Wrap Up

COS 418: Distributed Systems

Lecture 23

Wyatt Lloyd

Back in Lecture 1…

Distributed Systems, Why?

- Or, why not 1 computer to rule them all?
- Failure => Fault Tolerance
- Limited computation/storage => Scalability
- Physical location => Availability, Low Latency

Distributed Systems Goal

- Service with higher-level abstractions/interface
  - e.g., database, programming model, …
- Hide complexity - Do “heavy lifting” so app developer doesn’t need to
  - Reliable (fault-tolerant)
  - Scalable (scale-out)
  - Strong guarantees (consistency and transactions)
- Efficiently
  - Lower latency (faster interactions, e.g., page load)
  - Higher throughput (fewer machines)

What We Learned

(Much of it at least, at a *very* high level)

Network communication

- How can multiple computers communicate?
- Networking stack solves this for us!
- We use it to build distributed systems, relying on the guarantees it provides.

Remote Procedure Calls

- Additional layer on top of networking stack
- At least once – dealing with failures!
- At most once – ensuring correctness despite concurrency and failures

Time, logical clocks

- Concurrency!
- Real time often inadequate
  for distributed systems?
- Lamport clocks: A → B => LC(A) < LC(B)
- Vector clocks: A → B <=> VC(A) < VC(B)

Eventual Consistency, Bayou

- Favor availability above all else
  - e.g., disconnected dropbox operation
- Eventual consistency
- Bayou system design
  - Operation log (logical, not physical, replication)
  - Causal consistency from log propagation and lamport timestamps

P2P Systems & DHTs

- Efficiency of various designs
- Goal: scale lookup state, lookup computation, storage; fault tolerant
- Scale lookup state, lookup computation w/ Chord
- Scale storage with sharding
- Fault tolerance through replication, robust protocols

Dynamo

- Favor availability above all + scalable storage
- Eventual consistency (really eventual)
- Zero-hop DHT on top of data sharded with consistent hashing
  - Virtual nodes enable better load balancing (improves throughput), but design to still ensure fault tolerance

So far…

- Can build systems that are fault tolerant, scalable, provide low latency, highly available

- But…

- Weak guarantees

	Fault Tolerant	Scalable	Highly Available & Low Latency	Guarantees
Bayou	yes	no	yes	causal
Dynamo	yes	yes	yes	eventual

Strong Guarantees + Fault Tolerance

- Linearizability: acts just like 1 machine processing requests 1 at a time!

- Replicated state machines:
    - Log of operations, execute in order
    - Primary-backup (and VM-FT)
        - Special mechanism for failure detection
        - React to failure
    - Paxos, RAFT
        - Built in failure detection using quorums (f+1 out of 2f+1)
        - Mask non-leader failure

	Fault Tolerant	Scalable	Highly Available & Low Latency	Guarantees
Bayou	yes	no	yes	causal
Dynamo	yes	yes	yes	eventual
Paxos/RAFT	yes	no	no	linearizability

Impossibility Results Guide Us

- CAP: Must choose either availability of all replicas or consistency between replicas
- PRAM: Must choose either low latency of operations or consistency between replicas

Availability + Low Latency + Scalability + Strong*er* Guarantees

- COPS provides causal consistency
  - Stronger guarantees impossible w/ low latency
  - Like a scalable Bayou

- Sharding to scale storage within a datacenter
- Geo-replicate data across datacenters
  - Replication and sharding!

- New protocols for replicating writes between replicas and reading data
  - Distributed protocols w/ work on only some machines in each replica for scalability
  - Consistently reading data across shards required transactions

	Fault Tolerant	Scalable	Highly Available & Low Latency	Guarantees
Bayou	yes	no	yes	causal
Dynamo	yes	yes	yes	eventual
Paxos/RAFT	yes	no	no	linearizability
COPS	yes	yes	yes	Causal & read-only txns

Strong Guarantees + Scalability

- Strict Serializability: acts just like 1 machine processing requests 1 at a time with transactions across shards
- Atomic Commit w/ 2PC
- Concurrency control
  - 1 Big Lock: No concurrency ☹
  - 2PL: Growing phase then shrinking phase
  - OCC: Assume you will succeed, only acquire locks during 2PC

