LONDON
INTERNATIONAL
SOFTWARE DEVELOPMENT
CONFERENCE 2015

goto; conference

✨Keynote✨

Join the conversation #gotoldn

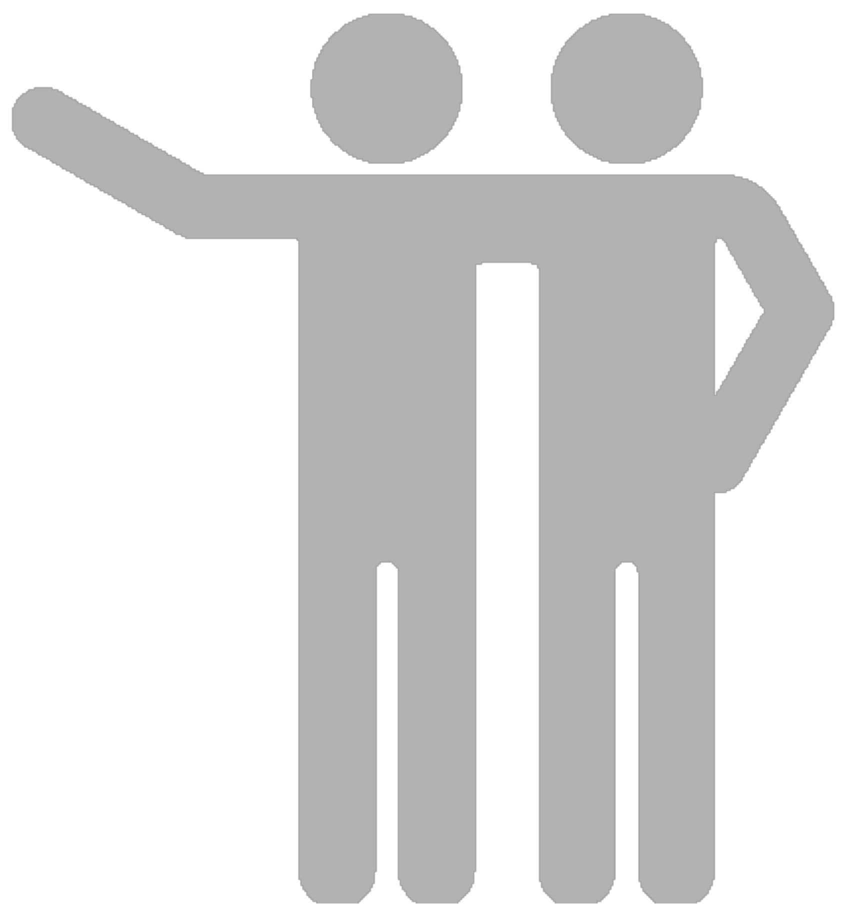
Workshops: Sept 14-15 // Conference: Sept 16-18, 2015

Agile, Lean, Rugged

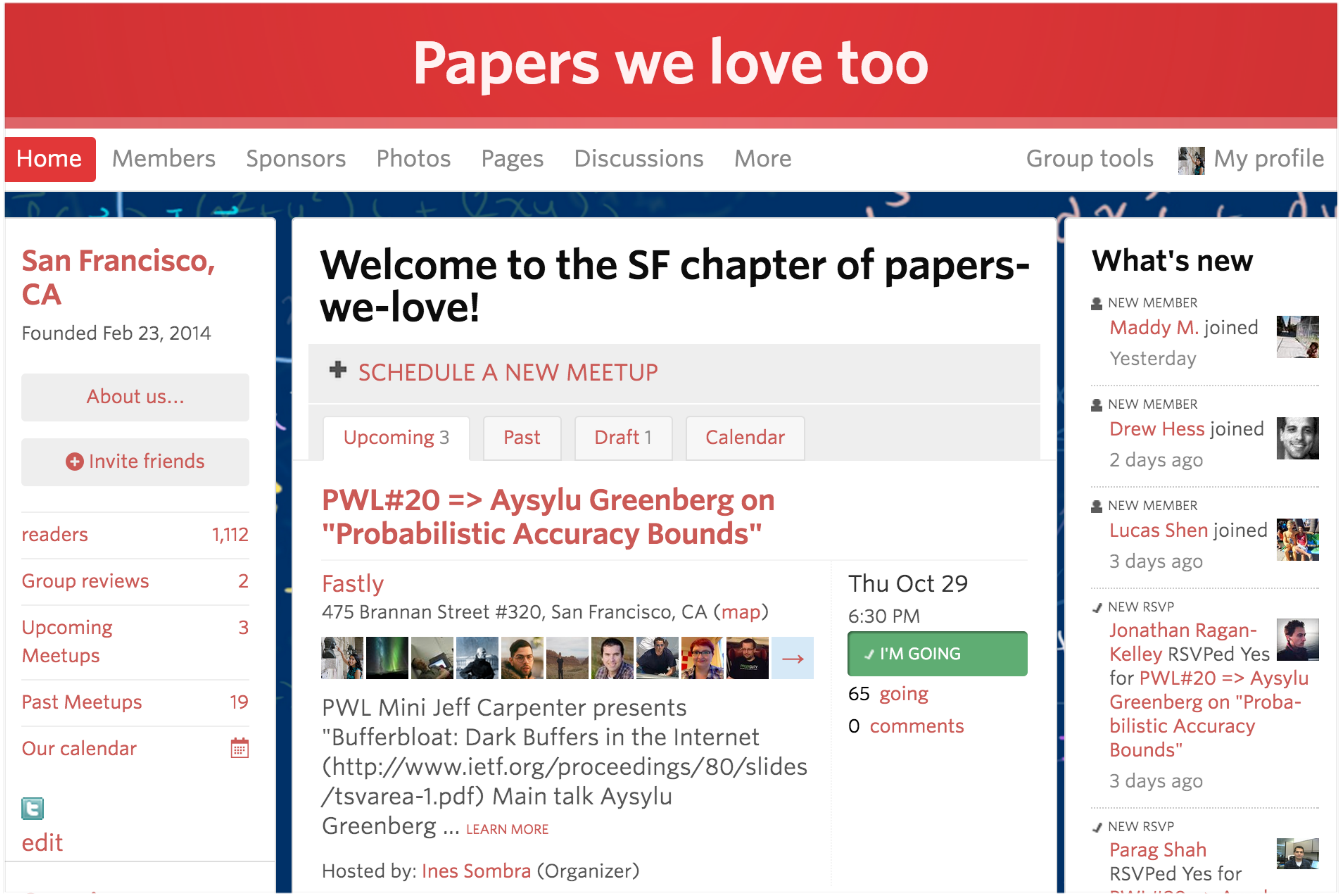
The Paper Edition!

First

.Introductions



Ines Sombra



@Randommood

the morning paper

an interesting/influential/important paper from the world of CS every weekday morning, as selected by Adrian Colyer

Mining and Summarizing Customer Reviews

AUGUST 28, 2015

Mining and Summarizing Customer Reviews – Hu and Liu 2004

This is the third of the three 'test-of-time' award winners from KDD'15. From the awards page:

> *The paper introduces the problem of summarizing customer reviews and decomposes the problem into the three steps of (1) mining product features (aspects), (2) identifying opinion sentences and their corresponding feature in each review and (3) summarizing the results. The paper has inspired the new research direction of Aspect-Based Sentiment Analysis/Aspect-Based Opinion Mining, and the proposed framework has been widely adopted in research and applications, as seen from the very large number of citations.*

The goal is to mine an existing corpus of product reviews and produce summaries of the form:

```
Digital Camera XYZ:
  Feature: Picture Quality
     Positive: 253
        "Overall this is a good camera with a really good pict
```

SUBSCRIBE

never miss an issue! The Morning Paper delivered straight to your inbox.

SEARCH

type and press enter

ARCHIVES

Select Month

MOST READ IN THE LAST FEW DAYS

MillWheel: Fault-Tolerant Stream Processing at

Adrian Colyer

@adriancolyer

A challenge!

Foundation

The Rules

No Cheating!

Only 5 minutes per paper

Frontier

A paper tour of

Agile

Foundation

Software Aging

Invited Plenary Talk

David Lorge Parnas

Communications Research Laboratory
Department of Electrical and Computer Engineering
McMaster University, Hamilton, Ontario, Canada L8S 4K1

ABSTRACT

Programs, like people, get old. We can't prevent aging, but we can understand its causes, take steps to limits its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable. A sign that the Software Engineering profession has matured will be that we lose our preoccupation with the first release and focus on the long term health of our products. Researchers and practitioners must change their perception of the problems of software development. Only then will Software Engineering deserve to be called Engineering.

inevitable, but like human aging, there are things that we can do to slow down the process and, sometimes, even reverse its effects.

Software aging is not a new phenomenon, but it is gaining in significance because of the growing economic importance of software and the fact that increasingly, software is a major part of the "capital" of many high-tech firms. Many old software products have become essential cogs in the machinery of our society. The aging of these products is impeding the further development of the systems that include them.

