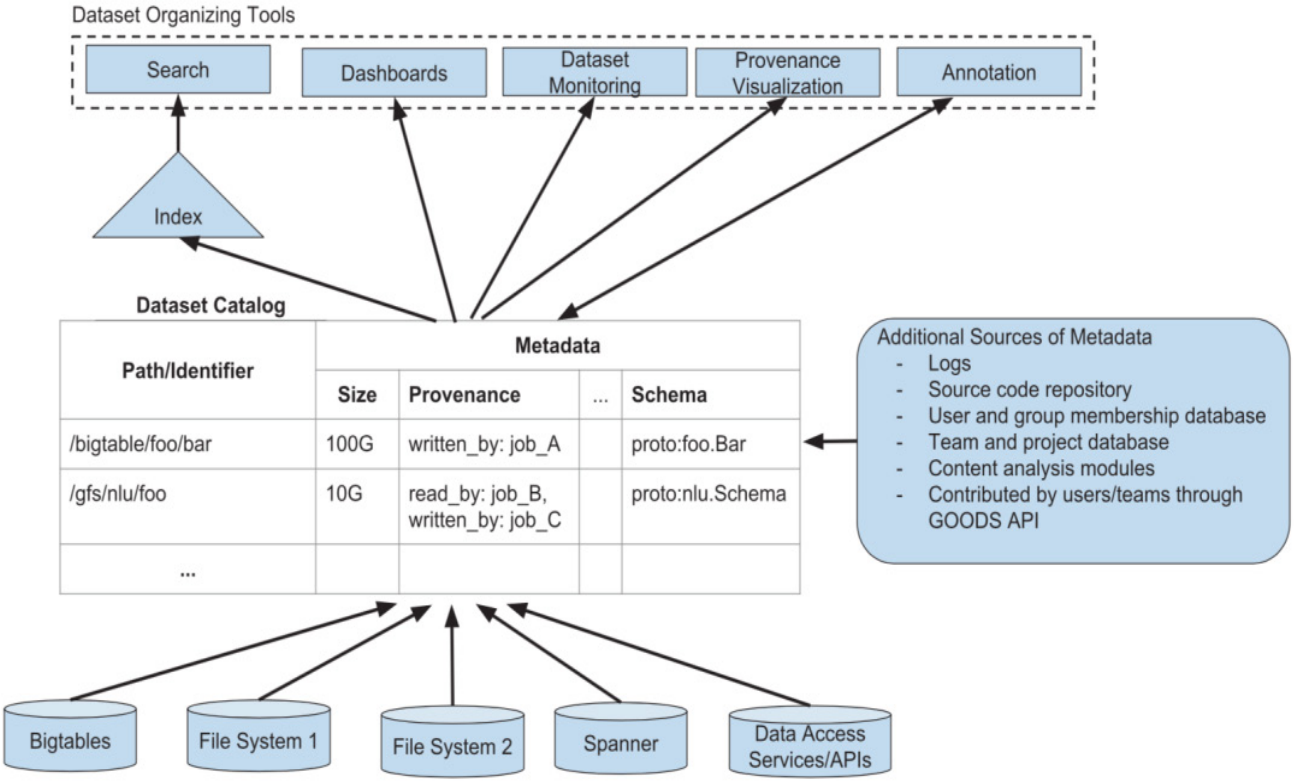


Figure 1: Overview of Google Dataset Search (Goods). The figure shows the Goods dataset catalog that collects metadata about datasets from various storage systems as well as other sources. We also infer metadata by processing additional sources such as logs and information about dataset owners and their projects, by analyzing content of the datasets, and by collecting input from the Goods users. We use the information in the catalog to build tools for search, monitoring, and visualizing flow of data.

(Halevy et al. 2016)

Goods crawls datasets from all over Google, extracts as much metadata as possible from them, joins this with metadata inferred from other sources (e.g., logs, source code, and so on) and makes this catalogue available to all of Google's engineers. Goods quickly became indispensable. Here's an example of a team working on natural language understanding (NLU):

Goods uses this catalog to provide Google engineers with services for dataset management. To illustrate the types of services powered by Goods, imagine a team that is responsible for developing natural language understanding (NLU) of text corpora (say, news articles). The engineers on the team may be distributed across the globe and they maintain several pipelines that add annotations to different text corpora. Each pipeline can have multiple stages that add annotations based on various techniques including phrase chunking, part-of-speech tagging, and co-reference resolution. Other teams can consume the datasets that the NLU team generates, and the NLU team's

pipelines may consume datasets from other teams. Based on the information in its catalog, Goods provides a dashboard for the NLU team (in this case, dataset producers), which displays all their datasets and enables browsing them by facets (e.g., owner, data center, schema). Even if the team's datasets are in diverse storage systems, the engineers get a unified view of all their datasets and dependencies among them. Goods can monitor features of the dataset, such as its size, distribution of values in its contents, or its availability, and then alert the owners if the features change unexpectedly.

Goods also tracks dataset provenance, figuring out which datasets were used in the creation of a given dataset, and which datasets consume it downstream. The provenance visualisation is very useful when figuring out what upstream changes may be responsible for a problem or for working out the potential consequences of a change under consideration. This reminds me very much of the infrastructure Google put in place for tracking "input signals" (streams and datasets) in their automated feature-management system for machine learning.

The full set of metadata that Goods tracks is illustrated below.