	Fault Tolerant	Scalable	Highly Available & Low Latency	Guarantees
Bayou	yes	no	yes	causal
Dynamo	yes	yes	yes	eventual
Paxos/RAFT	yes	no	no	linearizability
COPS	yes	yes	yes	causal & read-only txns
2PL	no	yes	-	strict serializability

Strong Guarantees + Scalability + Fault Tolerance

- Google's Spanner
  - Sharding to scale storage
  - Paxos for fault tolerance
  - 2PL + 2PC for read-write transactions
    - Strict serializability
    - Scalable processing … mostly

- So many reads, make read-only txns efficient!
  1. Strictly serializable read-only transactions that block, but do not acquire any locks
  2. Stale read-only transactions that do not even block

- Enabled by TrueTime
  - TrueTime gives bounded wall-clock time interval
  - Commit wait ensures a transaction completes after its wall-clock commit time

	Fault Tolerant	Scalable	Highly Available & Low Latency	Guarantees
Bayou	yes	no	yes	causal
Dynamo	yes	yes	yes	eventual
Paxos/RAFT	yes	no	no	linearizability
COPS	yes	yes	yes	causal & read-only txns
2PL	no	yes	-	strict serializability
Spanner (stale-read)	yes	yes	no (yes)	strict serializability (stale)

Strong Guarantees + Scalability + Low Latency?

- SNOW is impossible for read-only transactions

- Must choose either the strongest guarantees
  (Strict Serializability & Write transactions)
  or the lowest latency
  (Non-blocking & One Round)

- PRAM / CAP are for replication
  SNOW / NOCS is for sharding

Now You Can!

- Build systems that are fault tolerant, scalable, provide low latency, highly available
    - + strong*er* guarantees, but not the strongest

- OR

- Build systems that are fault tolerant, scalable, and provide the strongest guarantees