The authors and owners of new software products often look at aging software with disdain. They be-

We disdain old software

"The only systems that don't get changed are those that are so bad nobody wants to use them"

When software gets older

Preventative medicine

Design for change

Embrace modularity & information hiding

Stress clarity & documentation

Amputate disease-ridden parts

Plan for eventual replacement

Fast Database Restarts at Facebook

Aakash Goel,* Bhuwan Chopra, Ciprian Gerea, Dhrúv Mátáni,
Josh Metzler, Fahim Ul Haq, and Janet L. Wiener
Facebook, Inc.

Frontier

ABSTRACT

Facebook engineers query multiple databases to monitor and analyze Facebook products and services. The fastest of these databases is Scuba, which achieves subsecond query response time by storing all of its data in memory across hundreds of servers. We are continually improving the code for Scuba and would like to push new software releases at least once a week. However, restarting a Scuba machine clears its memory. Recovering all of its data from disk — about 120 GB per machine — takes 2.5-3 hours to read and format the data per machine. Even 10 minutes is a long downtime for the critical applications that rely on Scuba, such as detecting user-facing errors. Restarting only 2% of the servers at a time mitigates the amount of unavailable data, but prolongs the restart duration to about 12 hours, during which users see only partial query results and one engineer needs to monitor the servers carefully. We need a faster, less engineer intensive, solution to enable frequent software upgrades.

1. INTRODUCTION

Facebook engineers query multiple database systems to monitor and analyze Facebook products and services. Scuba[5] is a very fast, distributed, in-memory database used extensively for interactive, ad hoc, analysis queries. These queries typically run in under a second over GBs of data. Scuba processes almost a million queries per day for over 1500 Facebook employees. In addition, Scuba is the workhorse behind Facebook's code regression analysis, bug report monitoring, ads revenue monitoring, and performance debugging.

One significant source of downtime is software upgrades, yet upgrades are necessary to introduce new features and apply bug fixes. At Facebook, we are accustomed to the agility that comes with frequent code deployments. New code is rolled out to our web product multiple times each week [9]. The Facebook Android Alpha program also releases code multiple times a week [18, 17]. We would like to deploy new code to Scuba at least once a week as well.

However, any downtime on Scuba's part is a problem for

What do we want?

We want agile

Development

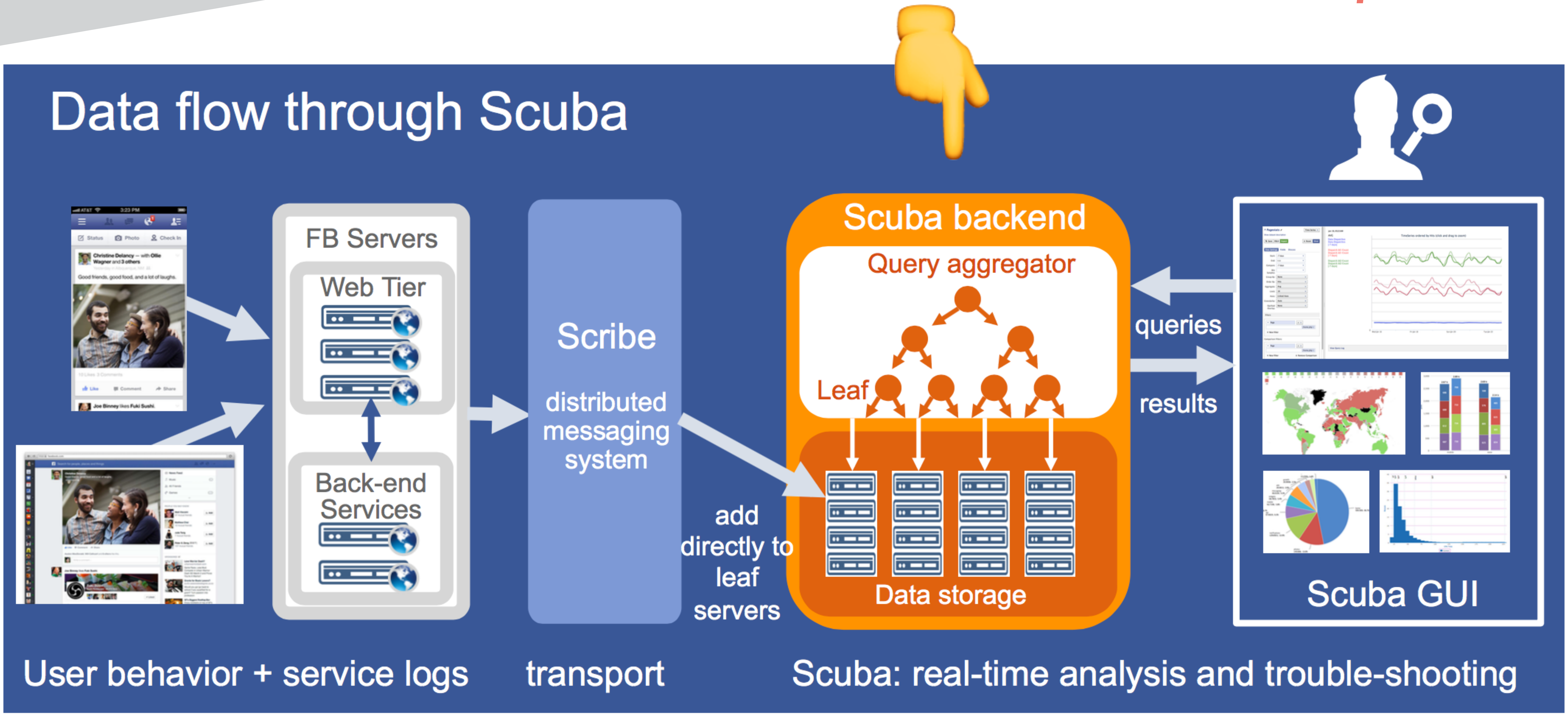
Testing and verification

Delivery

and we want ***agility of operations*** *too!*

Facebook Scuba

Data lives in server's heap



The problem with state

Operationally expensive & slow!