For dataset consumers, Goods provides a search mechanism for finding important or relevant datasets. Every dataset has its own profile page that helps users understand its schema, lists users and provenance, and indicates other datasets that contain similar content. The Goods system allows a user to provide their own data annotations, which are also indexed. This mechanism has facilitated further applications built on top of Goods.

The profile page of a dataset cross-links some of the metadata with other, more specialized, tools. For example, the profile page links the provenance metadata, such as jobs that generated the dataset, to the pages with details for those jobs in job-centric tools. Similarly, we link schema metadata to code-management tools, which provide the definition of this schema. Correspondingly, these tools link back to Goods to help users get more information about datasets. The profile page also provides access snippets in different languages (e.g., C++, Java, SQL) to access the contents of the dataset. We custom-tailor the generated snippets for the specific dataset: For example, the snippets use the path and schema of the dataset (when known), and users can copy-paste the snippets in their respective programming environment.

The Goods profile page for a dataset has become a natural handle for bookmarking and sharing dataset information. The Google File System browser provides direct links to Goods pages for datasets within a directory for example.

HOW GOOGLE BUILT GOODS

The challenge of creating a central data repository in a post hoc manner without relying on any collaboration from engineers is that you need to piece together the overall puzzle from multiple sources of information — and even then, you may never be *certain*. At Google scale, the sheer number of datasets in question (26 billion) means that you also need to be a little bit savvy about how often you crawl and process datasets.

Even if we spend one second per dataset (and many of the datasets are too large to process in one second per dataset), going through a catalog with 26 billion datasets using a thousand parallel ma-

Metadata Groups	Metadata
Basic	size, format, aliases, last modified time, access control lists
Content-based	schema, number of records, data fingerprint, key field, frequent tokens, similar datasets
Provenance	reading jobs, writing jobs, downstream datasets, upstream datasets
User-supplied	description, annotations
Team and Project	project description, owner team name
Temporal	change history

Table 2: Metadata in the Goods catalog.

(Halevy et al. 2016)

chines still requires around 300 days....

Moreover, datasets are created and deleted all the time — about 5% (about 1 billion) datasets in the catalogue are deleted every day. Two tactics are used to help deal with the volume of datasets: a twin-track approach to crawling and processing and dataset clustering.

The twin-track approach designates certain datasets as "important" — those that have a high provenance centrality and those where users have taken the effort to provide their own additional metadata annotations. One instance of the *Schema Analyzer* (the most heavyweight job in the pipeline) runs daily over these important datasets and can get through them quickly. A second instance processes all datasets, but may only get through a fraction of them within any given day.

In practice, as with Web crawling, ensuring good coverage and freshness for the "head" of the importance distribution is enough for most user scenarios.

Clustering datasets helps to make both the cognitive overload for users lighter, as well as to reduce processing costs. Consider a dataset that is produced every day, and saved to e.g. `/dataset/2015-10-10/daily_scan`. By abstracting out the date portion, it is possible to get a generic representation of the `daily_scan` file. This can be shown as a single top-level entity in the Goods, and also saves processing time on the assumption, for example, that all files in the series share the same schema.

By composing hierarchies along different dimensions, we can construct a granularity semi-lattice structure where each node corresponds to a different granularity of viewing the datasets.... Table 3 (below) lists the abstract dimensions that we currently use.

Goods contains an entry for each top-most element in the semi-lattice. This avoids having too many clusters and helps to keep the set of clusters stable over time. Some clusters can be very large!

Goods uses a variety of techniques to try and infer the metadata for a dataset.

Because Goods explicitly identifies and analyzes datasets in a post-hoc and non-invasive manner, it is often impossible to determine all types of metadata with complete certainty. For instance, many datasets consist of records whose schema conforms to a specific protocol buffer... Goods tries to uncover this implicit association through several signers: For instance, we "match" the dataset contents against all registered types of protocol buffers within Google, or we consult usage logs that may have recorded the actual protocol buffer.

The provenance metadata is especially interesting. It helps us to understand how data flows through the company and across the boundaries of internal teams and organisations. Provenance is mined from production logs, which contain information on which jobs read and write each dataset. To make this tractable, Goods only samples the logs and computes the downstream and upstream relationships over a few hops as opposed to the full transitive closure.

To find the schema associated with a dataset, Goods needs to find the protocol buffer used to read and write its records. Since these are nearly always checked into Google's source-code repos-

Abstraction Dimension	Description	Examples of paths with instances
Timestamps	All specifications of dates and times	/gfs/generated_at_**20150505T20:21:56**
Data-center Names	Specification of data center names	/gfs/**oregon**/dataset
Machine Names	Hostnames of machines (either user's or one in the data center)	/gfs/dataset/**foo.corp.google.com**
Version	Numeric and hexa-numeric version specifications	/gfs/dataset/**0x12ab12c**/bar
Universally Unique Identifier	UUIDs as specified in RFC4122[6]	/gfs/dataset/**30201010-5041-7061-9081-F0E0D0C0B0AA**/foo

Table 3: The dimensions that Goods uses to abstract dataset paths. The examples illustrate portions of the paths that correspond to the abstraction dimensions.

(Halevy et al. 2016)

itory, Goods also crawls the code repository to discover protocol buffers. It is then possible to produce a short list of protocol buffers that *could* be a match.

We perform this matching by scanning a few records from the file, and going through each protocol message definition to determine whether it could conceivable have generated the bytes we see in those records.... The matching procedure is speculative and can produce multiple candidate protocol buffers. All the candidate protocol buffers, along with heuristic scores for each candidate, become part of the metadata.