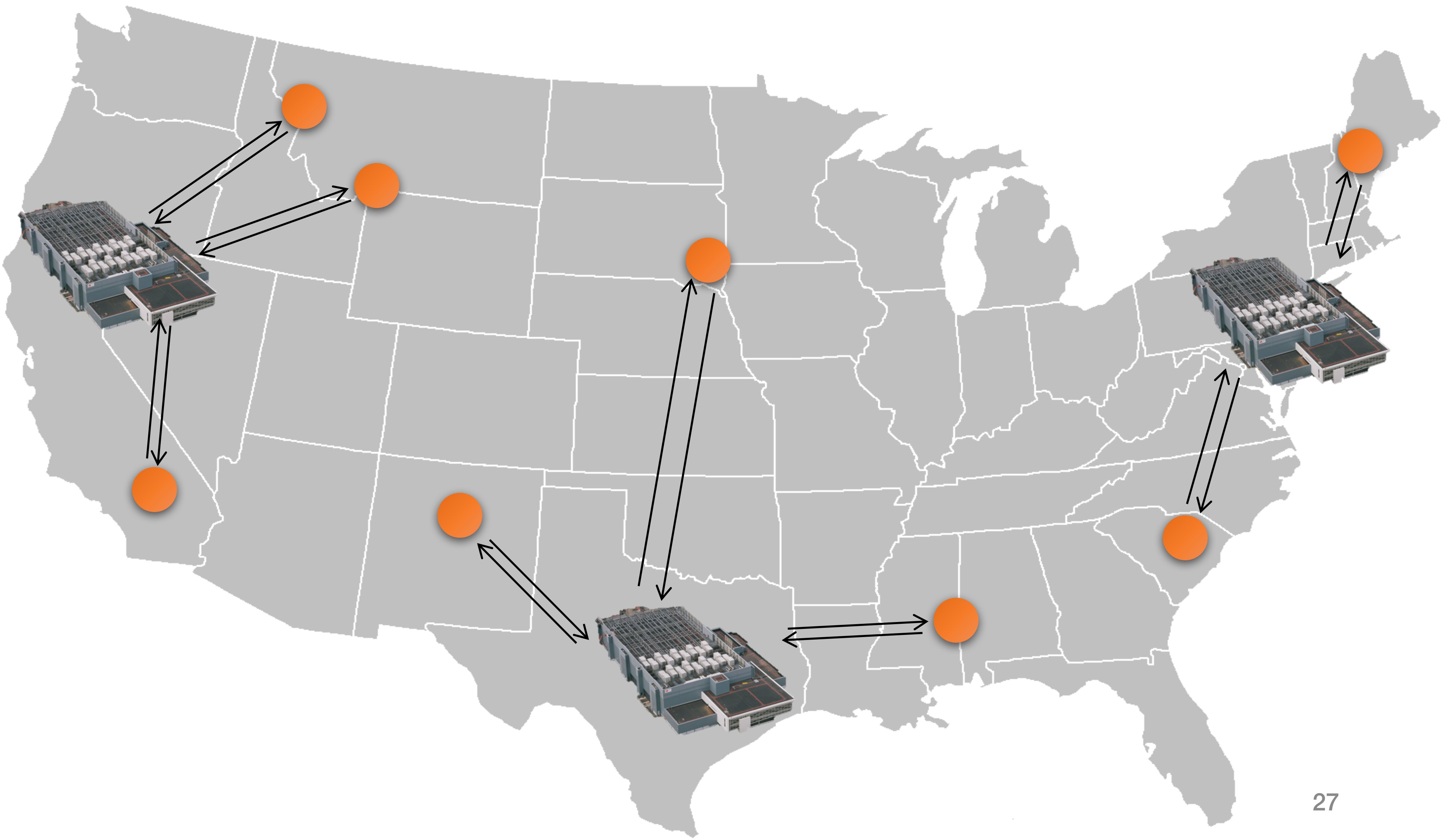
Making Systems Faster

- Exploiting many types of parallelism in Facebook's Streaming Video Engine