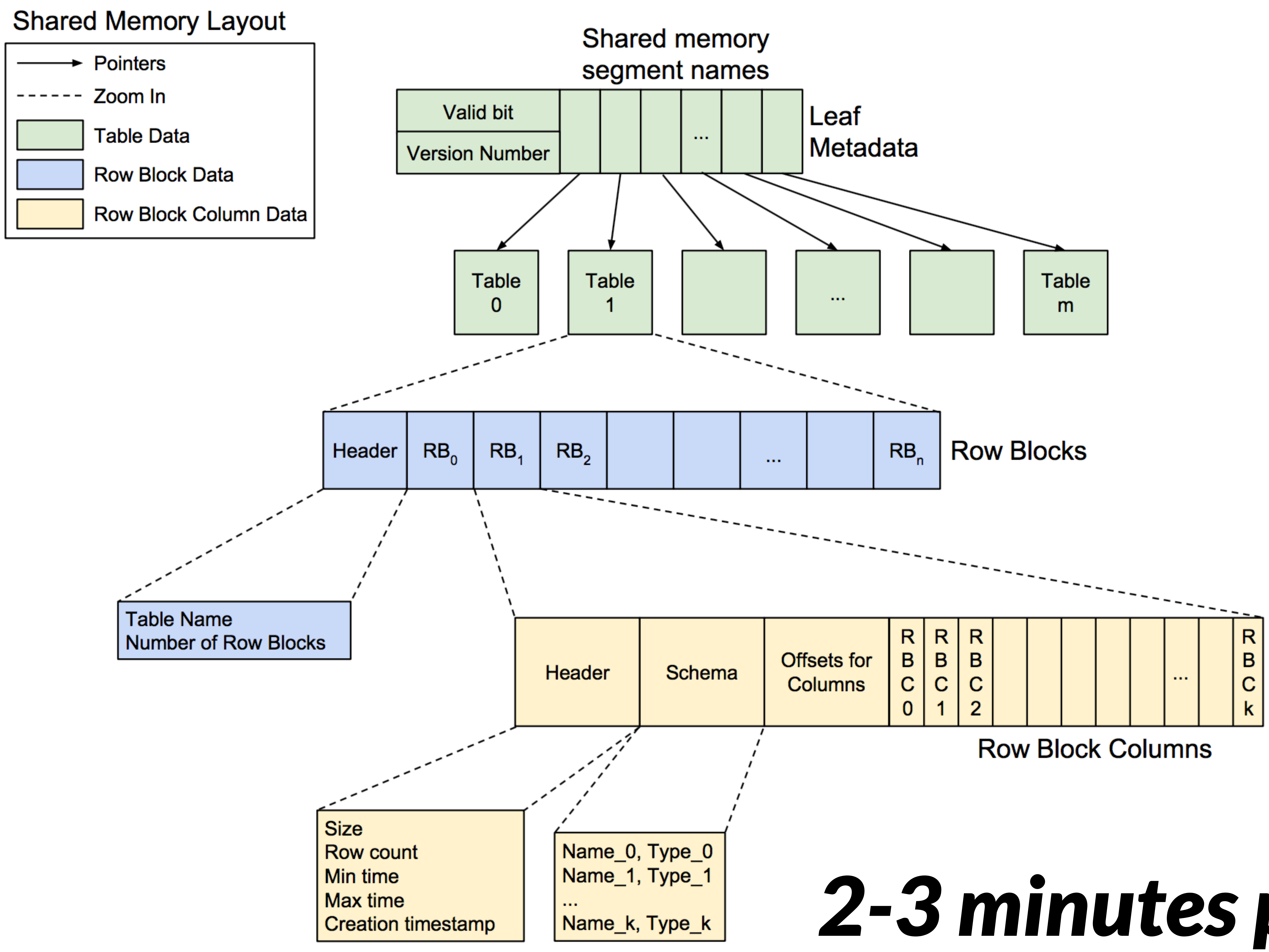
Restarting a database clears its memory

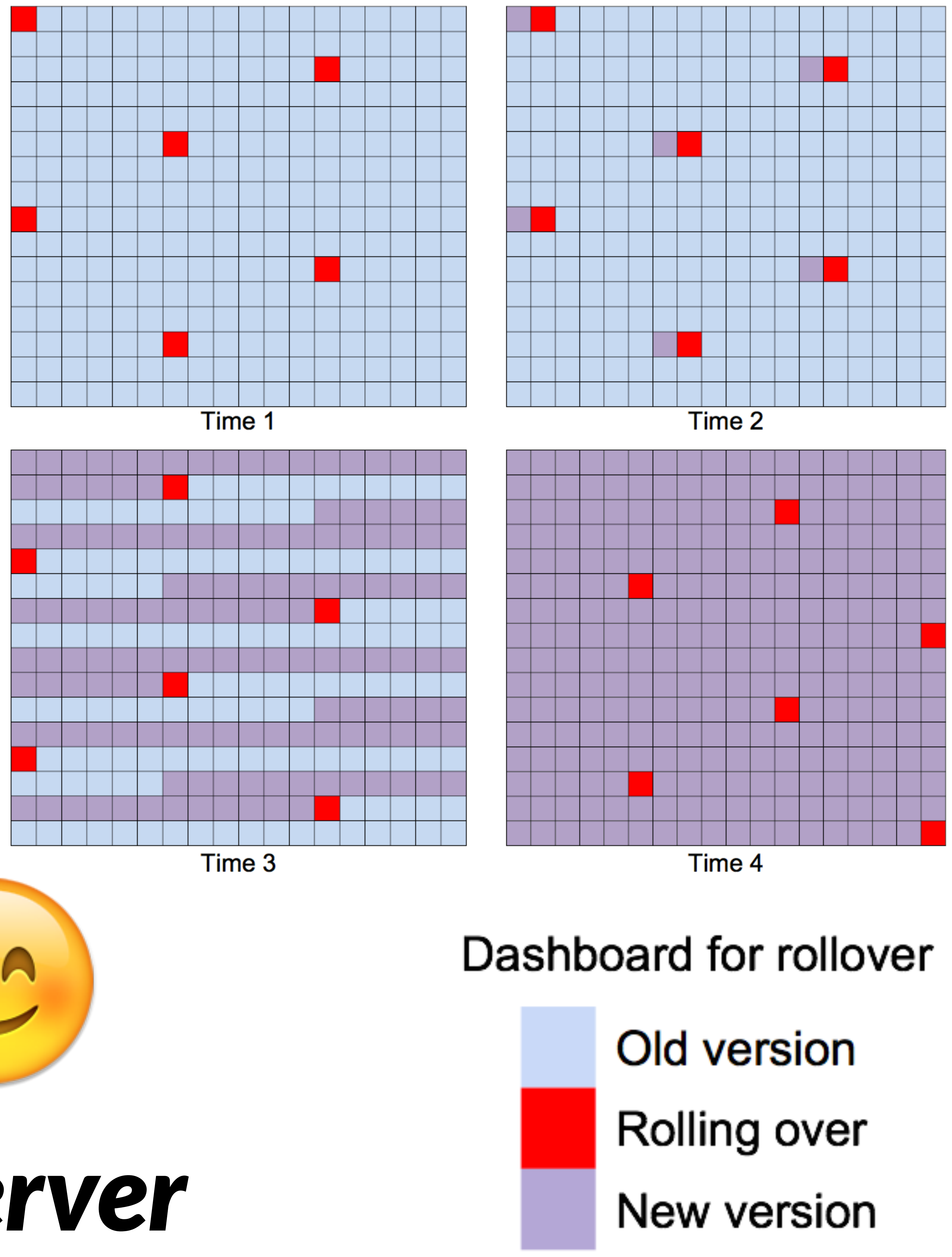
Reading 120GB of data from disk takes about 3 hours per server (8 per machine)

*Even with **orchestrated restarts** & **partial queries** total of **~12 hours to restart a fleet***

"When we shutdown a server for a planned upgrade, we know that the memory state is good… so we decided to decouple the memory's lifetime from the process's lifetime"

Fleet restarts < 1 hour now!

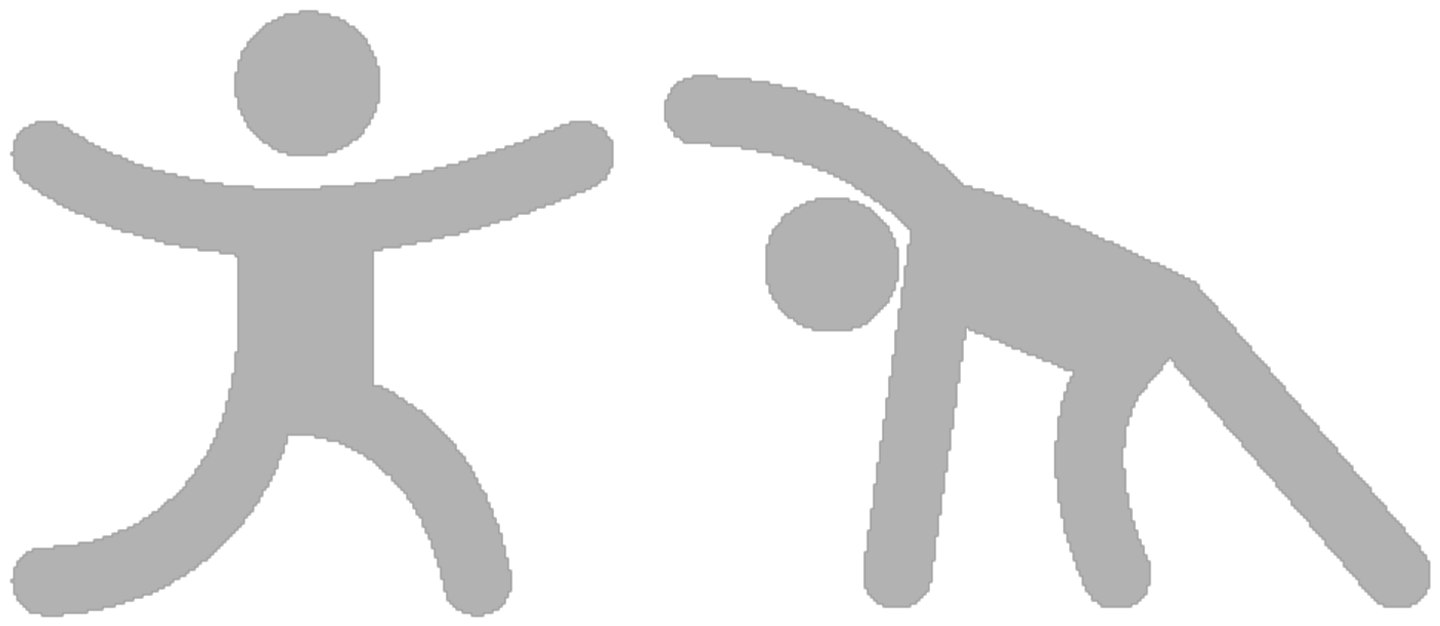




2-3 minutes per server

A paper tour of

Lean



Scalability! But at what COST?

Frank McSherry
Unaffiliated

Michael Isard
Microsoft Research

Derek G. Murray
Unaffiliated*

Abstract

We offer a new metric for big data platforms, COST, or the Configuration that Outperforms a Single Thread. The COST of a given platform for a given problem is the hardware configuration required before the platform outperforms a competent single-threaded implementation. COST weighs a system's scalability against the overheads introduced by the system, and indicates the actual performance gains of the system, without rewarding systems that bring substantial but parallelizable overheads.

We survey measurements of data-parallel systems recently reported in SOSP and OSDI, and find that many systems have either a surprisingly large COST, often hundreds of cores, or simply underperform one thread for all of their reported configurations.

