Architecture, Algorithms and Methodology of Ultra Large Systems

(With a special look at storage concepts)

Prof. Walter Kriha

Computer Science and Media Faculty

HdM Stuttgart

www.kriha.org

Motivation

You may never need to grow as big as an ULS but some parts of your architecture will probably experience scalability problems one day.

By studying ULS you will be able to read the signs and know what to do.

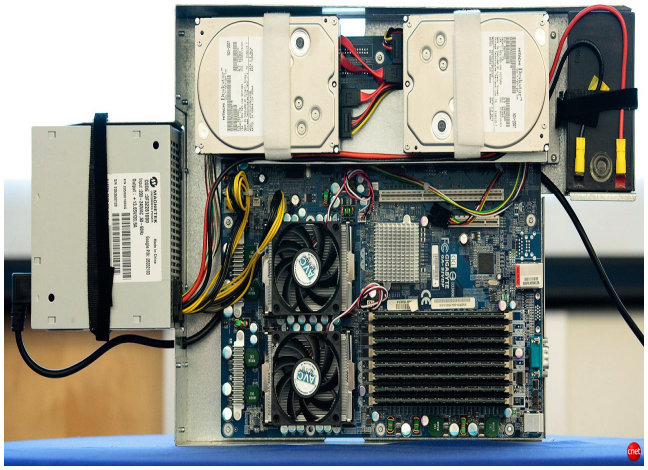
Topics

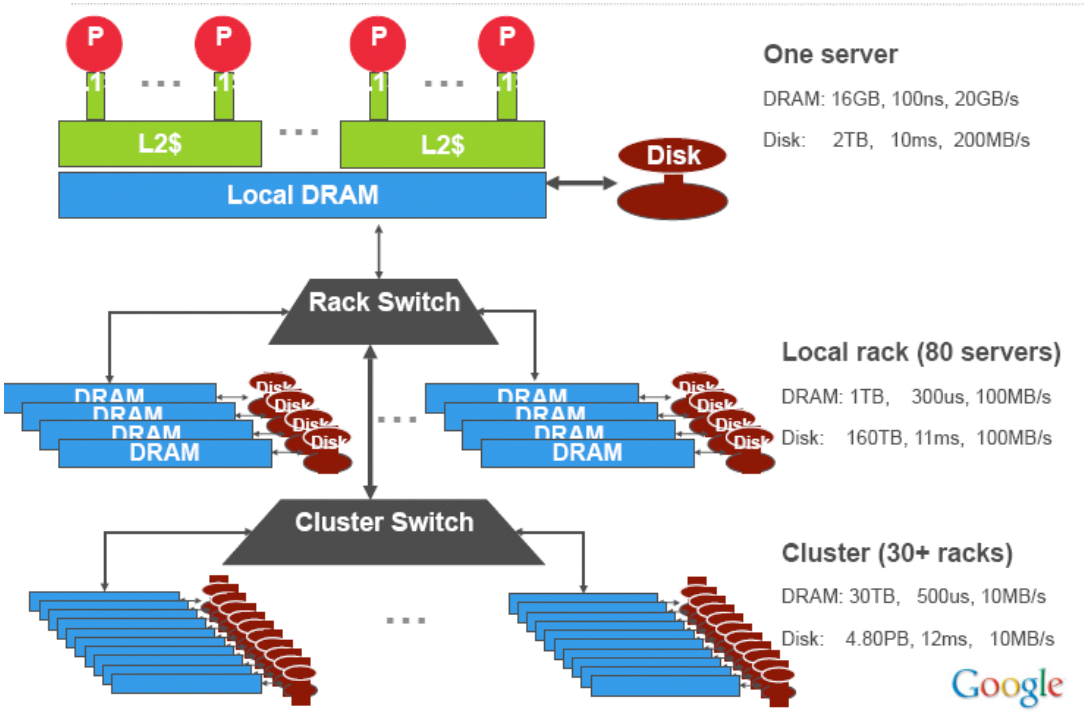
- How big are they? A look at Facebook and co.
- Methodology, architectures, concepts and algorithms
- Storage: the move towards NOSQL, Grid Storage Scalability
- Current trends: automation, data-flow processing, transparency
- The Future: ambient clouds and scaling beyond our imagination
- Resources

How big are they?

- 20000 to over 2 Mio. servers
- Worldwide replicated data centers, 24/7 operation
- Petabyte of stored data to be analyzed every day
- Terabyte of new data/media every day
- 400.000.000 (twitter) to over 1.000.000.000 (facebook) users with growth rates beyond 300.000 new users every day (farmville: 1 Mio after 4 days, 10 Mio. after 10)
- Billions of requests per API every day

Google Datacenter Architecture





Google cluster, rack and blades, after Jeff Dean, Designs, Lessons and Advice from Building Large Distributed Systems

ULS Methodology

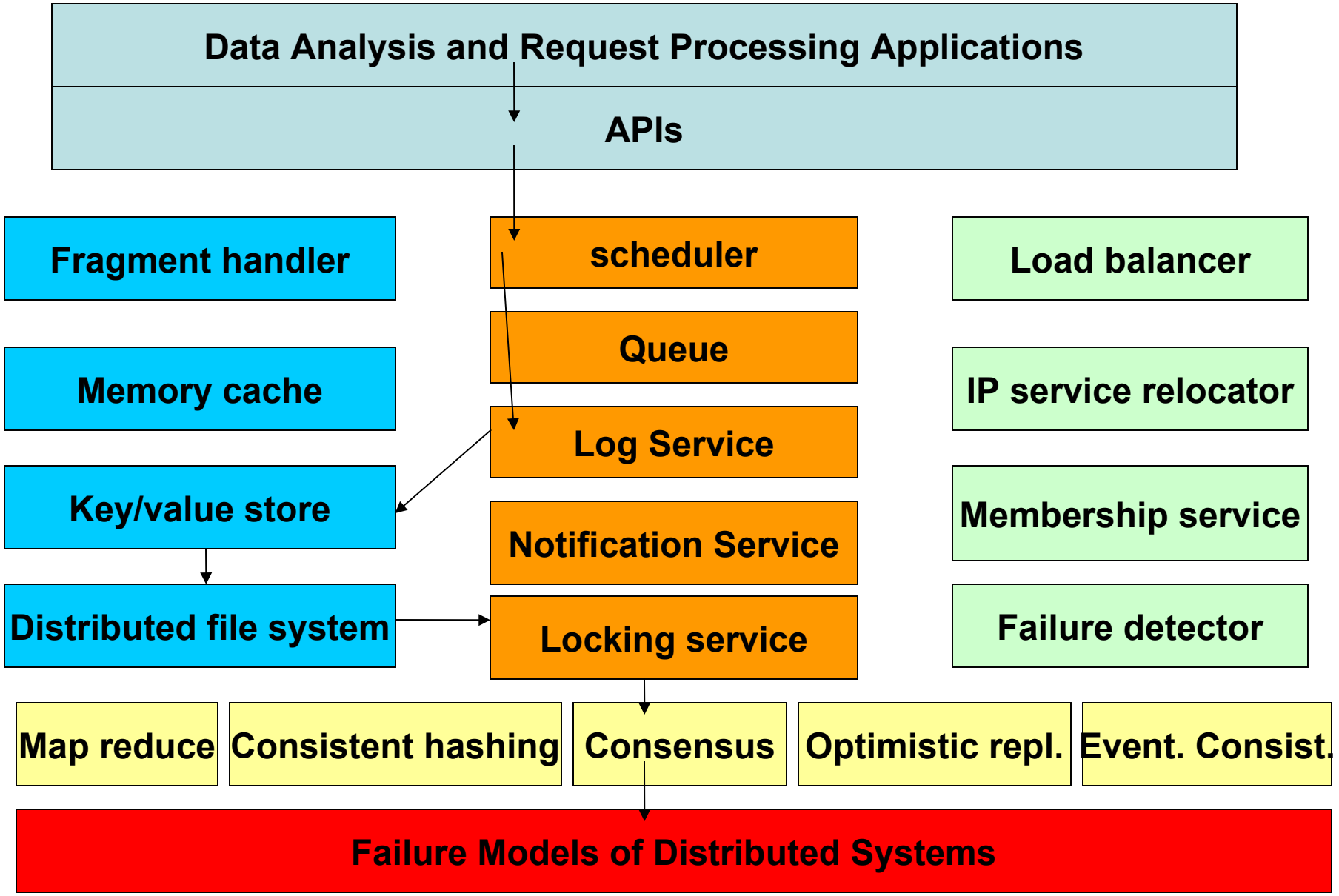
while (true)
{
identify_and_fix_bottlenecks();
drink();
sleep();
notice_new_bottleneck();
}
This loop runs many times a day.
(Todd Hoff, youtube article)

Research: can we evolve systems instead? Does scaling always mean new system architecture (Think about the financials too!)? Does capacity planning work at the growth rates given above?

ULS Architecture

- Horizontally scaled clusters of blades running LAMP, Ruby etc.
- 10s – 100s of Terabyte memcached caches (twitter keeps all messages in RAM cache)
- Asynchronous message processing via queues and pre-processing
- Sharded DBs or distributed key/value stores
- Globally distributed data centers with async. replication.
- Dedicated components for parallel scheduling, parallel processing, locking, file serving using small multi-cast clusters for control