To facilitate searching for datasets, Goods also collects metadata summarizing the content of a dataset.

We record frequent tokens that we find by sampling the content. We analyze some of the fields to determine if they contain keys for the data, individually or in combination. To find potential keys, we use the HyperLogLog algorithm to estimate cardinality of values in individual fields and combinations of fields and we compare this cardinality with the number of records to find potential keys. We also collect fingerprints that have checksums for the individual fields and locality-sensitive hash (LSH) values for the content. We use these fingerprints to find datasets with content that is similar or identical to the given dataset, or columns from other datasets that are similar or identical to columns in the current dataset. We also use the checksums to identify which fields are populated in the records of the dataset.

There are many other implementation details in the paper; I wanted to focus this write-up only on the big idea as I find it compelling. One of the significant challenges still open, according to the authors, is improving the criteria for ranking datasets and identifying *important* datasets.

We know from the users' feedback that we must improve ranking significantly.... We need to be able to distinguish between production and test or development datasets, between datasets that provide input for many other datasets, datasets that users care about, and so on....

Finally, we hope that systems such as Goods will provide an impetus to instilling a "data culture" at data-driven companies today in general, and at Google in particular. As we develop systems that enable enterprises to treat datasets as core assets that they are, through dashboards, monitoring, and so on, it will hopefully become as natural to have as much "data discipline" as we have "code discipline."

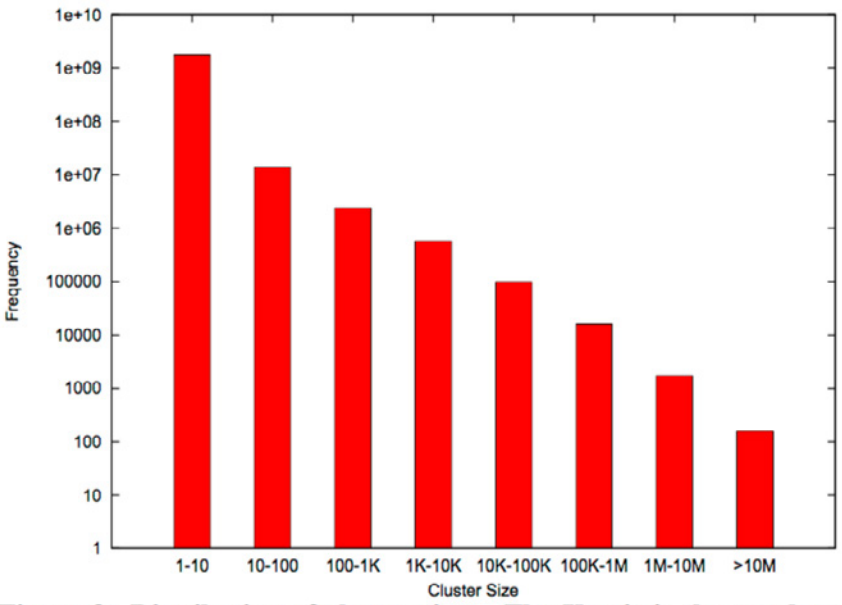


Figure 3: Distribution of cluster sizes. The X axis is the number of datasets in a cluster (cluster size). The Y axis is the number of clusters of the corresponding cluster size.

FLEXIBLE PAXOS: QUORUM INTERSECTION REVISITED

Consensus algorithms are a hard-core topic within distributed systems, much studied and notoriously difficult to get right (think Paxos, Raft).

Howard et al., 2016.

The Paxos algorithm has been around for 18 (or 26, depending on how we count) years now, and has been *extensively* studied. (For some background, see the 2 two-week mini-series on consensus that I put together last year.) In "Flexible Paxos: Quorum intersection revisited", Howard et al. (2016) make a simple(?) observation that has significant consequences for improving the fault-tolerance and throughput of Paxos-based systems. After so much time and so much study, you can't help but admire that.

Paxos proceeds in two phases: in the first phase, a quorum chooses a proposer known as the *leader*; in the second phase, the leader proposes a value to be agreed upon by the participants (*acceptors*) and brings a quorum to agreement. Paxos, like many other consensus systems, uses *majority agreement* for its quorums — i.e., at least (n/2)+1 out of n acceptors — which guarantees an intersection between the participants in phase one and phase two. Since we normally want to forge agreement over a series of values (referred to in the literature as "*slots*"), we need to run a distinct instance of Paxos each time. *Multi-Paxos* recognises that the first phase (leader election) is independent of the proposal phase, so why not perform that once and then *retain* that leader through a whole series of phase two agreements?

Here's the key new insight: Paxos is more conservative than necessary in requiring a quorum majority in each phase. *All we actually need to guarantee is that the phase one and phase two quorums intersect.* We don't need to guarantee that phase one quorums across successive rounds intersect, nor that phase two quorums intersect. If Q1 represents the participants in the first-phase quorum, and Q2 represents the participants in the second-phase quorum, then **|Q1|+|Q2|>n** (for n overall participants) will provide the necessary guarantee. And that gets really interesting when you consider that (hopefully) leader elections happen much less frequently than phase-two slot decisions. So you can trade off: for example, requiring a larger quorum when you *do* need to elect a new leader while having a smaller quorum requirement during regular operations. The authors call this variant "*Flexible Paxos*" (FPaxos). If you're like me, you'll want to see some kind of proof or justification for the quorum intersection claim — I spent the first eight pages of the paper hungry for it! It's in section five, starting on page nine. There's also a model-checked formal specification in TLA+ in the appendix.