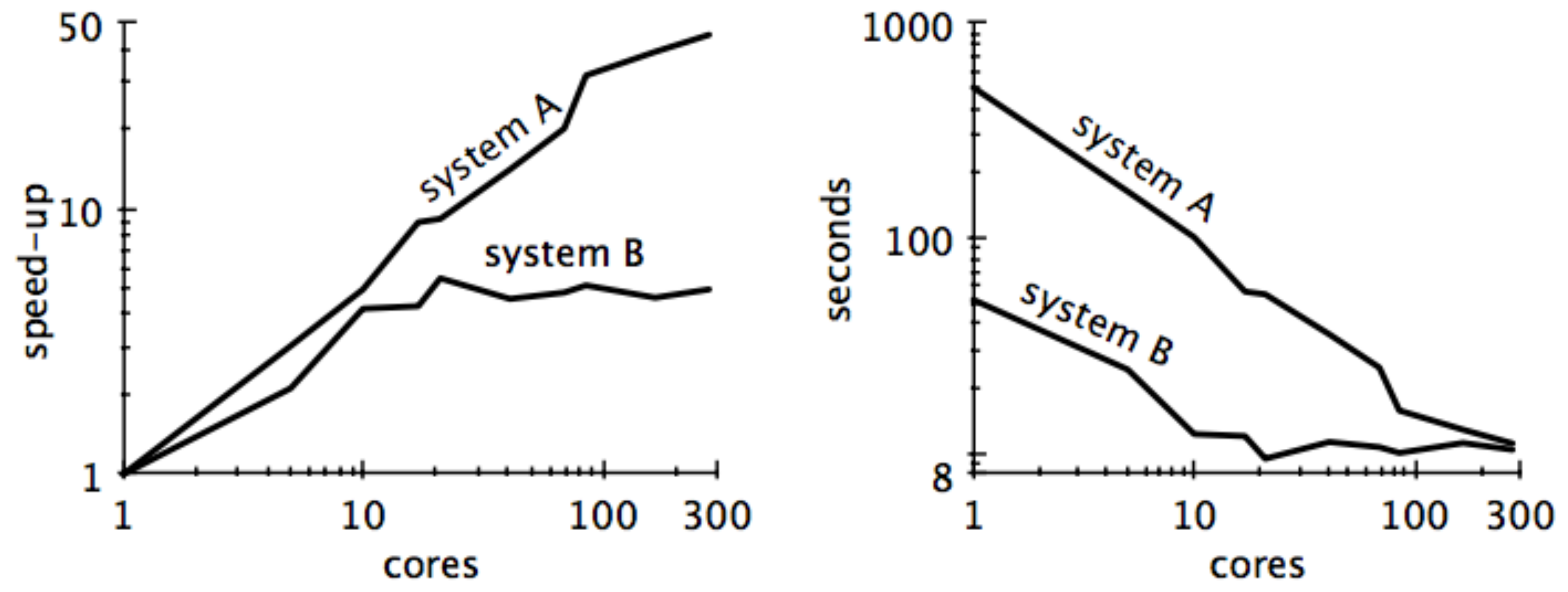
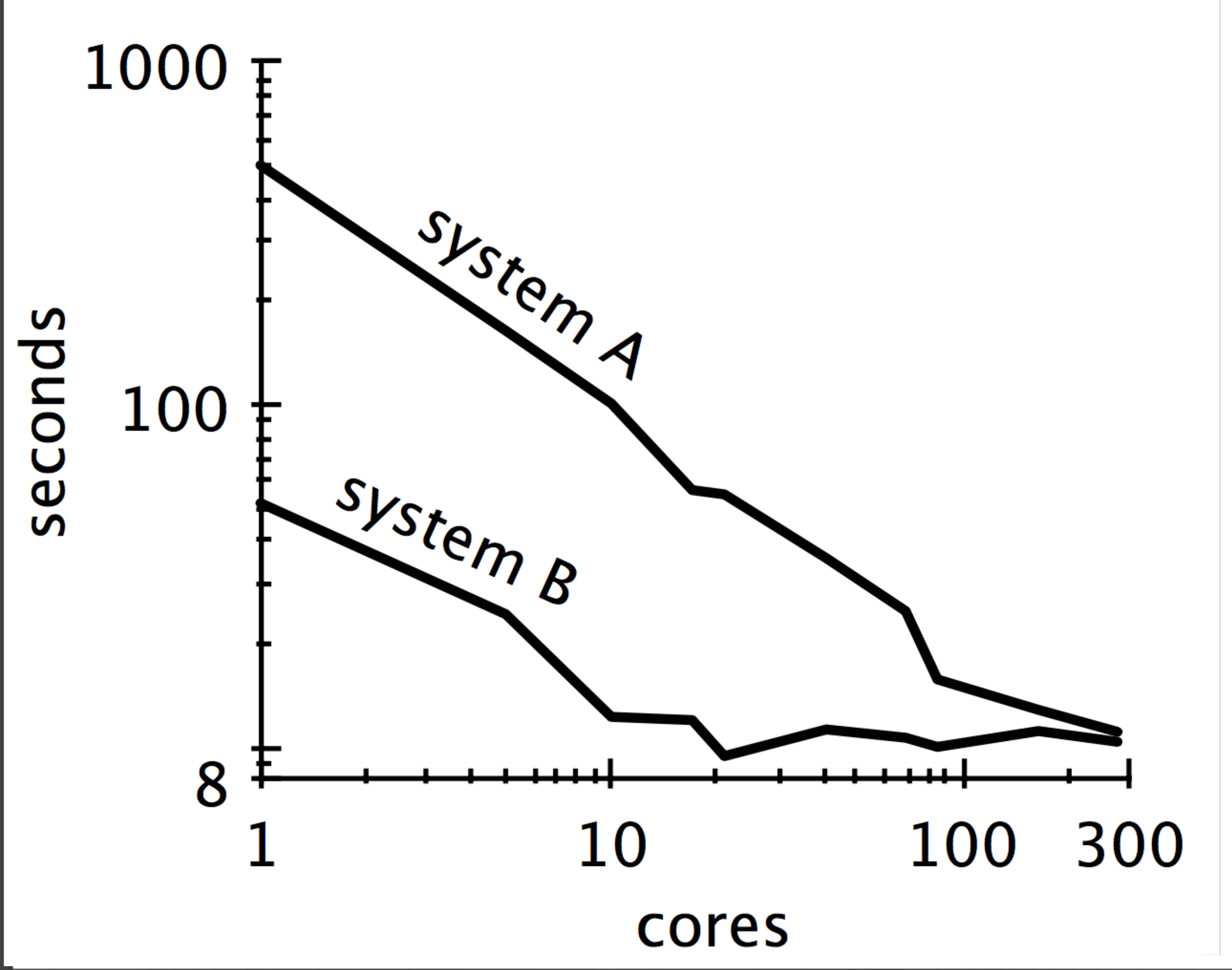


Figure 1: Scaling and performance measurements for a data-parallel algorithm, before (system A) and after (system B) a simple performance optimization. The unoptimized implementation "scales" far better, despite (or rather, because of) its poor performance.

argue that many published big data systems more closely resemble system A than they resemble system B.

Foundation

Which system is better?



Single-minded pursuit of scalability is the wrong goal

Why does this happen?

Common wisdom

Effective scaling is ***evidence of solid system building***

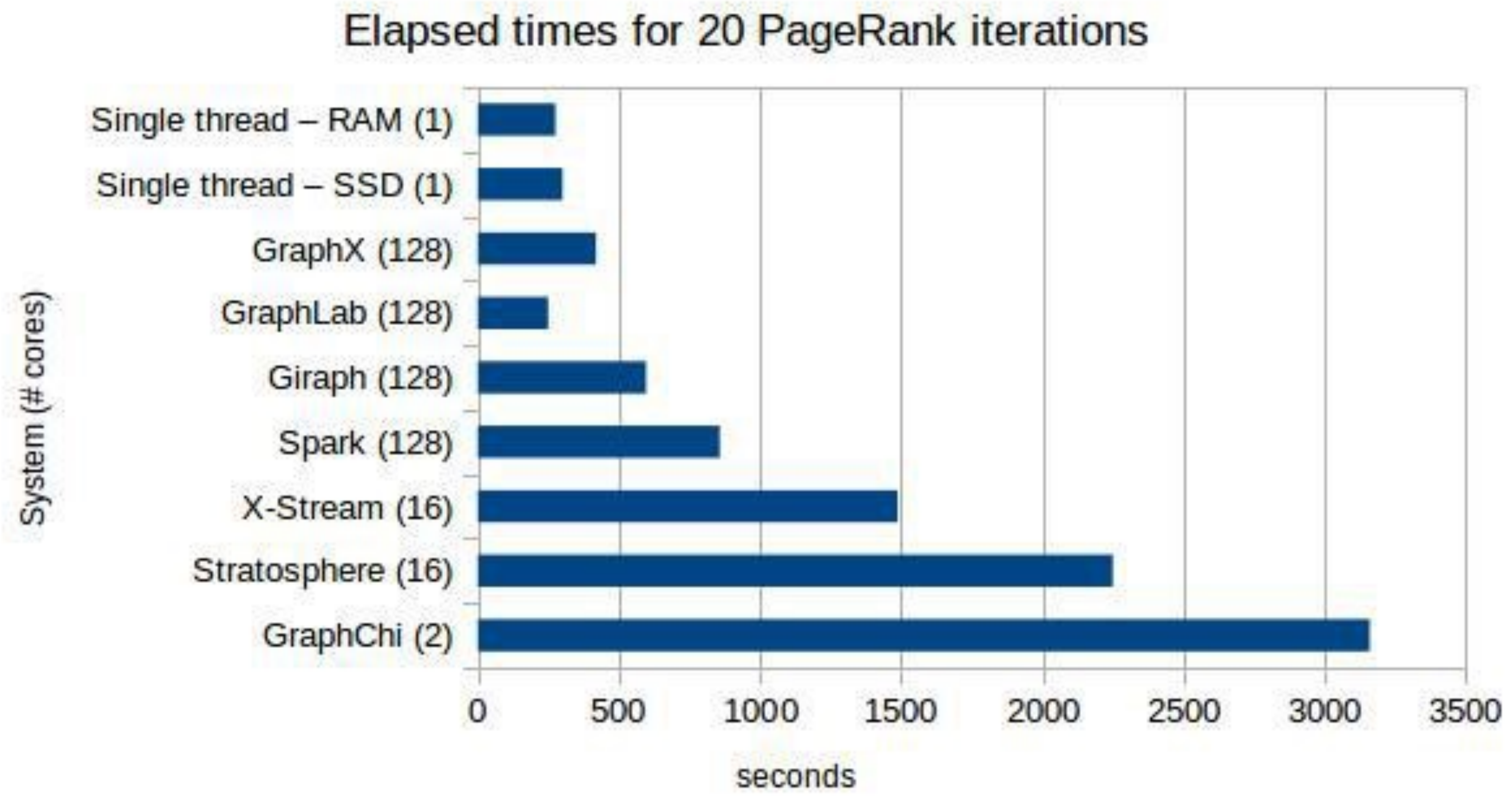
McSherry et al.