ULS Architecture Components



Wrong Optimization: MySpace

Membership Milestones:
- 500,000 Users: A Simple Architecture Stumbles

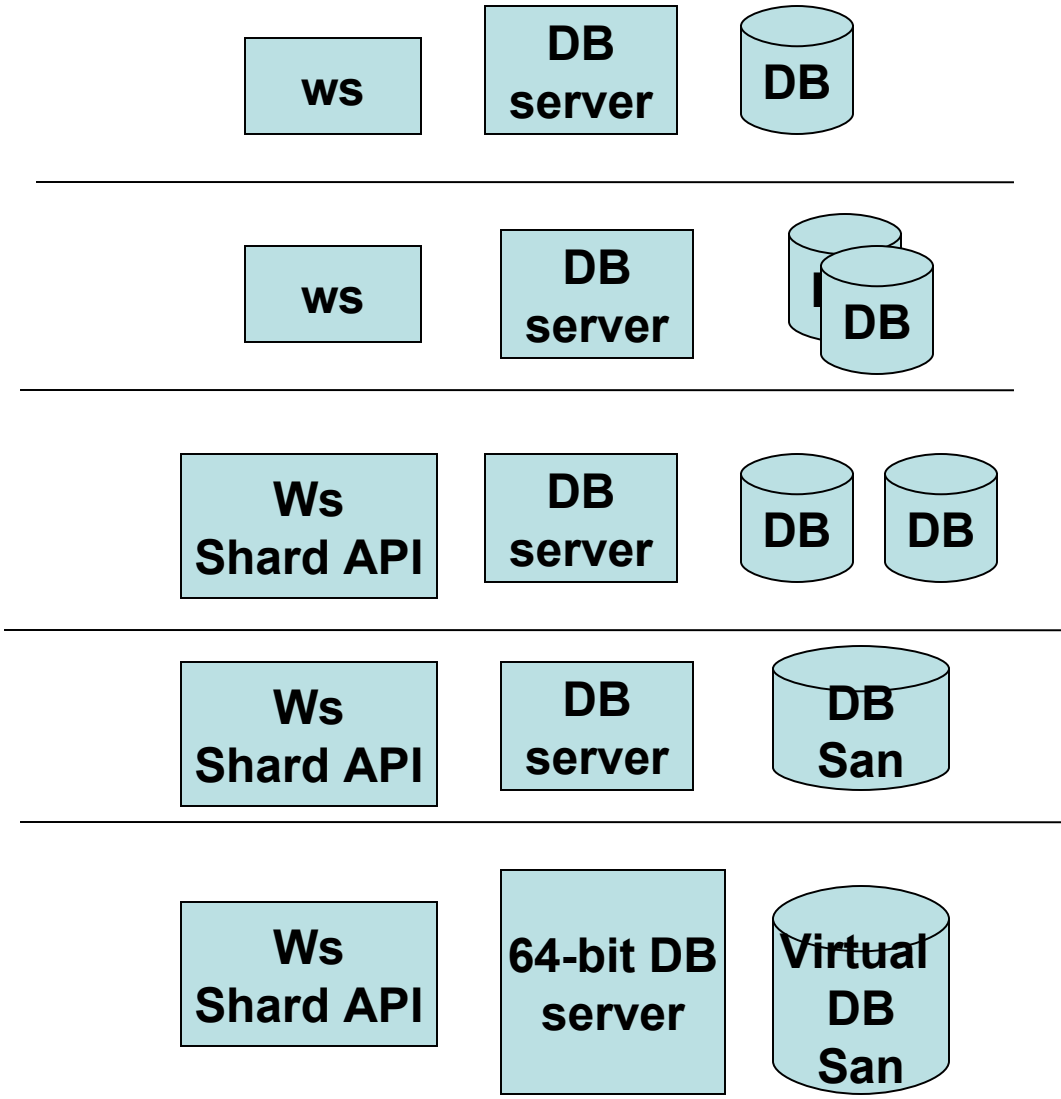
-1 Million Users:Vertical Partitioning Solves Scalability Woes

-- 3 Million Users: Scale-Out Wins Over Scale-Up

-- 9 Million Users: Site Migrates to ASP.NET, Adds Virtual Storage

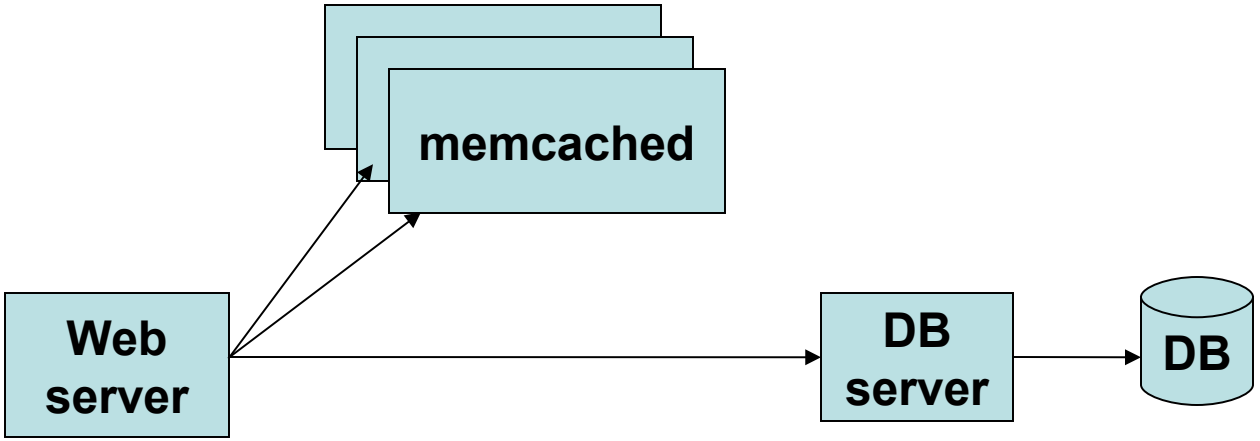
-- 26 Million Users: MySpace Embraces 64-Bit Technology

(after Todd Hoff's Myspace article)



What is MISSING????

End-to-End Optimization



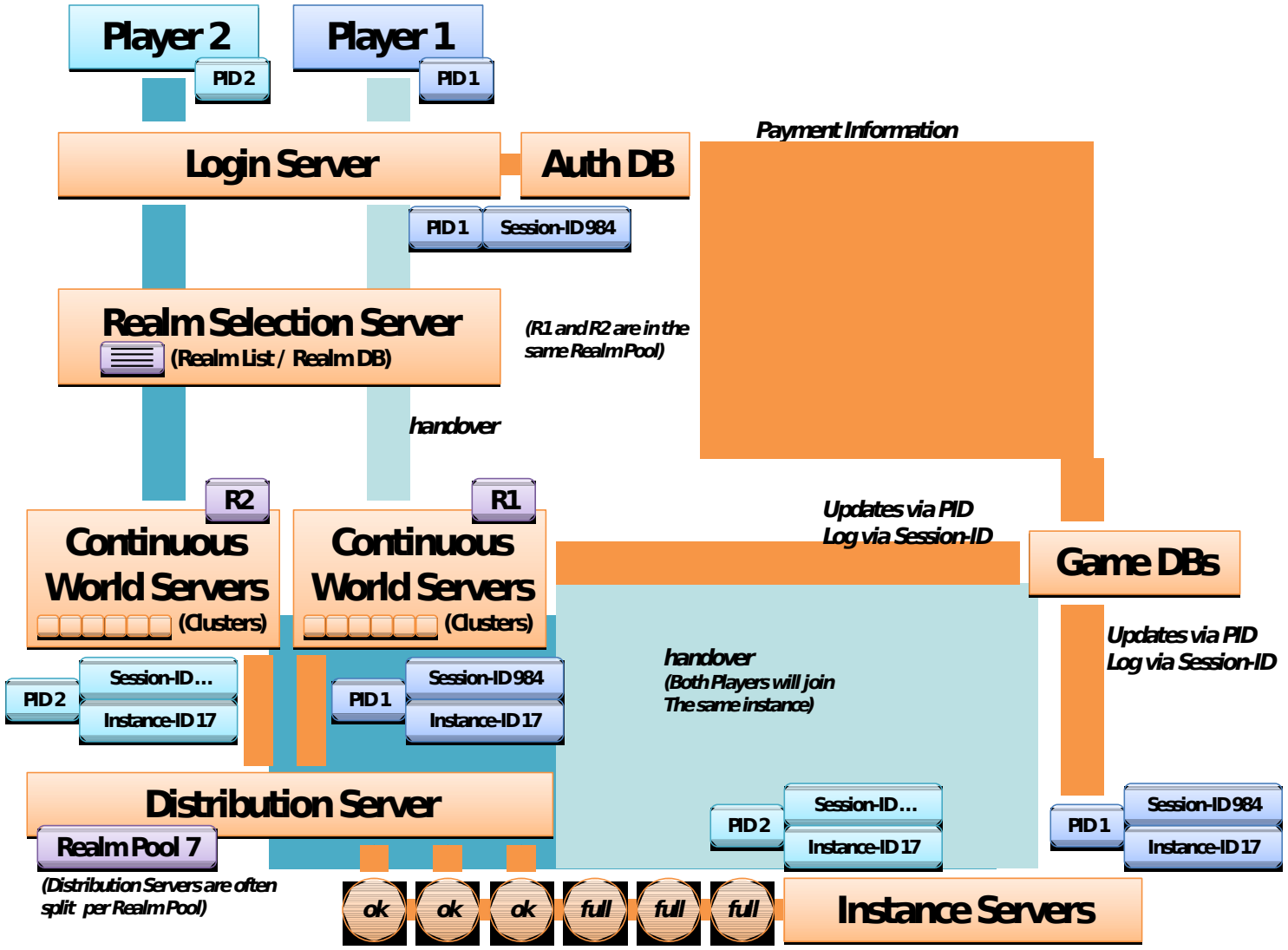
Adding a cache changes read/write ratios considerably and make many DB optimizations potentially redundant.

The „google principle“: don‘t spend much effort on optimizing one binary. Set it up and go for scalability right away (try to solve the problem with adding resources. If this does not work out easily, re-think your design.

ULS Concepts

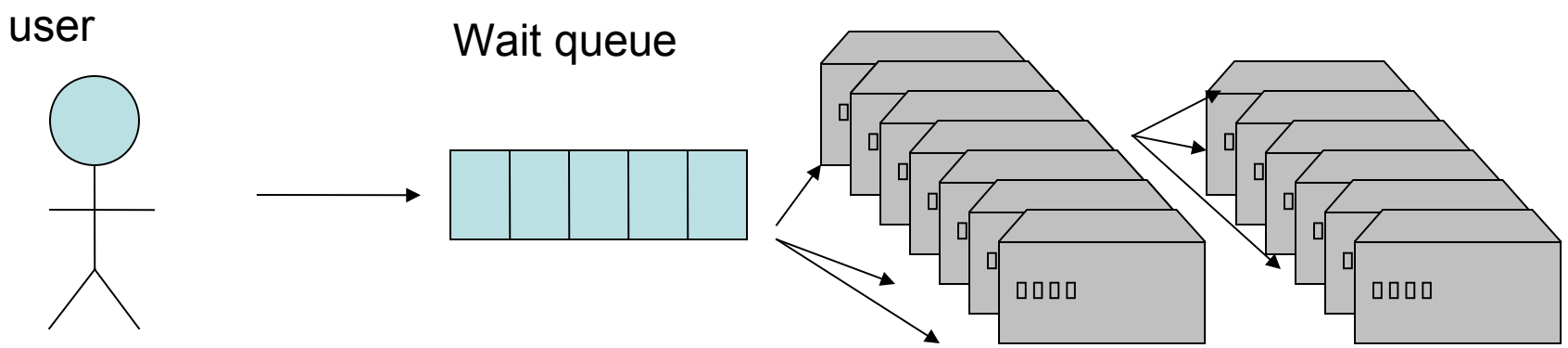
- **Partitioning** of data, services and users
- Avoid locking and state in requests. **Shared nothing** architecture
- **De-normalize data** for fast access
- Always measure **read/write ratios** across **sequential** vs. **random** access dimensions
- Avoid deletes (garbage collection). Use **versions**, timestamps.
- Match **applications and infrastructures** (but beware for requirment changes)
- Avoid **joins and complex queries**, use application level processing instead (what did you learn at DB-school???)
- Multi-level **caching**, multi-get methods (learn from queuing theory)
- Strict **feature and service management** (SLA, turn-off, content distribution, user management, connection management)
- Exact **measurements** of infrastructure
- Instrumented code for permanent real-time monitoring and alerting
- **Best-of-breed** thinking
- Multi-paradigm, **concept based** selection of languages and technology
- **Open source only** (for license and instrumentations reasons)
- Avoid **transactions**
- Use dynamic languages for frontend apps to get fast development cycles