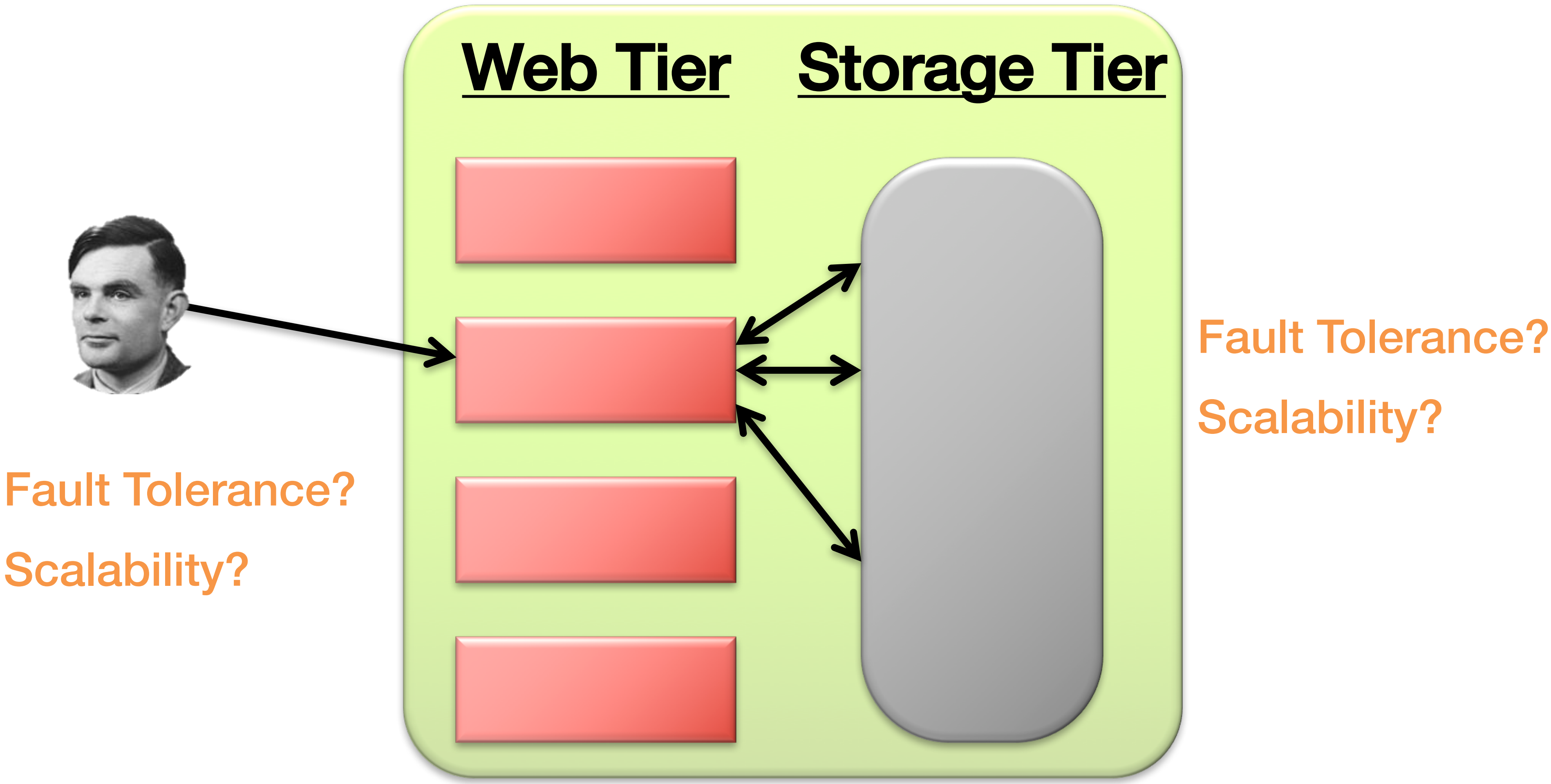
- Reasoning about the performance of distributed systems using a mental model