In this paper, we demonstrate that Paxos, which lies at the foundation of many production systems, is conservative. Specifically, we observe that each of the phases of Paxos may use non-intersecting quorums. Majority quorums are not necessary as intersection is required only across phases.

Paxos is more conservative than necessary in requiring a quorum majority in each phase. All we actually need to guarantee is that the phase one and phase two quorums intersect.

(All quotes from Howard et al. 2016.)

It has previously been noted that Paxos could be generalized to replace majority quorums with any quorum system that guarantees that any two quorums will have a non-empty intersection. FPaxos takes this generalisation one step further by relaxing the conditions under which such guarantees are needed in the first place.

Howard et al. (to the best of their knowledge) are the first to prove and implement the generalization. While they prepared their publication, Sougoumarane independently made the same observation and released a blog post that summarises it in August 2016.

Why does all this matter?

The fundamental theorem of quorum intersection states that [the resilience of a quorum system] is inversely proportional to the load on (hence the throughput of) the participants. Therefore, with Paxos and its intersecting quorums, one can only hope to increase throughput by reducing the resilience, or vice versa.... by weakening the quorum intersection requirement, we can break away from the inherent trade off between resilience and performance.

Howard et al. go on to illustrate the practical implications of the relaxed quorum requirement for FPaxos using majority quorums, simple quorums, and grid quorums. These however can be considered "naive" quorum systems, and FPaxos actually opens up a whole new world.

There already exists an extensive literature on quorum systems from the fields of databases and data replication, which can now be more efficiently applied to the field of consensus. Interesting example systems include weighted voting, hierarchies, and crumbling walls.

(And that's another addition to my known unknowns list. ;))

MAJORITY QUORUMS

If we stick with majority quorums, FPaxos allows us to make a simple improvement in the case when the number of acceptors *n* is even. Instead of needing *(n/2)+1* participants in Q2, we can reduce this to *n/2*.

Such a change would be trivial to implement and by reducing the number of acceptors required to participate in replication, we can reduce latency and improve throughput. Furthermore, we have also improved the fault tolerance of the system. As with Paxos, if at most n/2 -1 failures occur then we are guaranteed to be able to make progress. However unlike with Paxos, if exactly n/2 acceptors fails and the leader is still up then we are able to continue to make progress and suffer no loss of availability.

SIMPLE QUORUMS

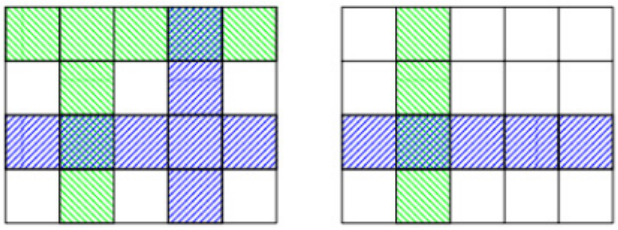
Simple quorums are for me the most natural way of understanding the benefits of FPaxos.

We will use the term simple quorums to refer to a quorum system where any acceptor is able to participate in a quorum and each acceptor's participation is counted equally. Simple quorums are a straightforward generalization of majority quorums.

Since we require only that |Q1|+|Q2|>n and we know that phase two happens a lot more often than phase one, we can reduce the quorum size requirement in phase two and make a corresponding increase in phase one. Take a classic Paxos deployment with five replicas. Whereas traditionally each phase requires at least three nodes (acceptors) to be up, we can tweak this to require four acceptors in a leader-election quorum but only *two* acceptors for Q2.

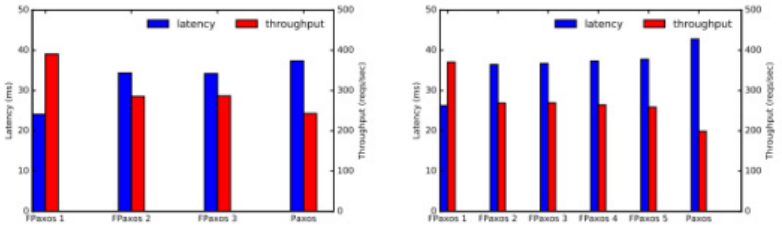
FPaxos will always be able to handle up to |Q2|-1 failures. However, if between |Q2| to N-|Q2| failures occur, we can continue replication until a new leader is required.

If we choose five replicas with |Q1|=4 and |Q2|=2, we therefore will always be able to handle any single failure, and we'll be able to continue replication with the loss of up to three replicas, so long as we don't need a new leader.



(a) Paxos (b) FPaxos

Figure 3: Example of using a 5 by 4 grid to form quorums for a system of 20 acceptors



(a) Performance of FPaxos and LibPaxos3 with 5 replicas.

(b) Performance of FPaxos and LibPaxos3 with 8 replicas.

Figure 4: Throughput and average latency of FPaxos with various quorum sizes and LibPaxos3.

(Howard et al. 2016)

Above are some results from the evaluation showing how the latency and throughput of FPaxos compares to regular Paxos with varying Q2 quorum sizes. (Figure 4)

GRID QUORUMS

Grid quorum schemes arrange the N nodes into a matrix of N1 columns by N2 rows, where N1*N2=N and quorums are composed of rows and columns. As with many other quorum systems, grid quorums restrict which combinations of acceptors can form valid quorums. This restriction allows us to reduce the size of quorums whilst still ensuring that they intersect.