Example: Partitioning in MMOGs



From Andreas Stiegler, MMOG infrastructures

Example: Service Management



1. SLA for every service with 99.99 percentile guaranteed
2. All services degradable. All services have about the same service time.
3. Wait time measured to avoid dead request processing
4. Balancing and failover using group communication
5. Services split into synchronous and asynchronous parts

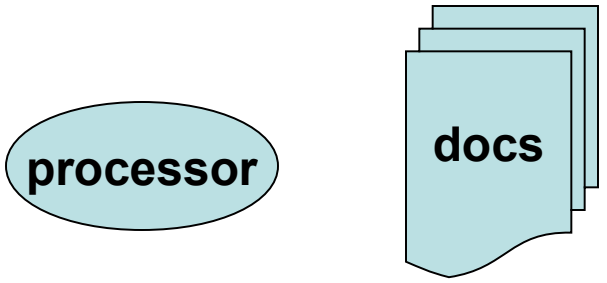
Scalable Algorithms

- **Asynchronous** I/O to avoid context switching of large numbers of threads (Nginx)
- **Non-locking** compare-and-swap, optimistic locking, software-transactional memory
- Algorithms as **functions** to avoid locking and side-effects (Erlang modules for DBs, instant messaging, DHTs)
- **Consistent hashing** to allow stable partitioning of data during re-configuration (Dynamo, Scalaris)
- **Paxos** for consistent, synchronous replication (Google chubby)
- **Eventual consistency** for fast replication (vector clocks to check for „read-your-own-writes“ etc. (Amazon Dynamo)
- **Map/reduce** for parallel processing of data (everybody)

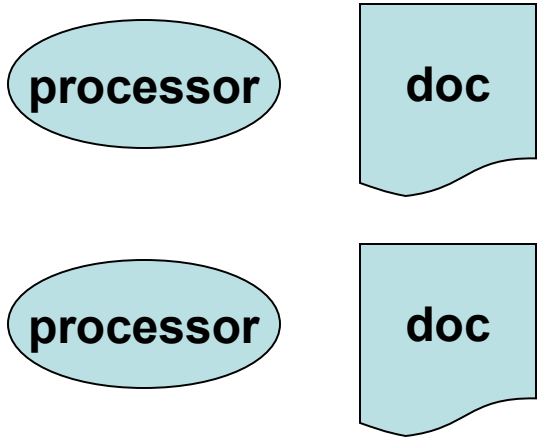
Watch out for the effects on application programmers! Not everybody wants to explicitly deal with inconsistencies…

Scalable Algorithm Example: map/reduce

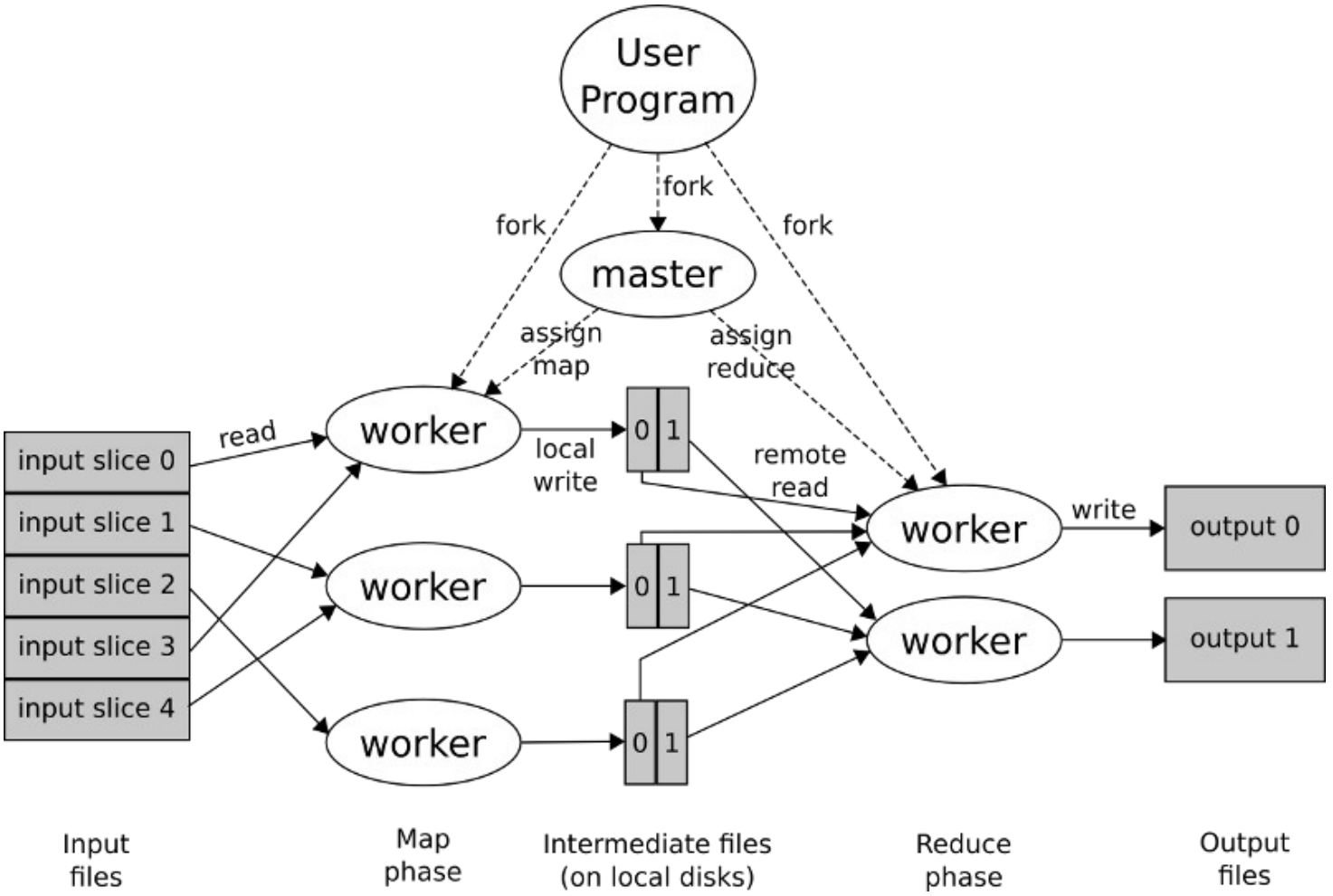
```
For (i=0;i<AllDocuments;i++)
    Document=nextDocument();
    Result=Process(Document)
    Write(Result)
```




```
Map(Documents, ProcessingFunction)
  For (i=0;i<AllDocuments;i++)
     New Thread(Document,
    ProcessingFunction)
```

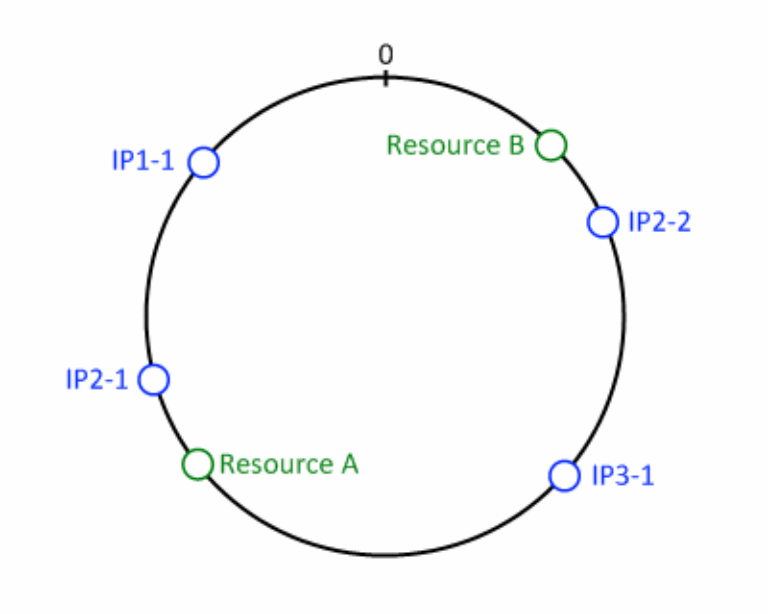


Infrastructure for Map/Reduce Jobs



From Jeff Dean/Sanjay Ghemawat, needs to deal with stragglers etc.

Consistent Hashing



A simple hash function like

X -> ax + b (mod P)

with P being the number of nodes would cause a “thundering herd” problem with all cache entries suddenly being invalid. Mapping both resources and machines into a ring assigns a certain range of keys per machine. Amazon’s Dynamo has even more indirections.

Secrets of Ultra Large Scale Systems?