Let’s See It In Action

Client → Frontend Server



Inside the Datacenter

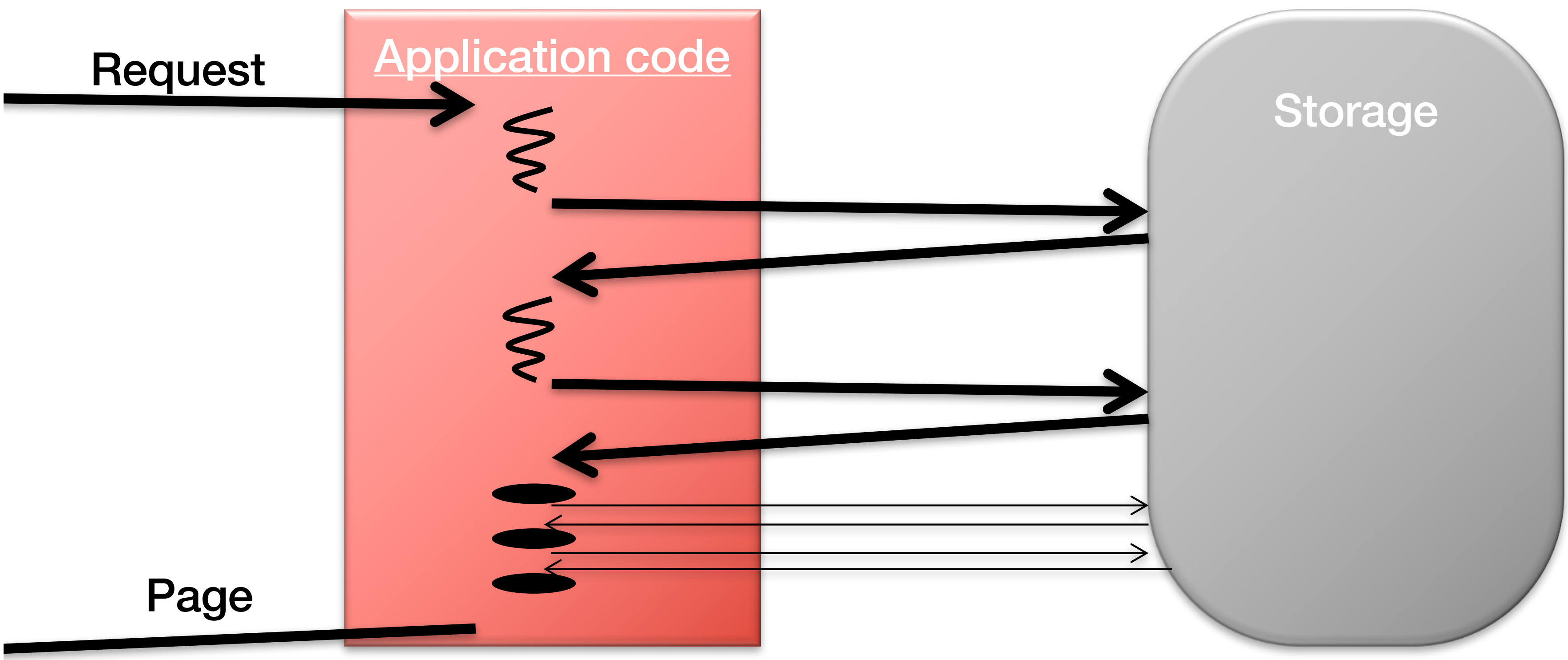


Executes frontend, application code

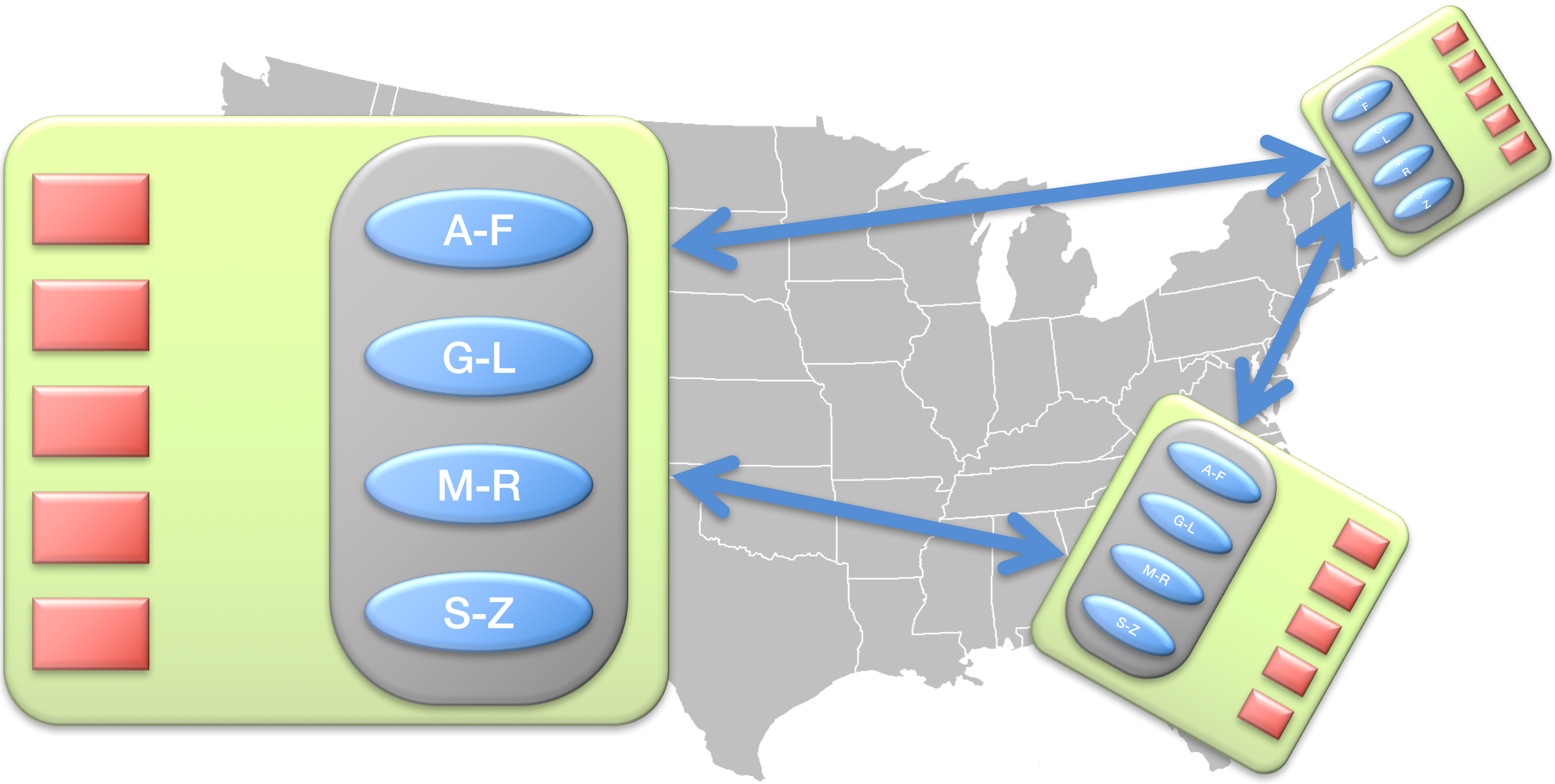