Any system can scale arbitrarily well with a ***sufficient lack of care*** *in its implementation*

COST

*Configuration that **outperforms a single thread***

*COST of a system is the **hardware platform** (number of cores) **required before the platform outperforms a competent single threaded implementation***


Elapsed times for 20 PageRank iterations
Single thread – RAM (1)
Single thread – SSD (1)
GraphX (128)
GraphLab (128)
Giraph (128)
Spark (128)
X-Stream (16)
Stratosphere (16)
GraphChi (2)
System (# cores)
0
500
1000
1500
2000
2500
3000
3500
seconds

“If you’re **building** a system, make sure it’s better than your laptop. If you’re **using** a system, make sure it’s better than your laptop”

McSherry

ApproxHadoop: Bringing Approximations to MapReduce Frameworks

Frontier

Íñigo Goiri[†*] Ricardo Bianchini[†‡] Santosh Nagarakatte[‡] Thu D. Nguyen[‡]

[‡]Rutgers University
{ricardob, santosh.nagarakatte, tdnguyen}@cs.rutgers.edu

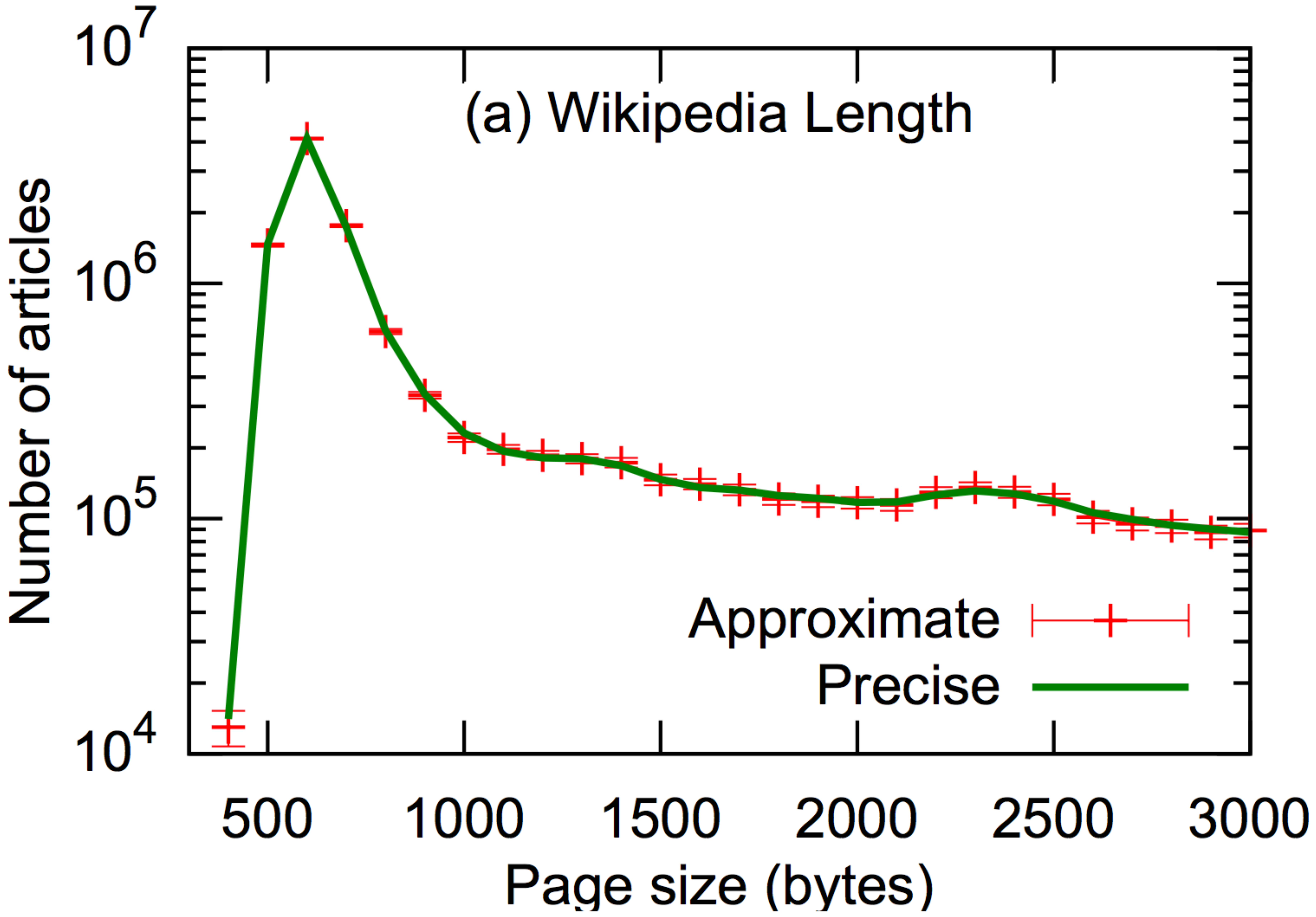
[†]Microsoft Research
{inigog, ricardob}@microsoft.com

Abstract

We propose and evaluate a framework for creating and running approximation-enabled MapReduce programs. Specifically, we propose approximation mechanisms that fit naturally into the MapReduce paradigm, including input data sampling, task dropping, and accepting and running a precise and a user-defined approximate version of the MapReduce code. We then show how to leverage statistical theories to compute error bounds for popular classes of MapReduce programs when approximating with input data sampling and/or task dropping. We implement the proposed mechanisms and error bound estimations in a prototype system called ApproxHadoop. Our evaluation uses MapReduce applications from different domains, including data analytics, scientific computing, video encoding, and machine learning.