- Don‘t be afraid of changing languages or code
- Don‘t be afraid of using things like PHP or MySQL
- Define your own processes
- Measure and trace ruthlessly
- Define APIs but don‘t overrate their effect on code stability
- Don‘t overrate Standards
- Do not use binary only components
- Get help by using a CDN if necessary
- Don‘t be too proud to install a degraded mode to deal with overload
- Use languages which allow you to change quickly, especially the front-end features

Storage Concepts

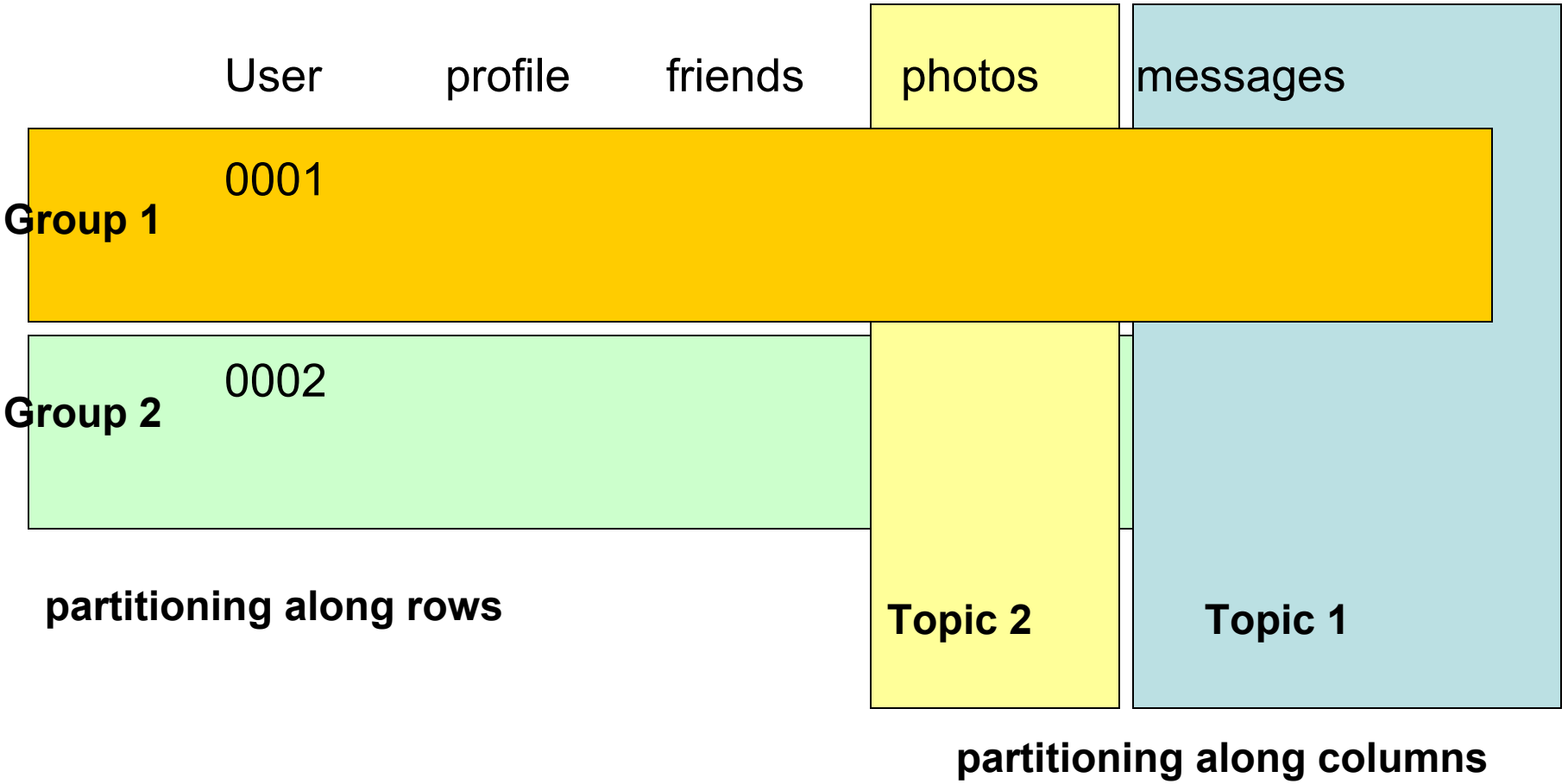
- From sharded databases to NoSQL stores
- grid file system scalability

Database Sharding

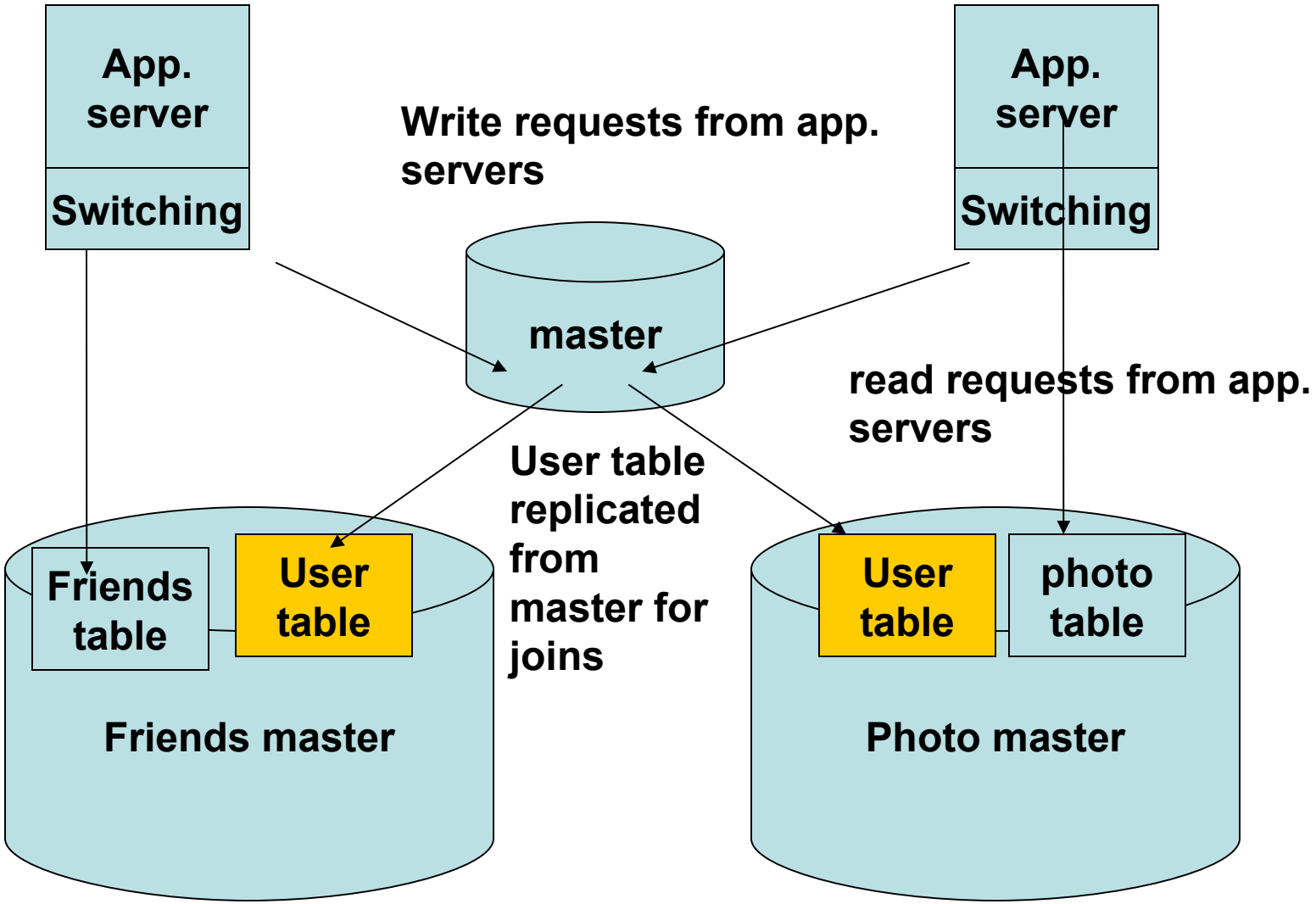
1. **Choose a data type for partitioning (row or column)**
2. **Decide on lookup strategy (algorithm or master table)**
3. **Deal with changes in the distribution of the data type over time**
4. **Adapt application to use meta-data for lookup (API)**
5. **Avoid joins and complex queries by using a powerful caching architecture which stores complex objects and not only row data**
6. **Allow for changes in the sharding logic.**

The combination of MySQL and Memcached is still able to run many very large sites (see Persyn for sharding logic)

Horizontal and Vertical DB Sharding



Data Duplication across Shards



Not shown: read slaves per master

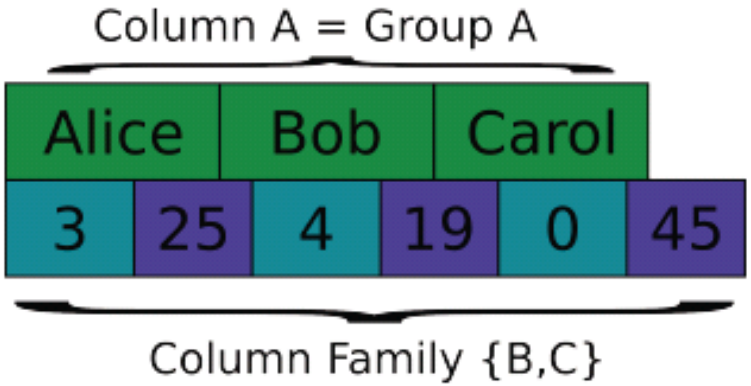
The Move to NoSQL Datastores

1. **No ACID properties needed in many social apps**
2. **No hard schema wanted or needed**
3. **Lack of SQL skills**
4. **No complex SQL queries possible (performance).**
5. **No joins possible (sharding)**
6. **Single resource type resources are hard to scale, sometimes doing sorts in the application scales better**
7. **Mostly object blobs needed to allow fast code changes**
8. **Most data held in cache anyway, DB used as backup only**

Why then are we manually scaling with sharding a SQL DB and using memcached? Why no use one system with some simple query functions, caching and automatic scaling?

NoSQL Column Stores (Bigtable, HBASE, Cassandra etc.)