Whereas with simple quorums any reduction in |Q2| must be paid for by a corresponding increase in |Q1|, with grid quorums we can make different trade-offs between quorum size, quorum selection flexibility, and fault tolerance. (Figure 3)

Since Paxos requires all quorums to intersect, one suitable scheme would be to require one row and one column to form a quorum: (a) in the figure above.

In FPaxos, we can safely reduce our quorums to one row of size N1 for Q1, and one column of size N2 for Q2. This construction is interesting as quorums from the same phase will never intersect, and may be useful in practice for evenly distributing the load of FPaxos across a group of acceptors.

A FURTHER ENHANCEMENT

FPaxos only requires that a given Q1 will intersect with all Q2s with *lower proposal numbers*. Given a mechanism to learn which Q2s have participated in lower numbered proposals, we have further flexibility in choosing a Q1 quorum.

The implications of this enhancement can be far reaching. For example, in a system of N=100 f nodes, a leader may start by announcing a fixed Q2 of size f+1 and all higher proposal numbers (and readers) will need to intersect with only this Q2. This allows us to tolerate N-f failures.

(And many other interesting variations are possible.)

THE BOTTOM LINE

Generalizing existing systems to use FPaxos should be quite straightforward. Exposing replication (phase two) quorum size to developers would allow them to choose their own trade-off between failure tolerance and steady state latency.

Secondly, by no longer requiring replication quorums to intersect, we have removed an important limit on scalability. Through smart quorum construction and pragmatic system design, we believe a new breed of scalable, resilient, and performant consensus algorithms is now possible.

ON DESIGNING AND DEPLOYING INTERNET-SCALE SERVICES

Want to know how to build cloud-native applications? You'll be hard-pushed to find a better collection of wisdom, best practices, and hard-won experience than "On designing and deploying Internet-scale services" from James Hamilton (LISA 2007).

Hamilton, 2007

It's amazing to think that all of this knowledge was captured and written down nine years ago — cloud-native isn't as new as you think! There's so much goodness in this paper, and in such condensed form, that it's hard to do it justice in a summary. If this write-up catches your interest, I fully recommend going on to read the original. The last time I read this paper, I was so struck by the value of the content that I based a services checklist off of it, which you might find handy if you're reviewing or designing a system.

The paper itself consists of three high-level tenets, followed by 10 subsections covering different aspects of designing and deploying operations-friendly services. I love this reminder that good ops start in design and development, and are not something you can just bolt on at the end:

> ***We have long believed that 80% of operations issues originate in design and development.... When systems fail, there is a natural tendency to look first to operations since that is where the problem actually took place. Most operations issues, however, either have their genesis in design and***

It's amazing to think that all of this knowledge was captured and written down nine years ago — cloud-native isn't as new as you think!

development or are best solved there. (All quotes from Hamilton 2007.)

Onto the three tenets, which form a common thread through all the other recommendations:

1. Expect failures.
2. Keep things simple. Complexity breeds problems and simpler things are easier to get right.
3. Automate everything. People make mistakes. People need sleep. People forget things. Automated processes are testable and fixable — and therefore ultimately much more reliable.

Simplicity is the key to efficient operations.

And the 10 subsections of recommendations break into the following categories:

1. overall application design (21 best practices);
2. automatic management and provisioning (11 best practices);
3. dependency management (six best practices);
4. release cycle and testing (12 best practices);
5. hardware selection and standardization (four best practices, and perhaps the section of the paper that has dated the most, but the overall point that you shouldn't depend on any special pet hardware is still spot-on);
6. operations and capacity planning (five best practices);

7. auditing, monitoring and alerting (10 best practices);

8. graceful degradation and admission control (three best practices);

9. customer and press communication plan (put one in place — before you need it!); and

10. customer self-provisioning and self-help (all customers to help themselves whenever possible).

Perhaps now you see why I say this paper is so hard to summarize — what to leave out?! Rest assured that all of the above are captured in my checklist. Some of the advice, though new at the time, has often since been repeated. I've tried to pick out some of the things that are less often discussed, give special food for thought, or are just so well put that I couldn't resist sharing them.

Four years before Netflix started talking about Chaos Monkey, Hamilton wrote:

The acid test for full compliance with this design principle is the following: is the operations team willing and able to bring down any server in the service at any time without draining the work load first?

He goes on to recommend that we analyse component-failure modes and the combinations thereof using the same approaches as we might use for security threat modelling: considering each potential threat and implementing adequate mitigation.

Very unusual combinations of failures may be determined sufficiently unlikely that ensuring the system can operate through them is uneconomical. Be cautious when making this judgment. We've been surprised at how frequently "unusual" combinations of events take place when running thousands of servers that produce millions of opportunities for component failures each day.

Strive to have only a single version of your service. The most economic services don't give customers control over the version they run, and only host one version. Recognize that multiple versions will be live during rollout and production testing. Versions n and n+1 of all components need to coexist peacefully.

Partition the system in such a way that partitions are infinitely adjustable and fine-grained, and not bounded by any real-world entity (person, collection, customer etc.).

If the partition is by company, then a big company will exceed the size of a single partition. If the partition is by name prefix, then eventually all the P's, for example, won't fit on a single server.