1. Introduction

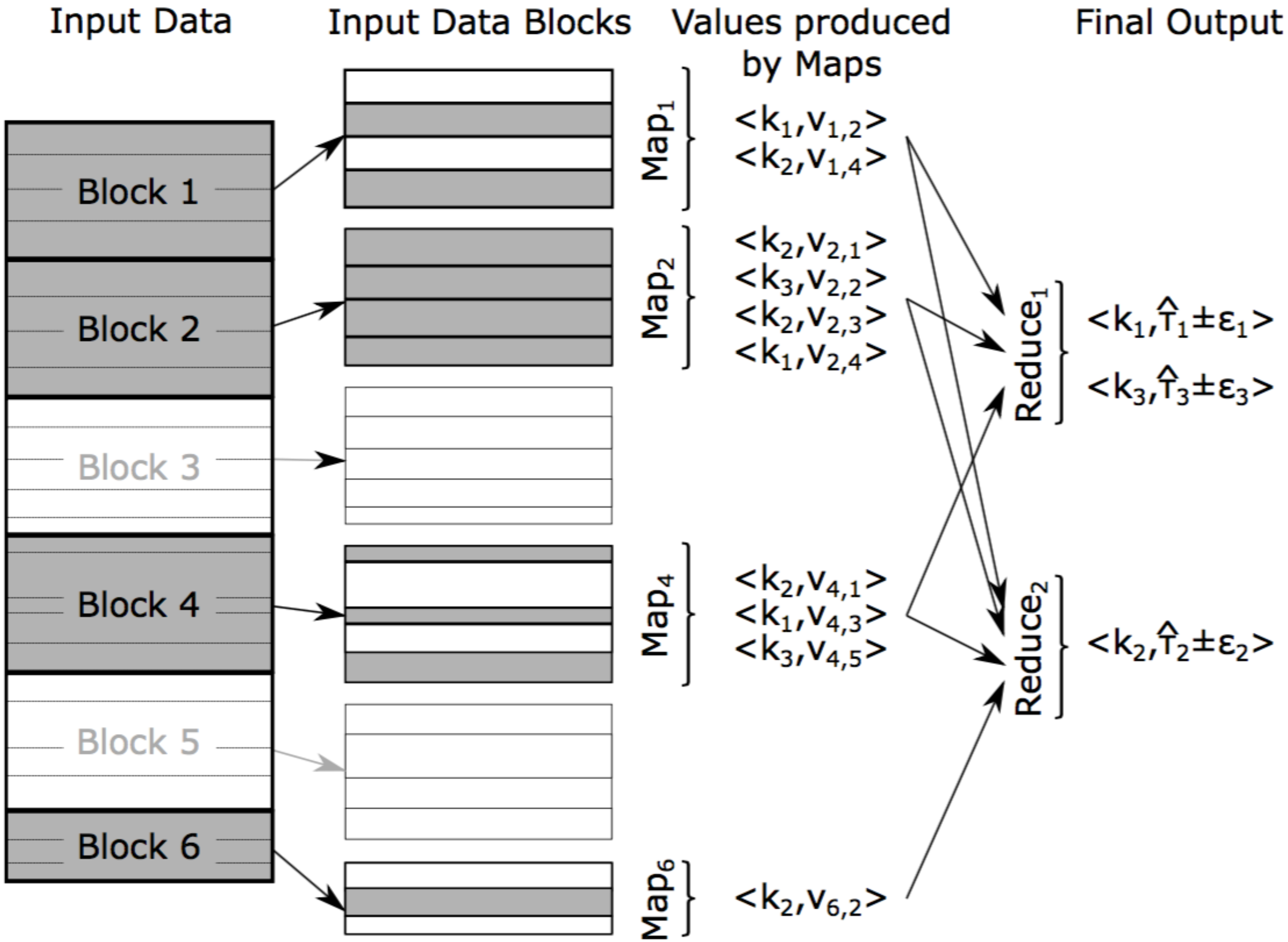
Motivation. Despite the enormous computing capacity that has become available, large-scale applications such as data analytics and scientific computing continue to exceed available resources. Furthermore, they consume significant amounts of time and energy. Thus, approximate computing has and continues to garner significant attention for reducing the resource requirements, computation time, and/or energy consumption of large-scale computing (*e.g.*, [5, 6, 10, 17, 38]). Many classes of applications are amenable to approximation, including data analytics, machine learning, Monte Carlo computations, and image/audio/video processing [4, 14, 25, 30, 41]. As a concrete example, Web site operators often want to know the popularity of individual Web pages, which can be computed from the access logs



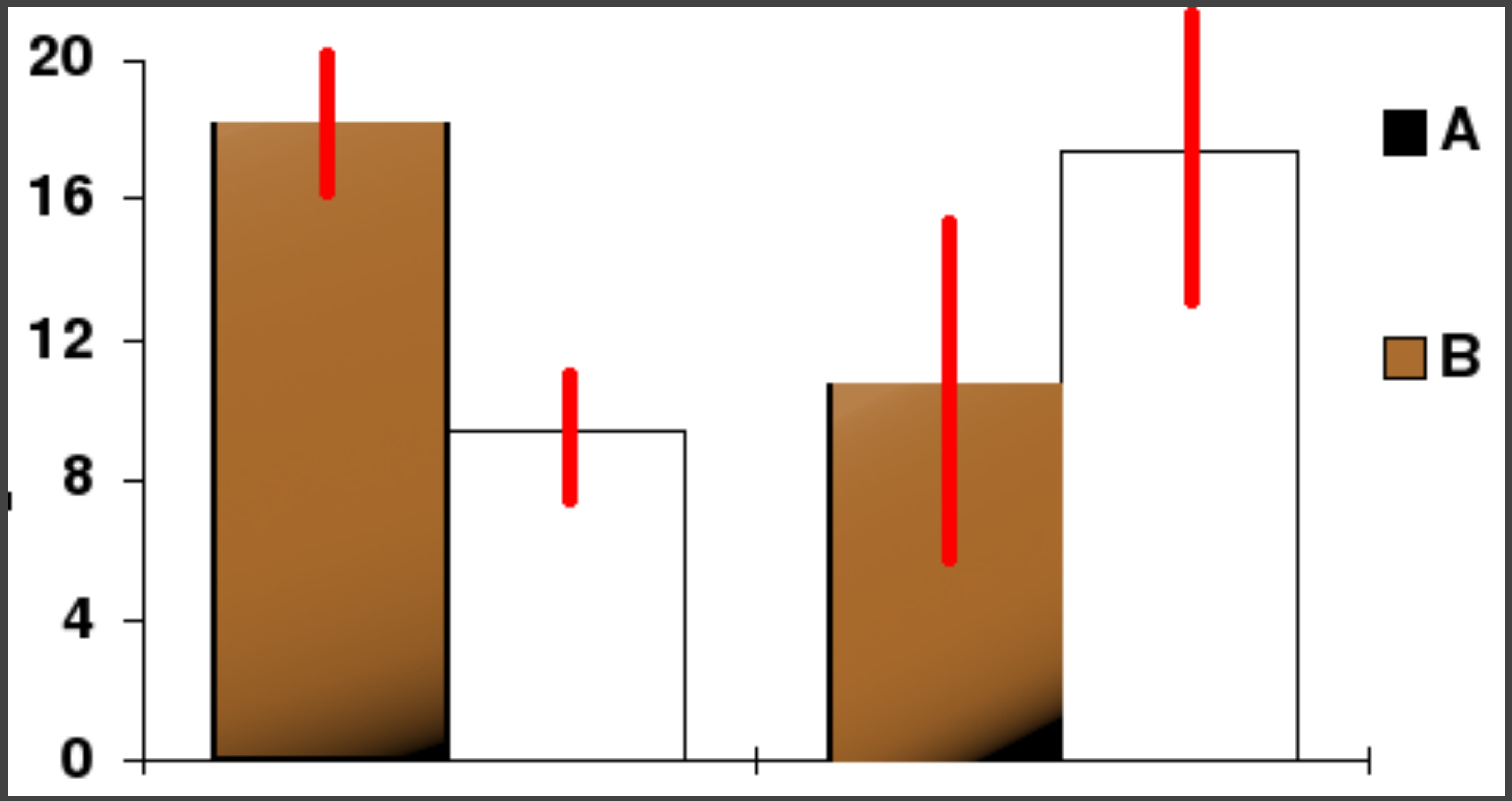
(a) Wikipedia Length
Number of articles
Page size (bytes)
Approximate
Precise
10^7
10^6
10^5
10^4
500
1000
1500
2000
2500
3000