Columnar Storage with Locality Groups

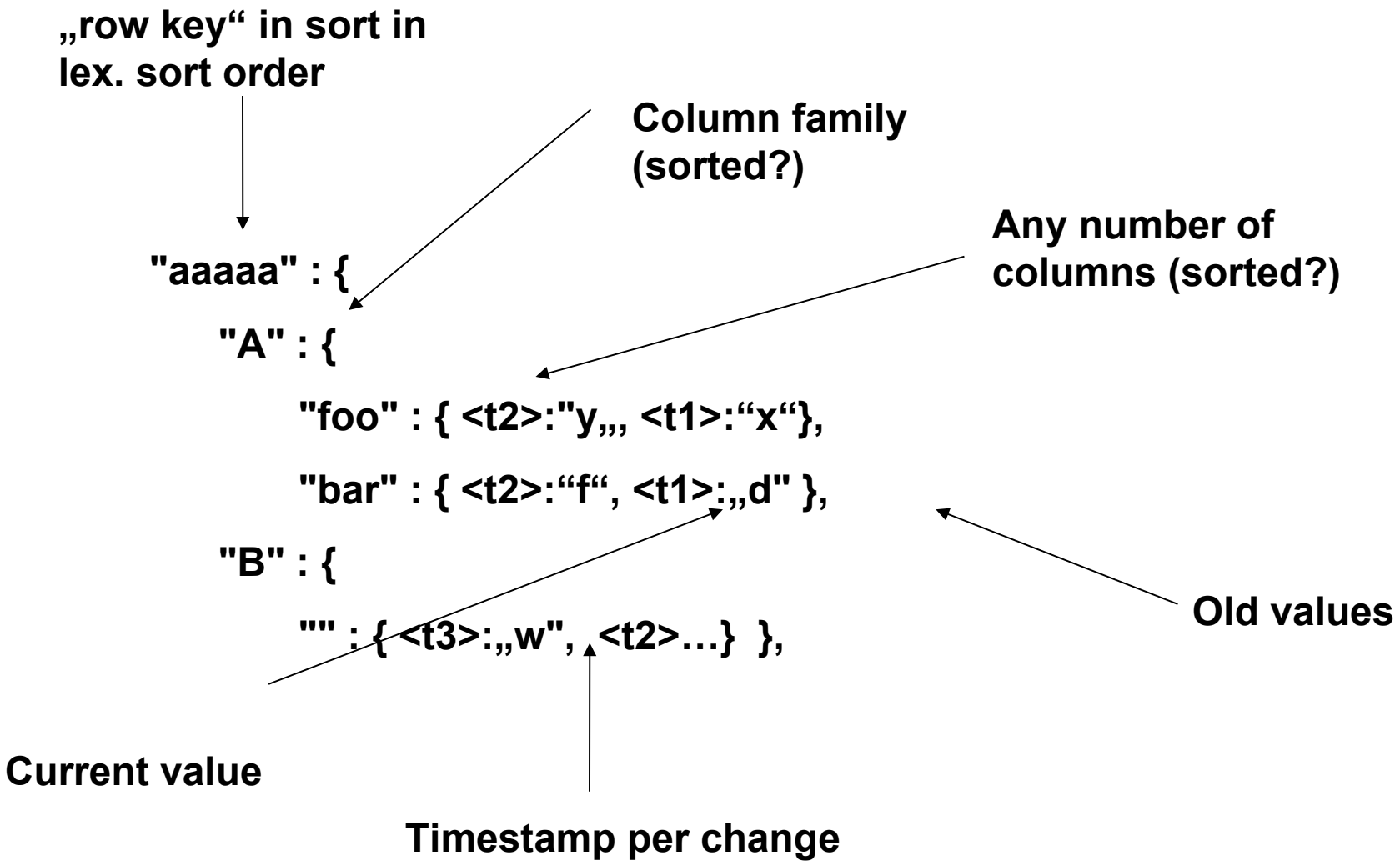


- Columns are organized into families ("locality groups")
- Benefits of row-based layout within a group.
- Benefits of column-based - don't have to read groups you don't care about.

cloudera

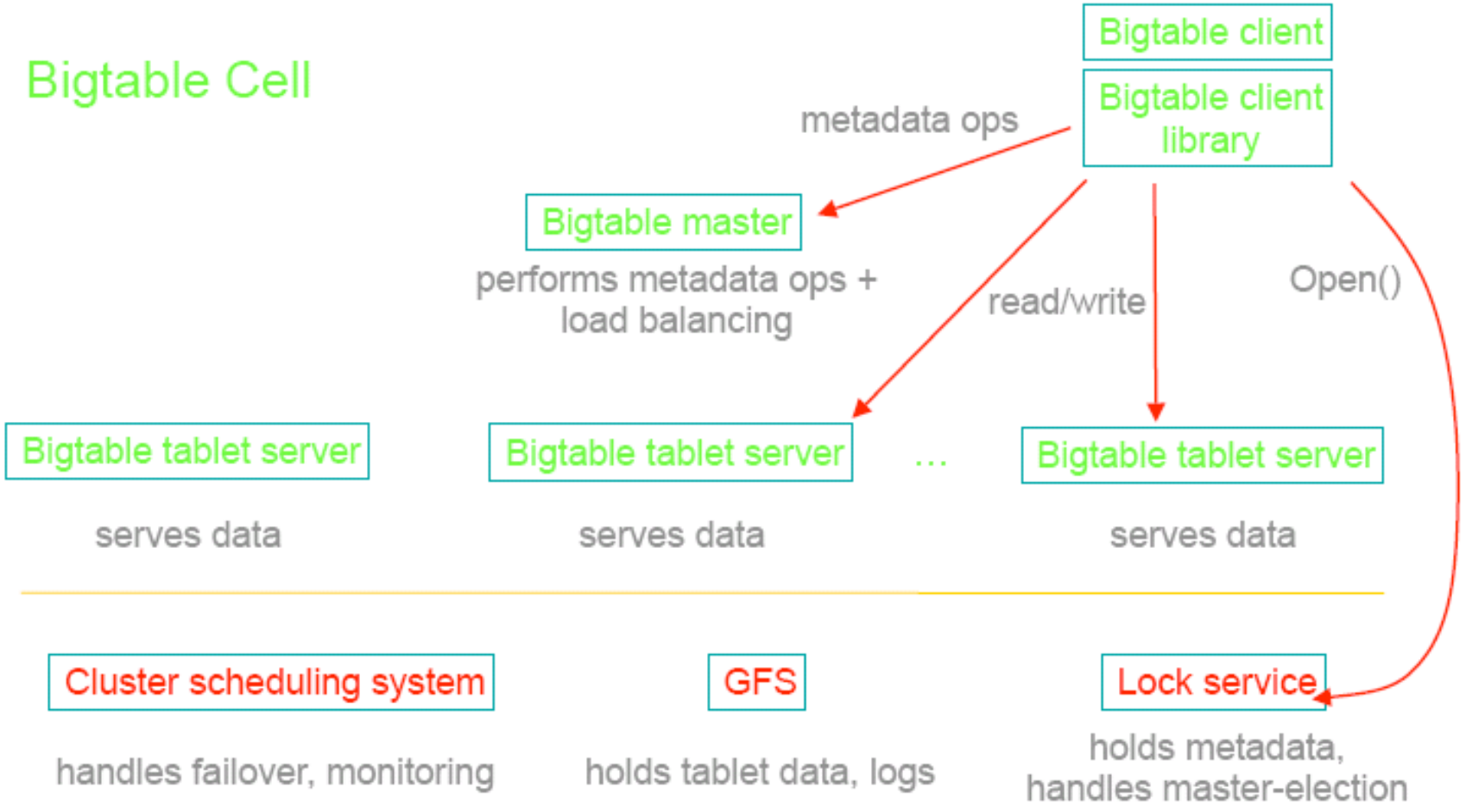
(after Lipcon)

HBASE Column Store



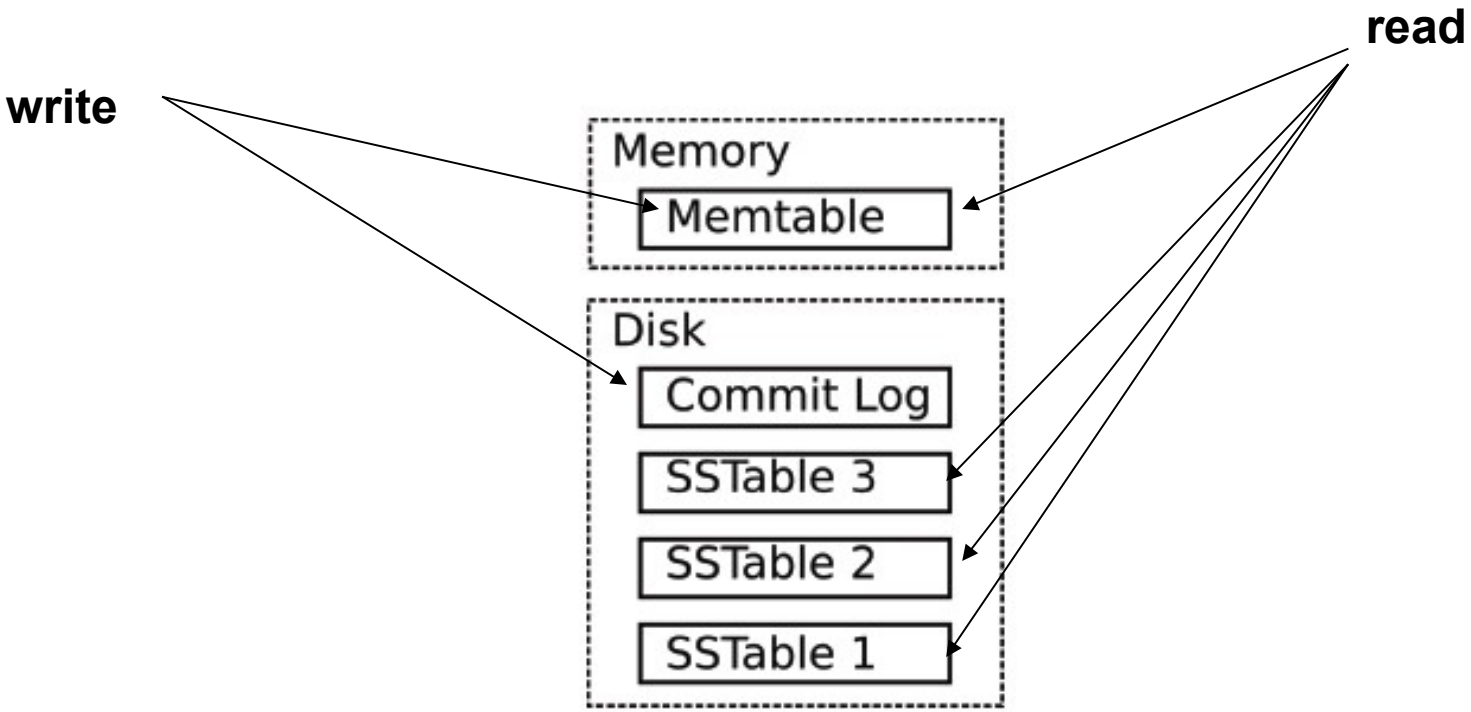
Loosely after: [Wilson]