Designing for automation may involve significant service-model constraints:

Automating administration of a service after design and deployment can be very difficult. Successful automation requires simplicity and clear, easy-to-make operational decisions. This in turn depends on a careful service design that, when necessary, sacrifices some latency and throughput to ease automation. The trade-off is often difficult to make, but the administrative savings can be more than an order of magnitude in high-scale services.

The example Hamilton gives in the paper is choosing to use synchronous replication in order to simplify the decision to failover, avoiding the complications of asynchronous replication.

Keep configuration and code as a unit throughout the lifecycle.

The most appropriate level to handle failures is the service level.

Handle failures and correct errors at the service level where the full execution context is available rather than in lower software levels. For example, build redundancy into the service rather than depending upon recovery at the lower software layer.

Be careful with your dependencies! It's interesting to contrast Hamilton's advice here with the trend towards microservices architectures.

Dependency management in high-scale services often doesn't get the attention the topic deserves. As a general rule, dependence on small components or services doesn't save enough to justify the complexity of managing them. Dependencies do make sense when: (1) the components being depended upon are substantial in size or complexity, or (2) the service being depended upon gains its value in being a single, central instance.

If you do introduce dependencies then: expect latency; isolate failures; use proven components (third-party deps); implement inter-service monitoring and alerts; decouple components so that operation can continue in degraded mode if a dependency fails; and remember that:

Dependent services and producers of dependent components need to be committed to at least the same SLA as the depending service.

Don't bother trying to create full staging environments as close as possible to production (as you get ever closer to production realism, the cost goes asymptotic and rapidly approaches that of production).

We instead recommend taking new service releases through standard unit, functional, and production test lab testing and then going into limited production as the final test phase.

To be able to do this with confidence you need to follow four rules:

- The production system must have sufficient redundancy to be able to quickly recover from a catastrophic failure.
- Data corruption or state-related failures have to be extremely unlikely (functional tests should be passing).
- Errors must be detected and the engineering team must be monitoring system health of the code in test.
- It must be possible to quickly roll back all changes and this rollback must be tested before going into production.

Another potentially counter-intuitive approach we favor is deployment mid-day rather than at night. At night, there is greater risk of mistakes. And, if anomalies crop up when deploying in the middle of the night, there are fewer engineers around to deal with them.

Ship often! Though Hamilton's definition of "often" is definitely showing its age.

We like shipping on a three-month cycle, but arguments can be made for other schedules. Our gut feel is that the norm will eventually be less than three months, and many services are already shipping on weekly schedules.

Three months sounds glacial to me today!

Use production data to find problems.

Quality assurance in a large-scale system is a data-mining and visualization problem, not a testing problem. Everyone needs to focus on getting the most out of the volumes of data in a production environment.

For test and development, make it easy to deploy the entire service on a single system. Where this is impossible for some component, write an emulator.

Without this, unit testing is difficult and doesn't fully happen. And if running the full system is difficult, developers will have a tendency to take a component view rather than a systems view.

Automate the procedure to move state off of damaged systems if the worst happens.

The key to operating services efficiently is to build the system to eliminate the vast majority of operations administrative interactions. The goal should be that

a highly reliable 24×7 service should be maintained by a small 8×5 operations staff.

Make the development team responsible.

If the development team is frequently called in the middle of the night, automation is the likely outcome. If operations is frequently called, the usual reaction is to grow the operations team.

Instrument everything: data is the most valuable asset. There's a very good list of what to capture in the paper.

The operations team can't instrument a service in deployment. Make substantial effort during development to ensure that performance data, health data, throughput data, etc. are all produced by every component in the system.

Control and meter admission.

It's vital that each service have a fine-grained knob to slowly ramp up usage when coming back on line or recovering from a catastrophic failure.

I'm going to leave things there, but at risk of repeating myself, there's so much good advice that I had to leave out in this summary that if the subject matter at all interests you, it's well worth reading the full paper.

> **Quality assurance in a large-scale system is a data-mining and visualization problem, not a testing problem. Everyone needs to focus on getting the most out of the volumes of data in a production environment.**

ON THE CRITERIA TO BE USED IN DECOMPOSING SYSTEMS INTO MODULES

David L. Parnas's "On the criteria to be used in decomposing systems into modules" (1971) is a true classic. If we give its title a slight twist to "On the criteria to be used in decomposing systems into services", it's easy to see how this 45-year-old paper can speak to contemporary issues.

Parnas, 1971

And from the very first sentence of the abstract, you'll find some shared goals with modern development: "This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time." Flexibility (we tend to call it agility) and faster development times remain top of mind today. Comprehensibility less so, but perhaps we should be paying more attention there. Say you're all bought into cloud-native, microservices-based architectures. Is splitting up your system into multiple independent services going to help you achieve your goals? Parnas brings a keen insight:

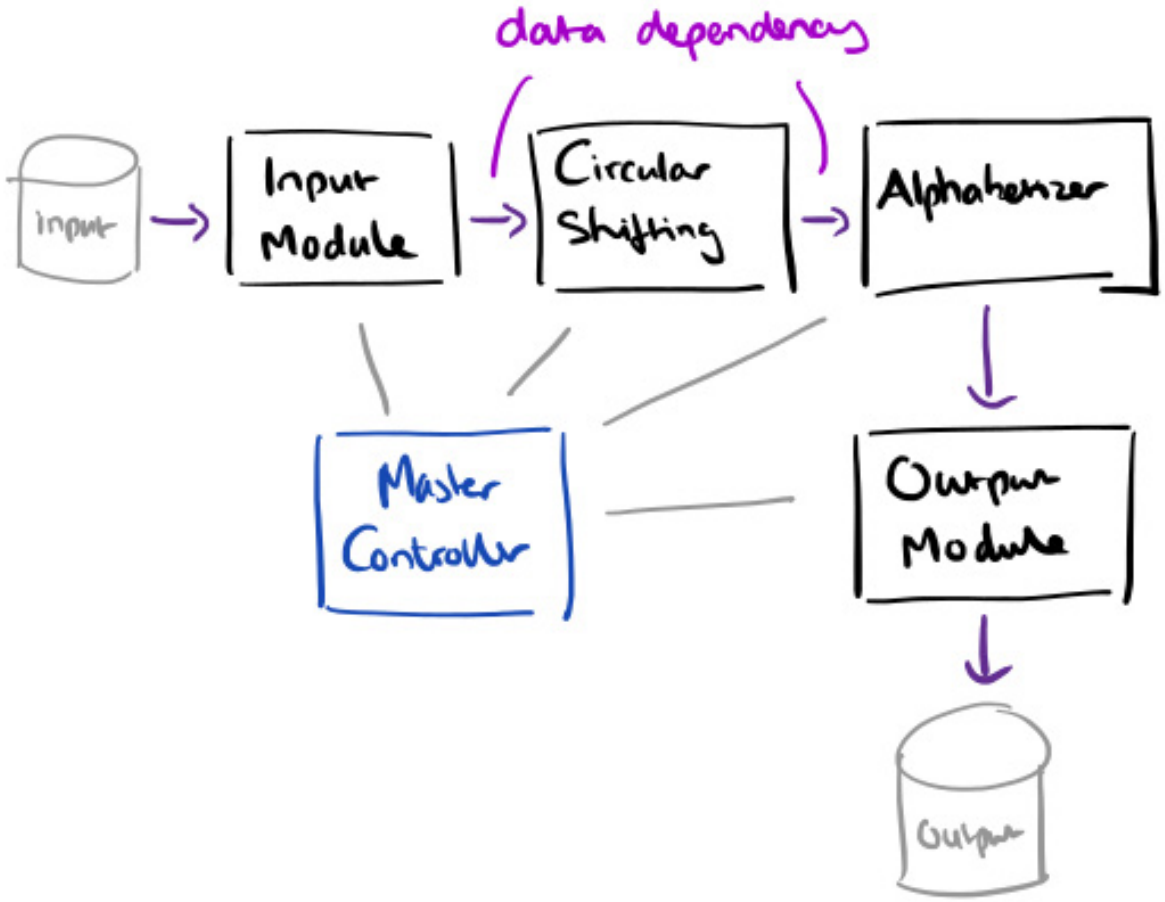
The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. (All quotes from Parnas 1971.)

It's a timely reminder that it's not simply about having lots of small modules: a large part of the success or otherwise of your system depends on *how you choose to divide the system into modules in the first place*. When Parnas talks about a "module" in the paper, his definition is a *work-assignment* unit rather than a subprogram unit.

Parnas sets out three expected benefits of modular programming. We can look at those through the lens of microservices, too:

1. Development time should be shortened because separate groups can work on each module (microservice) with

The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules.



little need for communication.

2. Product flexibility should be improved — it was hoped that it would be possible to make drastic changes or improvements in one module (microservice) without changing others.

3. It was hoped that the system could be studied a module (microservice) at a time with the result that the whole system could be better designed because it was better understood — i.e., increase comprehensibility.

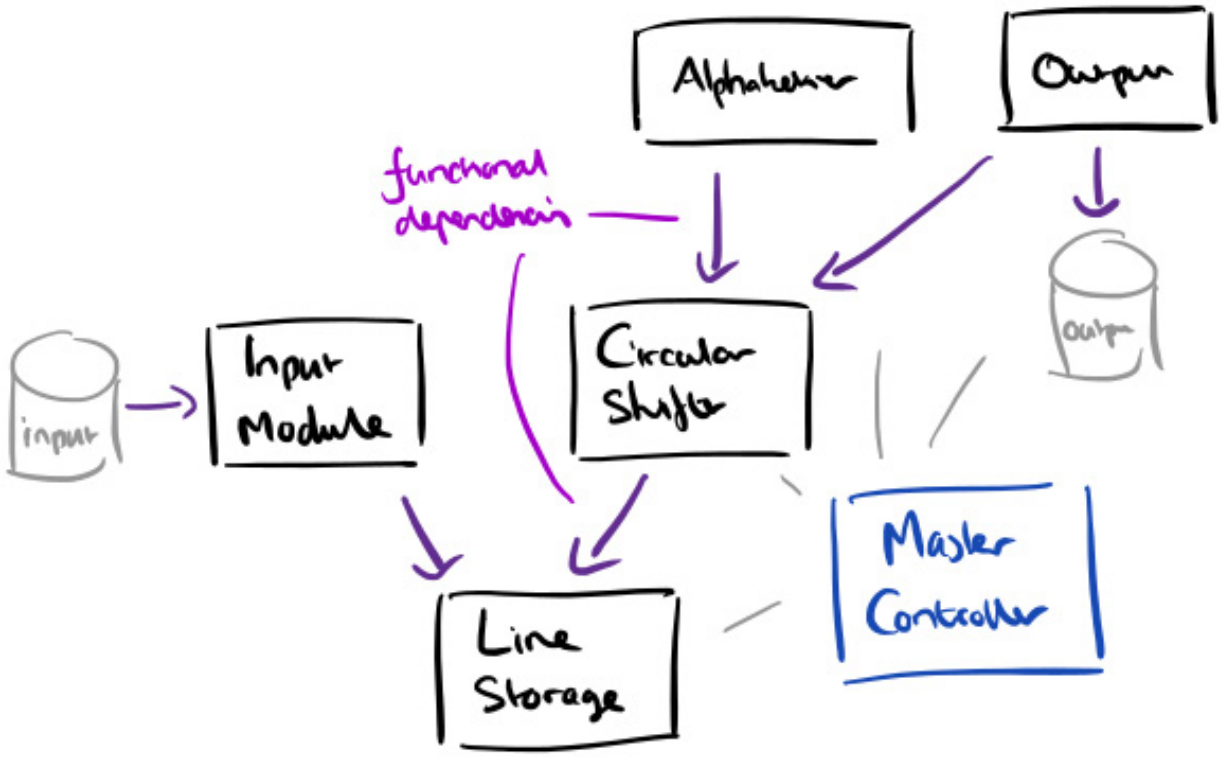
Different ways of dividing the system into modules bring with them different communication and coordination requirements between the individuals (or teams) working on those modules, and help to realise the benefits outlined above to greater or lesser extents.