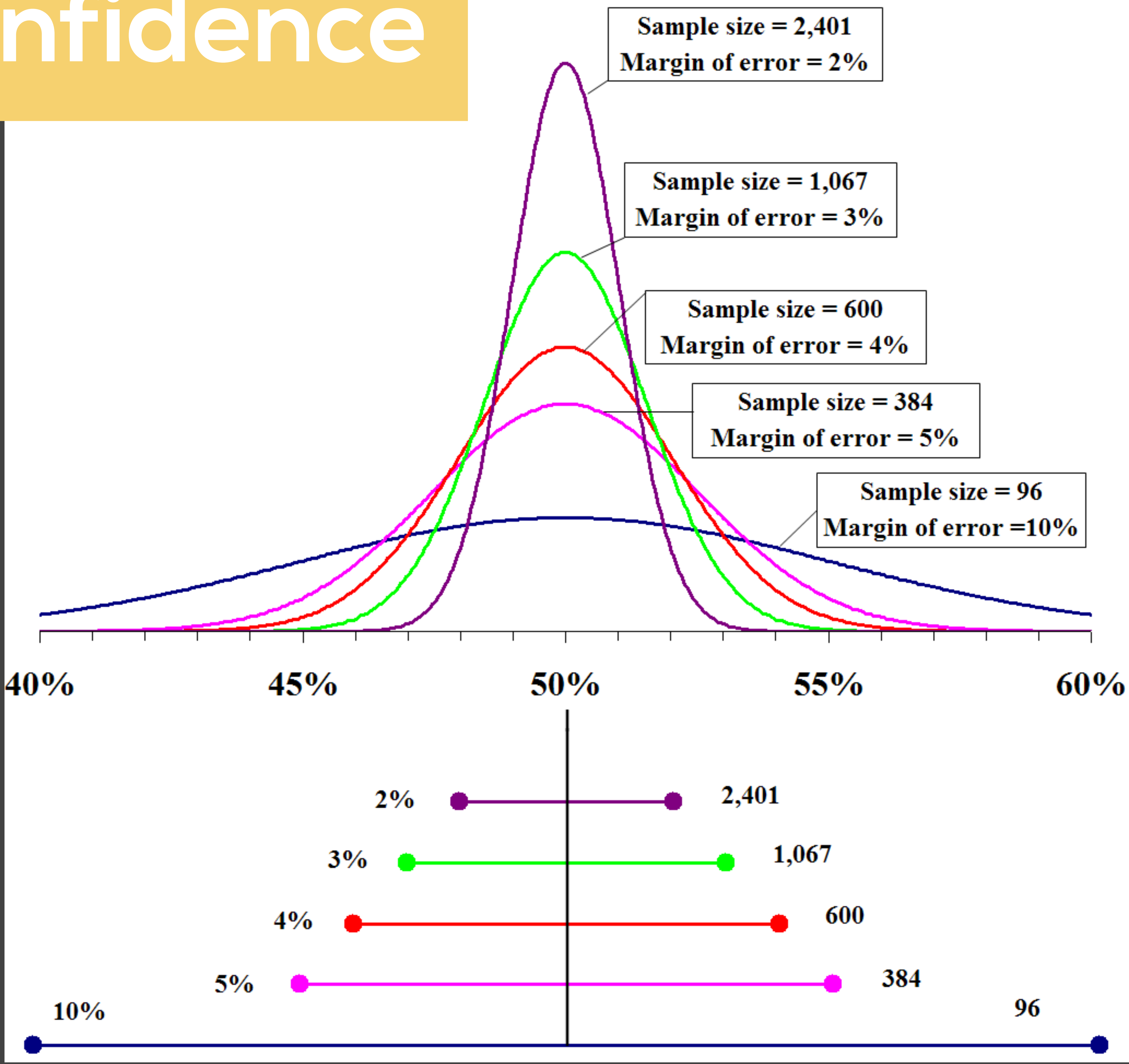
Query Response Time (seconds)
100000
10000
1000
100
10
1
Hive on Hadoop
Hive on Spark (without caching)
Hive on Spark (with caching)
BlinkDB (1% error)
2.5TB
7.5TB
Data Size (TB)

Sampling works!



Error bounds & confidence

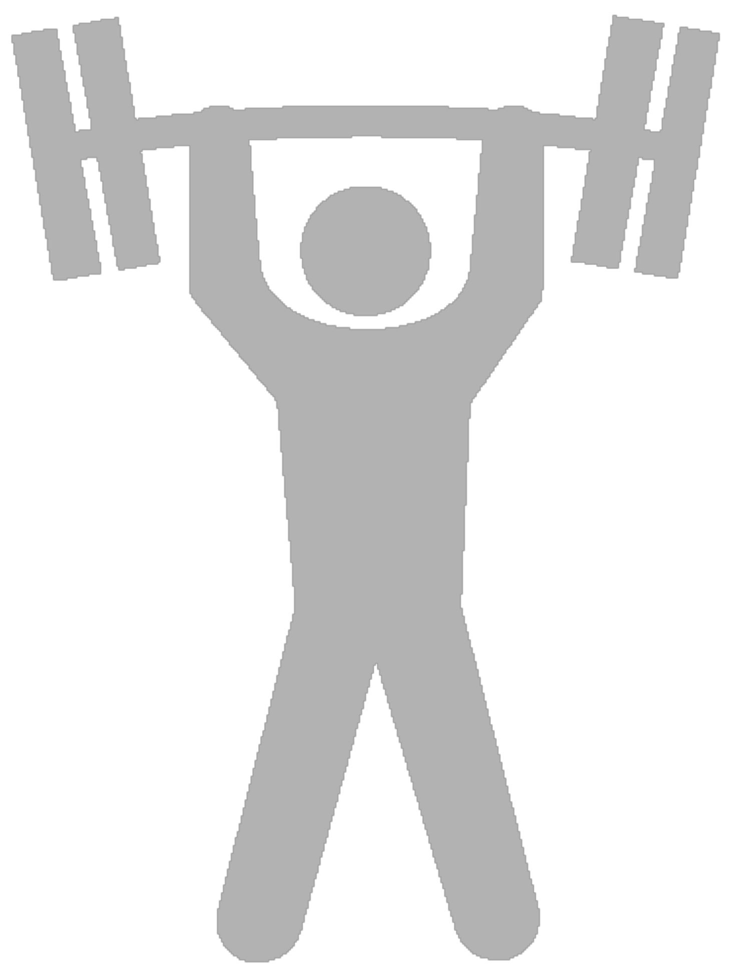




Don’t ask wasteful questions

A paper tour of

Rugged



Harvest, Yield, and Scalable Tolerant Systems

Foundation

Armando Fox
Stanford University
fox@cs.stanford.edu

Eric A. Brewer
University of California at Berkeley
brewer@cs.berkeley.edu

Abstract

The cost of reconciling consistency and state management with high availability is highly magnified by the unprecedented scale and robustness requirements of today's Internet applications. We propose two strategies for improving overall availability using simple mechanisms that scale over large applications whose output behavior tolerates graceful degradation. We characterize this degradation in terms of harvest *and* yield, *and map it directly onto engineering mechanisms that enhance availability by improving fault isolation, and in some cases also simplify programming. By collecting examples of related techniques in the literature and illustrating the surprising range of applications that can benefit from these approaches, we hope to motivate a broader research program in this area.*

degrading functionality rather than lack of availability of the service as a whole. The approaches were developed in the context of cluster computing, where it is well accepted [22] that one of the major challenges is the nontrivial software engineering required to automate partial-failure handling in order to keep system management tractable.

2. Related Work and the CAP Principle

In this discussion, *strong consistency* means single-copy ACID [13] consistency; by assumption a strongly-consistent system provides the ability to perform updates, otherwise discussing consistency is irrelevant. *High availability* is assumed to be provided through redundancy, e.g. data replication; data is considered highly available if a given consumer of the data can always reach *some* replica.

Ruggedness as availability

Strategies to ***enhance ruggedness*** *in the presence of failures*

Better way to ***think about system availability***

Yield: fraction of **answered queries**

Harvest: fraction of the **complete result**

Yield as response ruggedness

*Close to uptime (% requests answered successfully) but more useful because it directly **maps to user experience***

---

*Failure during high & low traffic **generates different yields.** Uptime misses this*

Focus on yield rather than uptime

Harvest as quality of response

$$harvest = \frac{data\ available}{total\ data}$$

66% harvest

From Coda Hale's "You can't sacrifice partition tolerance"

#1: Probabilistic Availability

Graceful *harvest* ***degradation under faults***

Randomness *to make the worst-case & average-case the same*

Replication of high-priority data *for greater harvest control*

Degrading ***results based on client*** *capability*

#2 Decomposition & Orthogonality

Decomposing into ***subsystems*** *independently intolerant to harvest degradation (fail by reducing yield). But* ***app can continue if they fail*** 💪

Only provide ***strong consistency for the subsystems*** *that need it*

Orthogonal mechanisms (state vs functionality)

Lineage-driven Fault Injection

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joshua Rosen
UC Berkeley
rosenville@gmail.com

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

ABSTRACT

Failure is always an option; in large-scale data management systems, it is practically a certainty. Fault-tolerant protocols and components are notoriously difficult to implement and debug. Worse still, choosing existing fault-tolerance mechanisms and integrating them correctly into complex systems remains an art form, and programmers have few tools to assist them.

We propose a novel approach for discovering bugs in fault-tolerant data management systems: *lineage-driven fault injection*. A lineage-driven fault injector reasons *backwards* from correct system outcomes to determine whether failures in the execution could have prevented the outcome. We present MOLLY, a prototype of lineage-driven fault injection that exploits a novel combination of data lineage techniques from the database literature and state-of-the-art satisfiability testing. If fault-tolerance bugs exist for a particular configuration, MOLLY finds them rapidly, in many cases using an order of magnitude fewer executions than random fault injection. Otherwise, MOLLY certifies that the code is bug-free for that configuration.

enriching new system architectures with well-understood fault tolerance mechanisms and henceforth assuming that failures will not affect system outcomes. Unfortunately, fault-tolerance is a *global* property of entire systems, and guarantees about the behavior of individual components do not necessarily hold under composition. It is difficult to design and reason about the fault-tolerance of individual components, and often equally difficult to assemble a fault-tolerant system even when given fault-tolerant components, as witnessed by recent data management system failures [16, 57] and bugs [36, 49].