BigTable System Structure



Google

Log Structured Merge Trees



Memtable flush: Memtable into SSTable4

Compaction: SSTables 1,2,3 into new SSTable 1‘

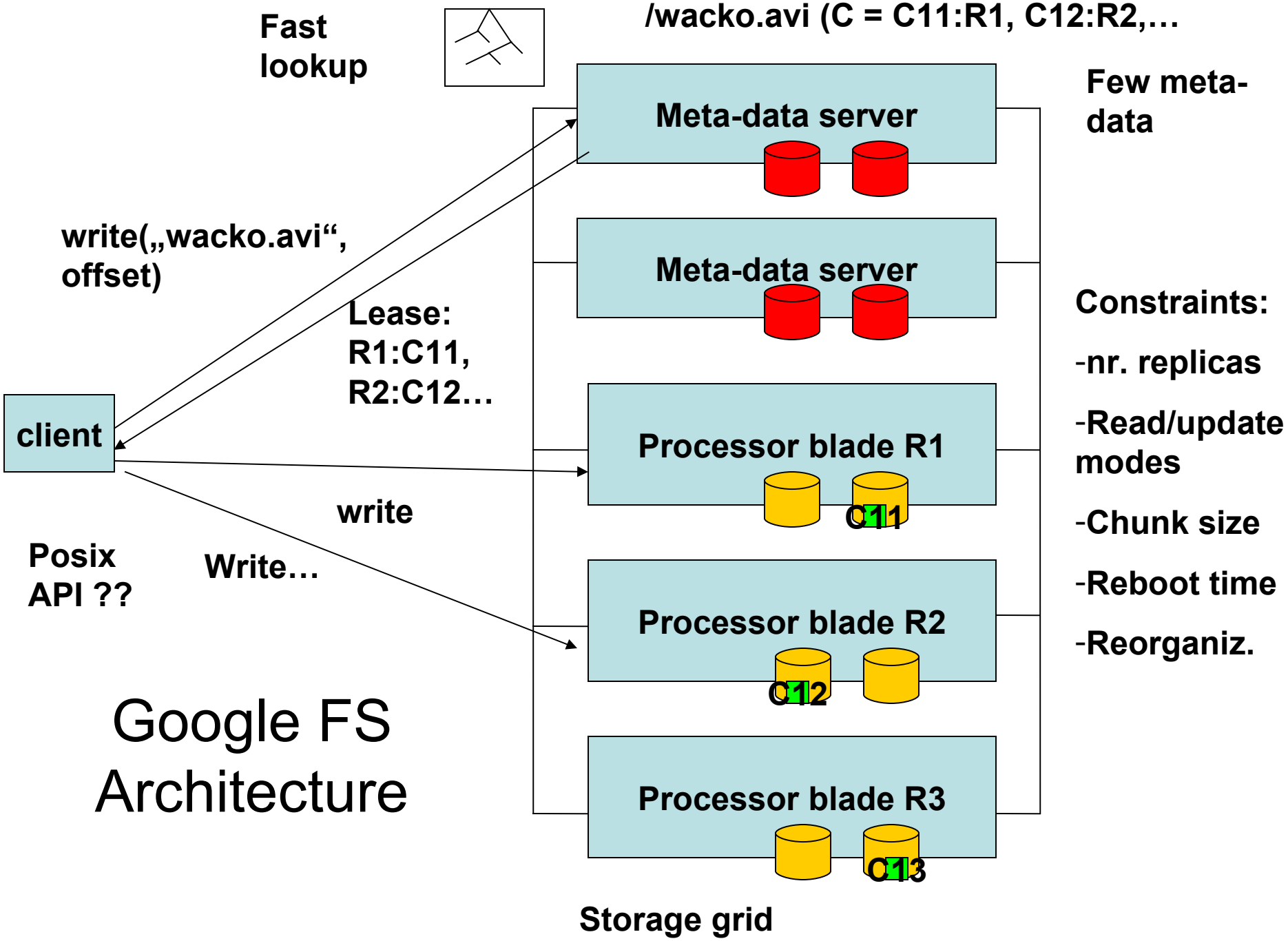
LSM turn random writes into sequential writes (after Lipcon)

cloudera

Distributed (Grid) Filesystems

Google File System Design Ideas

- most files are read and written sequentially (bandwidth over latency)
- appending writes were frequent, random writes almost non-existent (no latency issues)
- huge chunk size minimizes meta-data
- only google controlled applications would use it and could be therefore co-developed. No strict Posix-compatibility needed
- 1000s of storage nodes should be supported
- Some inconsistencies tolerable (apps deal with it)
- No data loss allowed (chunk replication)
- No extra caching needed
- Only commodity hardware available
- Failover in tens of seconds, no interactive apps
- Permanent check sums in background

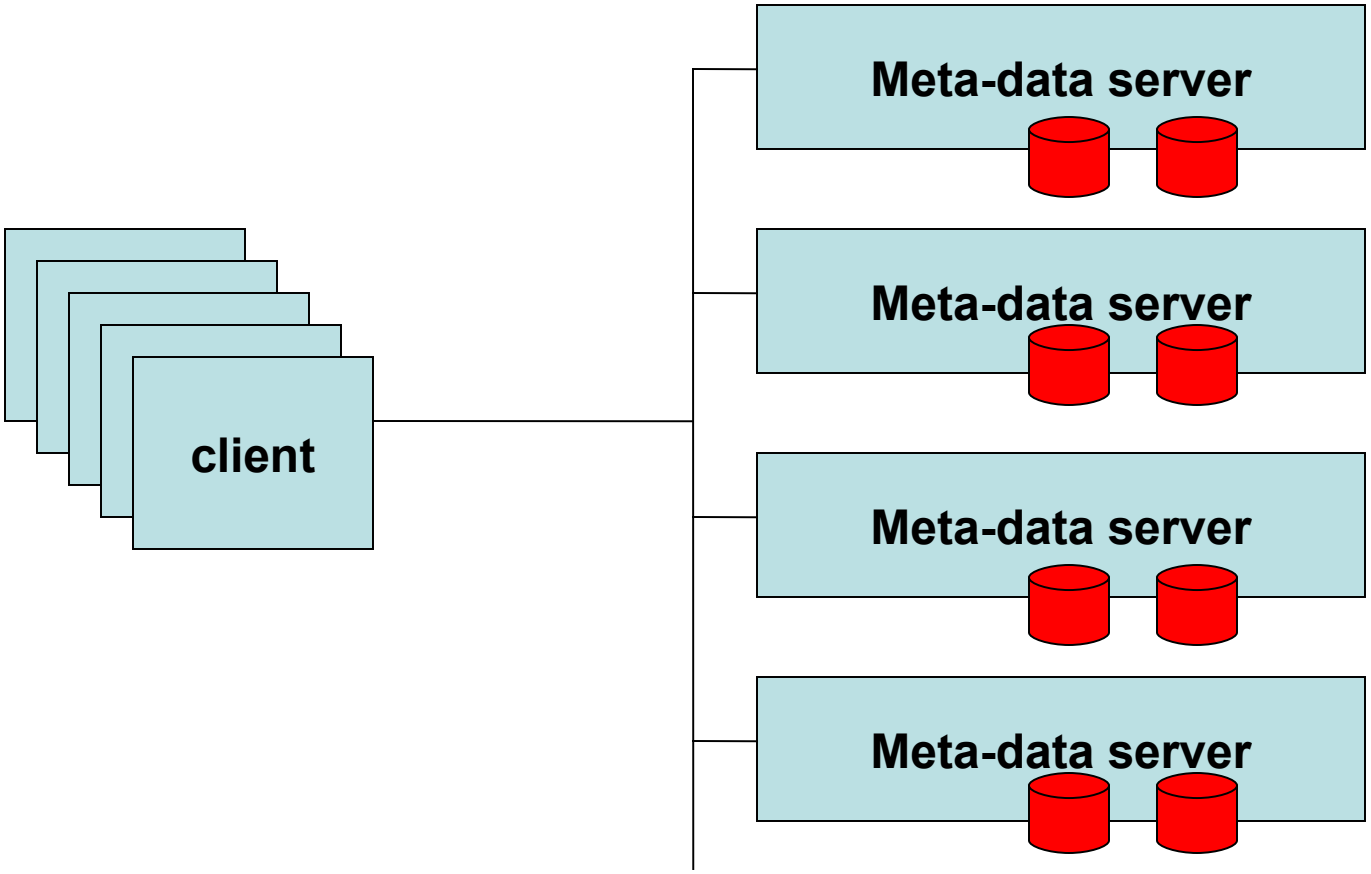

Fast lookup
/wacko.avi (C = C11:R1, C12:R2,…
Few meta-data
Meta-data server
Meta-data server
write(„wacko.avi“, offset)
Lease: R1:C11, R2:C12…
client
Processor blade R1
C11
write
Posix API ??
Write…
Processor blade R2
C12
Processor blade R3
C13
Constraints:
-nr. replicas
-Read/update modes
-Chunk size
-Reboot time
-Reorganiz.
Google FS Architecture
Storage grid

Design Limits Experienced

- change of use from batch processing only to also user facing/internet facing apps with low latency requirements
- number of files a critical resource: smaller files needed but meta-data size requires huge chunks
- failover and recovery times too long
- cell size limits reached/multi-cell kludges with static namespaces
- high numbers of requests (e.g. during map/reduce ops)
- inconsistencies critical to some applications and hard to understand

It looks like there is a need for a more general design for a distributed file system which is based on a distributed master concept.

Distributed Master File System

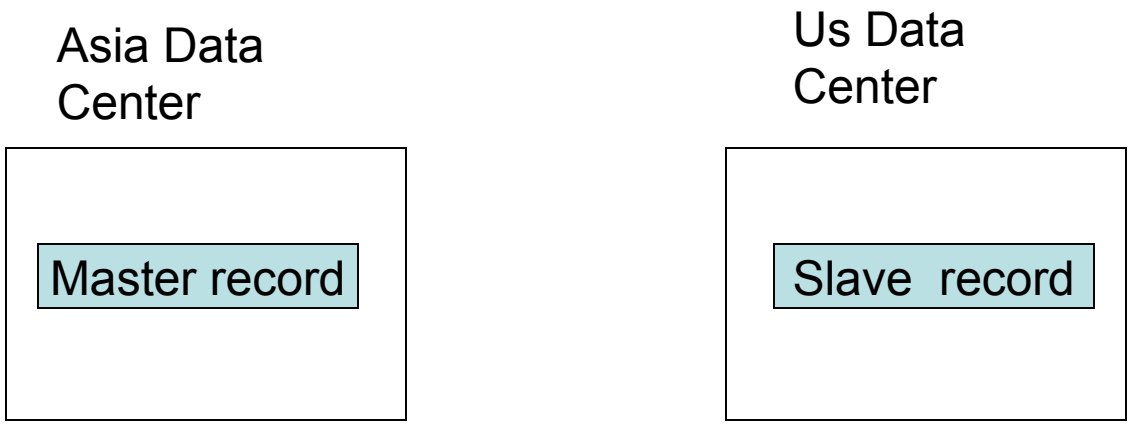


Do all masters have the same data? Synchronization based on multicast/Infiniband (Isilon solution). Will it scale? How does the network affect distributed algorithms?