The famous example at the heart of the paper is the development of a system to generate a "KWIC" index given an ordered set of lines as input. Any line can be circularly shifted by repeatedly removing the first word and adding it to the end of the line. The system outputs a listing of all circular shifts of all lines in alphabetical order. Parnas acknowledges that this is somewhat of a toy example, but treats it as if it were a large project.

Parnas examines two decompositions. In the first, each major step or task in the processing workflow is made into its own independent module (service). This leads to five modules as shown in the diagram above:

1. an input module that reads data lines from the input medium,
2. a circular shifter,
3. an "alphabetizer" (sorter),
4. an output module that creates a nicely formatted output, and
5. a master control module that sequences the other four.

The first four modules operate on shared data structures in memory.



This is a modularization in the sense meant by all proponents of modular programming. The system is divided into a number of relatively independent modules with well-defined interfaces; each one is small enough and simple enough to be thoroughly understood and well programmed. Experiments on a small scale indicate that this is approximately the decomposition (that) would be proposed by most programmers for the task specified.

As shown in the diagram above the second decomposition looks similar on the surface:

1. a line-storage module with routines for operating on lines,
2. an input module as before but which calls the line-storage module to have lines stored internally,
3. a circular shifter that shifts lines using the line-storage module,
4. an alphabetizer,
5. an output module that builds upon the circular shifter functions, and
6. a master control module.

This decomposition, however, was created on the basis of *information hiding*.

There are a number of design decisions (that) are questionable and likely to change under many circumstances.... It is by looking at changes such as these that we can see the differences between the two modularizations.

In the first decomposition, many changes (for example, the decision to have all lines stored in memory) require changes in every module but the second decomposition confines many more potential changes to a single module.

Furthermore, the interfaces between modules in the first decomposition are fairly complex formats and represent design decisions that cannot be taken lightly.

The development of those formats will be a major part of the module development and that part must be a joint effort among the several development groups.

The second decomposition's interfaces are simpler and more abstract, leading to faster independent development of modules.

Regarding comprehensibility, the first decomposition's system can only really be understood as a whole. Parnas notes, "It is my subjective judgement that this is not true in the second modularization."

Every module in the second decomposition is characterized by its knowledge of a design decision (that) it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings.

There's a potential drawback to the second decomposition though, which is even more important when packaging modules as independent services:

If we are not careful, the second decomposition will prove to be much less efficient. If each of the "functions" is actually implemented as a procedure with an elaborate calling sequence, there will be a great deal of such calling due to the repeated switching between modules. The first decomposition will not suffer from this problem because there is a relatively infrequent transfer of control between the modules.

To avoid this overhead, Parnas recommends a tool that enables programs to be written as if the functions were subroutines, but *assembled* via whatever mechanism is appropriate. This is more challenging in a microservices world!

In conclusion, while much attention is given to the need to divide a system into modules (microservices), much less attention has been given to the *criteria* by which we decide on module boundaries. As Parnas shows us, it might be a good idea to think about those criteria in your next project, as they have a strong influence on development time, system agility, and comprehensibility.

We have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions (that) are likely to change. Each module is then designed to hide such a decision from the others. Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in the processing.

HERE IS WHAT WE'VE COVERED IN THE PREVIOUS ISSUES

A SURVEY OF AVAILABLE CORPORA FOR BUILDING DATA-DRIVEN DIALOGUE SYSTEMS

HOW TO BUILD STATIC CHECKING SYSTEMS USING ORDERS OF MAGNITUDE LESS CODE

DEEP LEARNING IN NEURAL NETWORKS: AN OVERVIEW

THE AMAZING POWER OF WORD VECTORS

GORILLA: A FAST, SCALABLE, IN-MEMORY TIME-SERIES DATABASE

ARIES

NO COMPROMISES

HYPERLOGLOG IN PRACTICE

NOT-QUITE-SO-BROKEN TLS

ALL FILE SYSTEMS ARE NOT CREATED EQUAL