Top-down testing approaches—
behavior of complex systems—are
fication of individual components.
is the dominant top-down approac
and dependability communities.
vestment, fault injection can quickl
by a small number of independent
jection is poorly suited to discove
volving complex combinations of
faults (e.g., a network partition fol

Ruggedness via verification

Formal Methods

HUMAN ASSISTED PROOFS

SAFETY CRITICAL (*TLA+, COQ, ISABELLE*)

MODEL CHECKING

PROPERTIES + TRANSITIONS (*SPIN, TLA+*)

LIGHTWEIGHT FM

BEST OF BOTH WORLDS (*ALLOY, SAT*)

Testing

TOP-DOWN

FAULT INJECTORS, INPUT GENERATORS

BOTTOM-UP

LINEAGE DRIVEN FAULT INJECTORS

WHITE / BLACK BOX

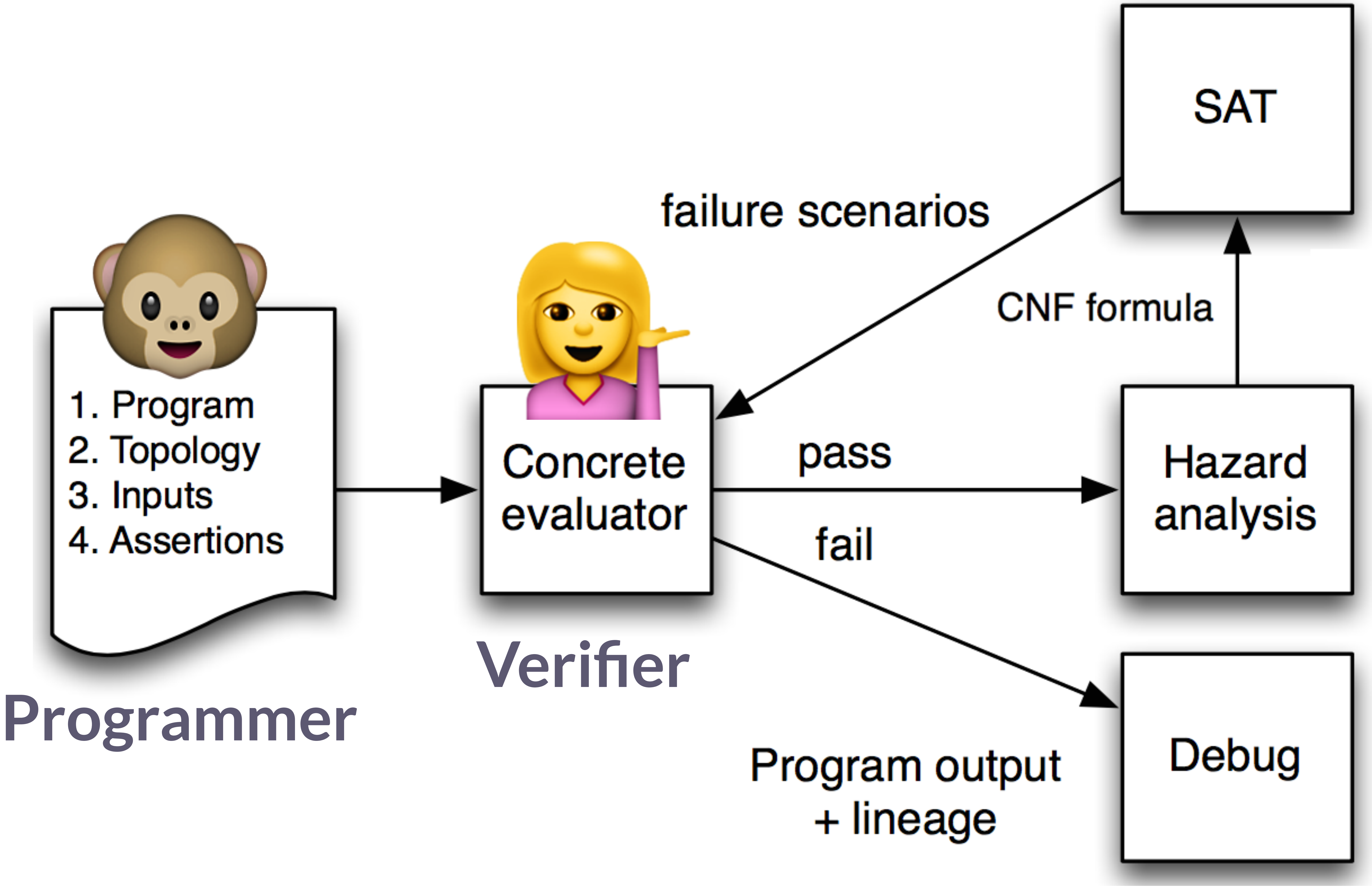
WE KNOW (OR NOT) ABOUT THE SYSTEM

MOLLY: Lineage Driven Fault Injection

Reasons backwards from correct system outcomes & determines if a failure could have prevented it

MOLLY only injects the failures ***it can prove*** *might affect an outcome*

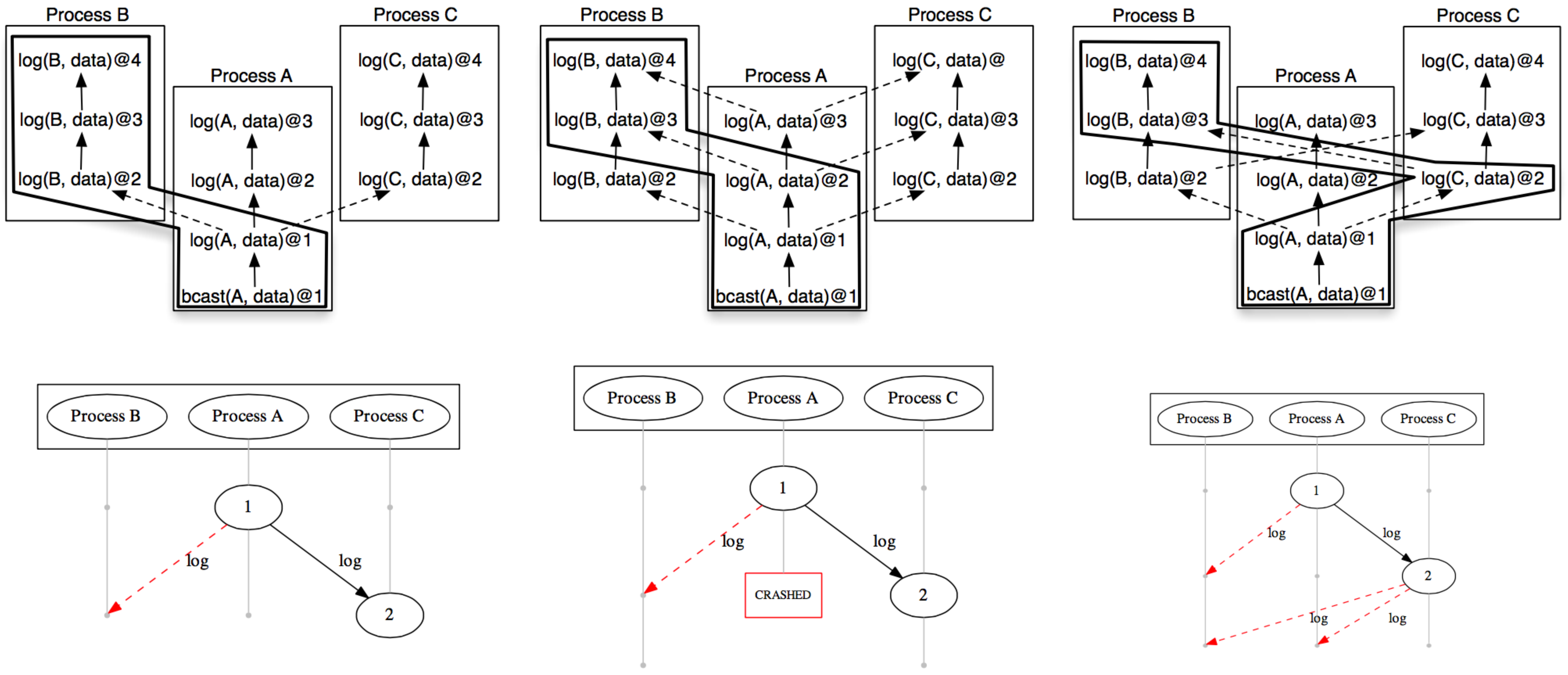
Ruggedness with MOLLY



"Without **explicitly forcing a system to fail**, you have **no confidence** that it will **operate correctly** in failure modes"

Caitie McCaffrey's pearls of wisdom

MOLLY helps us undestand failure



a: Round 1: **simple-deliv**

b: Round 2: **retry-deliv**

c: Round 5: **classic-deliv**

“Presents a middle ground between pragmatism and formalism, dictated by the importance of verifying fault tolerance in spite of the complexity of the space of faults”

Now let's

.Wrap things

tl;dr - foundations

Agile

Designing for change is designing for success

Lean

A scalable system may not be a lean system

---

*Pursuing **scalability out of context can be COSTly***

Rugged

*Think about availability in terms of **yield** and **harvest***

Graceful degradation is a design outcome

tl;dr - Frontiers

Agile

State can be challenging

Saving state in shared memory allows us to restart DB processes faster

Lean

*Asking the **wrong question is wasteful***

Think about what is truly needed

*Use **approximations***

Rugged

***Reasoning backwards** from correct system output helps us **determine the execution failures** that prevent it from happening*

Papers are a lot of fun!

github.com/Randommood/GotoLondon2015

Join your local PWL and
read The Morning Paper!

DRANKS!