Current Trends

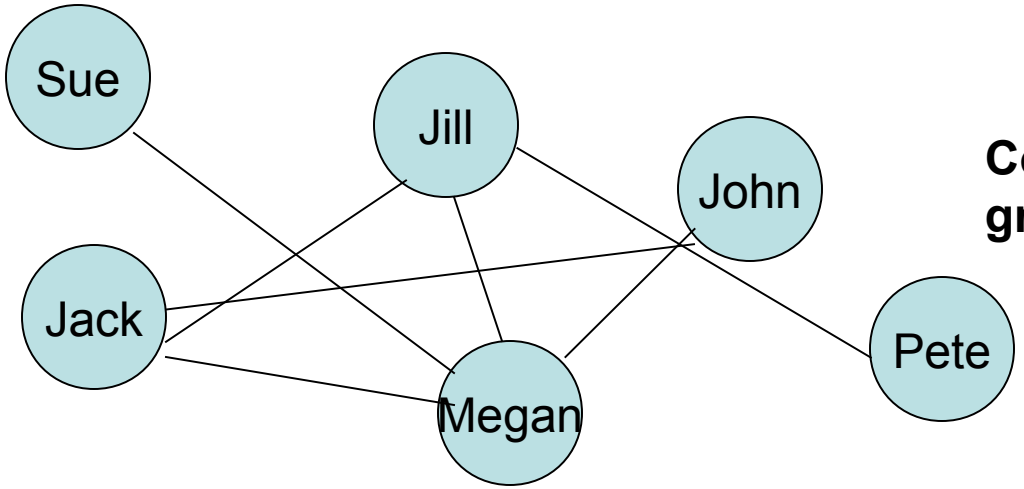
1. **Automation**
2. **Data-flow processing**
3. **Transparency**
4. **Transactions**
5. **Mixtures of eventually consistent and strongly consistent functions**

Current Trends in ULS: Automation

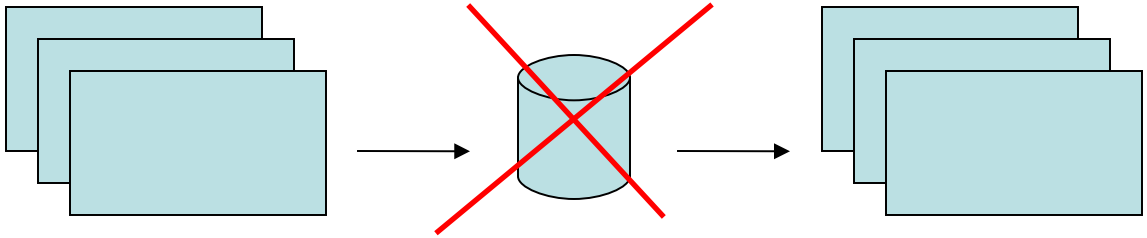


PNUTS (Yahoo) automatically shifts master records to the data center with the most writes. Most ULS are too large for manual scaling.

Current Trends in ULS: Real-time data-flow processors (CEP)



Complex social graphs



Graph processing is extremely compute intensive. Data flow processors use parallelism and avoid the disk bottleneck. Availability is worse though without intermediate results stored on disk. (see „large scale graph processing…“), Example: Twitter kafka/storm combo

Current Work: Spanner

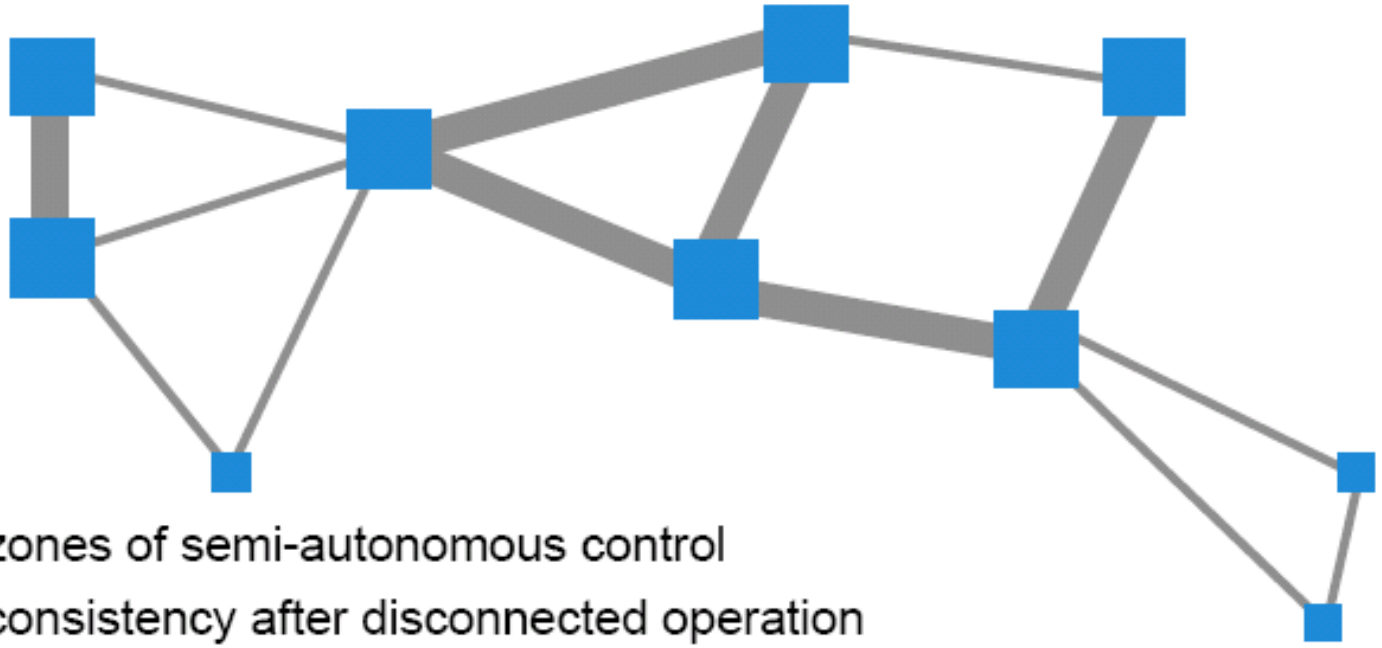
- Storage & computation system that spans all our datacenters
  - single global namespace
    - Names are independent of location(s) of data
    - Similarities to Bigtable: tables, families, locality groups, coprocessors, ...
    - Differences: hierarchical directories instead of rows, fine-grained replication
    - Fine-grained ACLs, replication configuration at the per-directory level
  - support mix of strong and weak consistency across datacenters
    - Strong consistency implemented with Paxos across tablet replicas
    - Full support for distributed transactions across directories/machines
  - much more automated operation
    - system automatically moves and adds replicas of data and computation based on constraints and usage patterns
    - automated allocation of resources across entire fleet of machines

Google

From: Jeff Dean

Design Goals for Spanner

- Future scale: ~10^6 to 10^7 machines, ~10^{13} directories, ~10^{18} bytes of storage, spread at 100s to 1000s of locations around the world, ~10^9 client machines



  - zones of semi-autonomous control
  - consistency after disconnected operation
  - users specify high-level desires:

    "99%ile latency for accessing this data should be <50ms"

    "Store this data on at least 2 disks in EU, 2 in U.S. & 1 in Asia"

Google

The Future ?

- Business Exchange Principles forming the Ambient Cloud
- Event-driven Systems
- Hierarchical Feedback loops for Self-Management
- Design Beyond Human Abilities: Emergent Systems

The Ambient Cloud (by Todd Hoff)

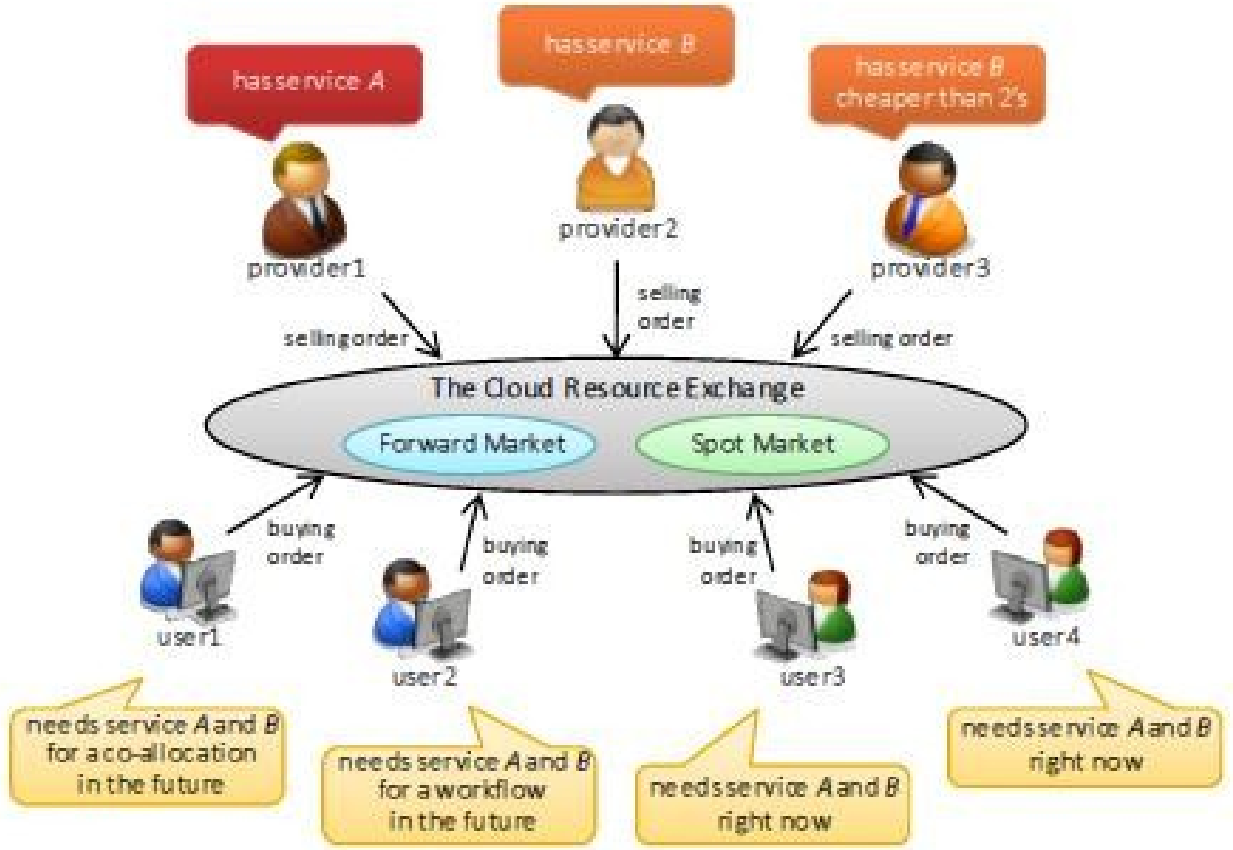
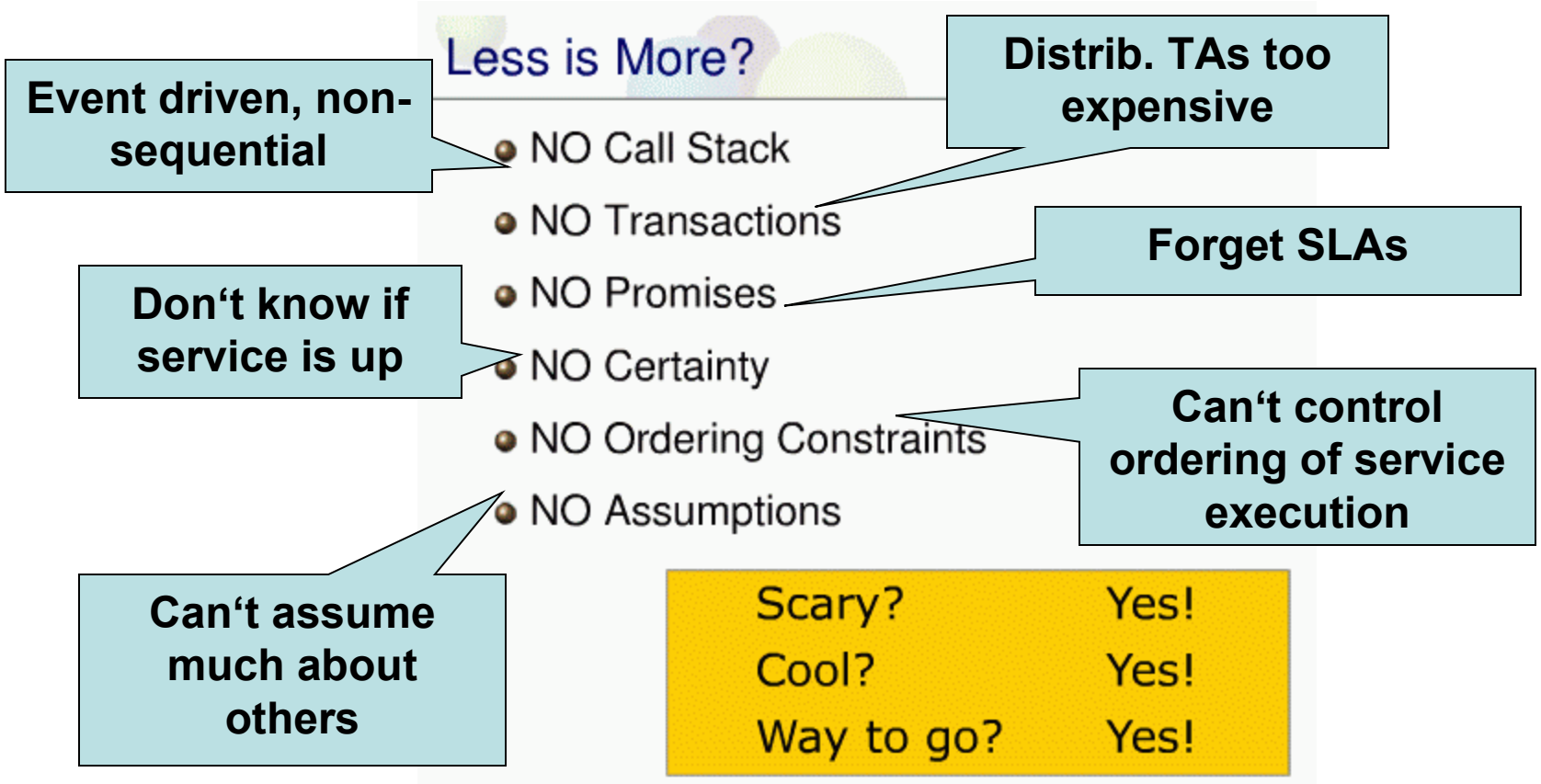


Fig. 1: Overview of the Cloud Resource Exchange

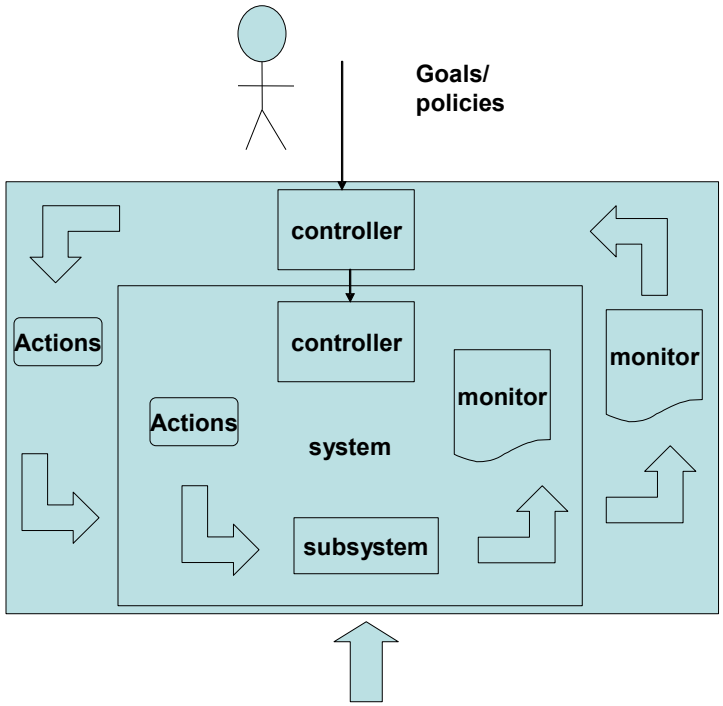
After all, there are 7 Billion people on this earth! Most of them will have powerful mobile phones and other personal computing devices.

Event-Driven Systems

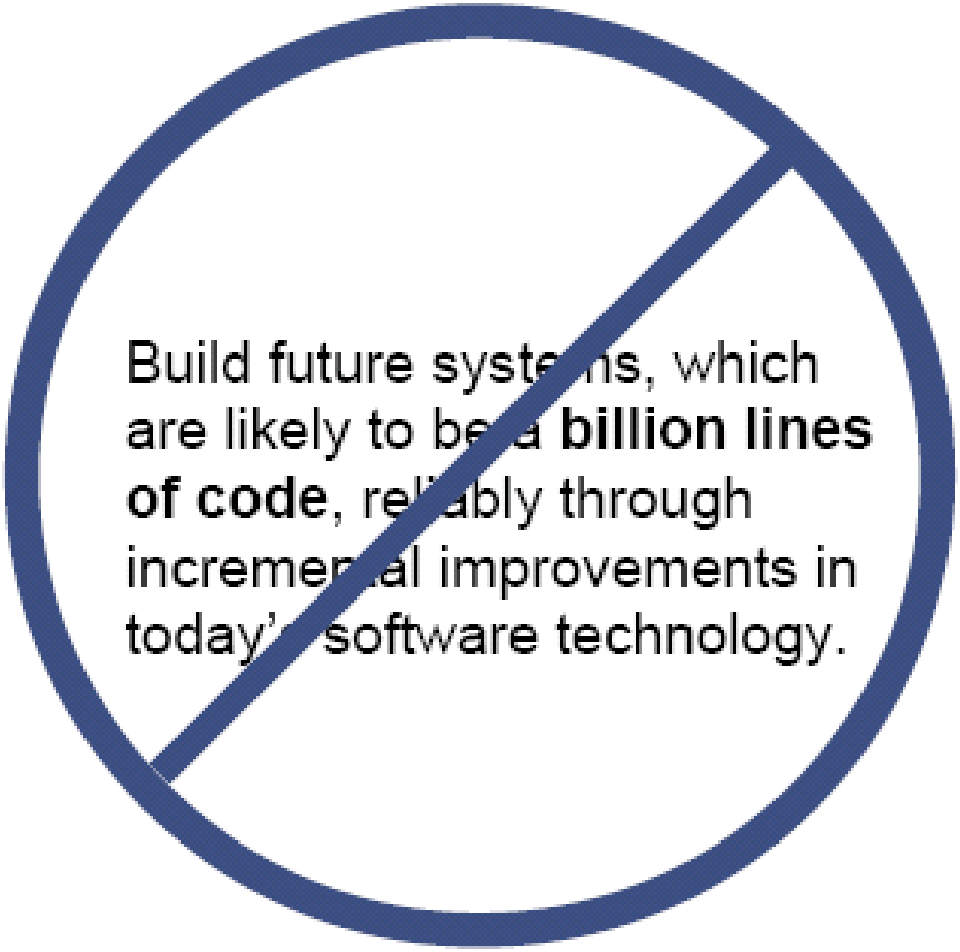


After Gregor Hohpe, Qcon Talk

Self-Organization through hierarchically organized feedback loops



After: Peter van Roy, Selfman.org



From: Linda Northrop, Scale changes Everything, OOPSLA 06

Richard P. Gabriel, Design beyond Human Abilities.

ULS are too big to design, to important to shut down and change by plan. They evolve somehow, grow. Neither forward nor inverse modelling works for them. Could we engineer them if P==NP?

Resources

- www.kriha.de/krihaorg/dload/ultra.pdf (draft)
- www.highscalability.com (Todd Hoff‘s portal for scalability)
- Jeff Dean, google tech talk, Designs, Lessons and Advice from Building Large Distributed Systems
- Ebay talk
- Richard P. Gabriel, Design beyond human abilities
- Linda Northrope
- Andreas Stiegler, MMOG infrastructures (http://www.hdm-stuttgart.de/~as147/mmo.pdf)
- Todd Hoff, http://highscalability.com/blog/2009/12/16/building-super-scalable-systems-blade-runner-meets-autonomi
- Kirk McKusick, Sean Quinlan, GFS: Evolution of fast forward
- [Persyn] Jurriaan Persyn, Database Sharding at Netlog, Presentation held at Fosdem 2009 http://www.jurriaanpersyn.com/archives/2009/02/12/database-sharding-at-netlog-with-mysql-
- Todd Lipcon, Design Patterns for Distributed Non-Relational Databases aka Just Enough Distributed Systems To Be Dangerous
- M.Stonebreaker, The end of an architectural era - It‘s time for a complete rewrite
- [Wilson] Jim R. Wilson, Understanding Hbase and BigTable, http://jimbojw.com/wiki/index.php?title=Understanding_Hbase_and_BigTable