Stores state, provides …

Application Code Reads/Writes to the Storage Tier

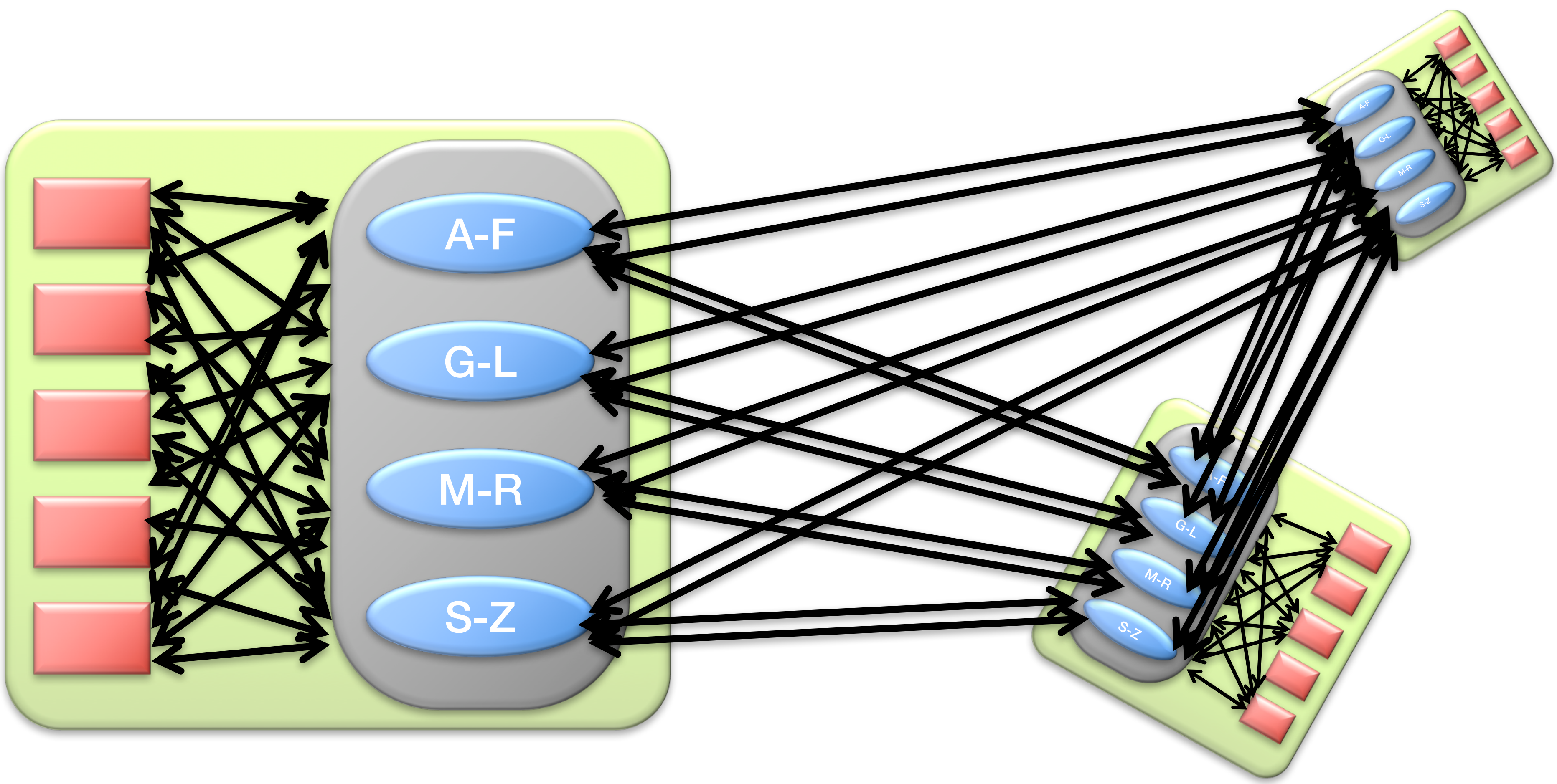
Facebook page load has 1000s of reads,
chains of sequential reads dozens long [HotOS '15]



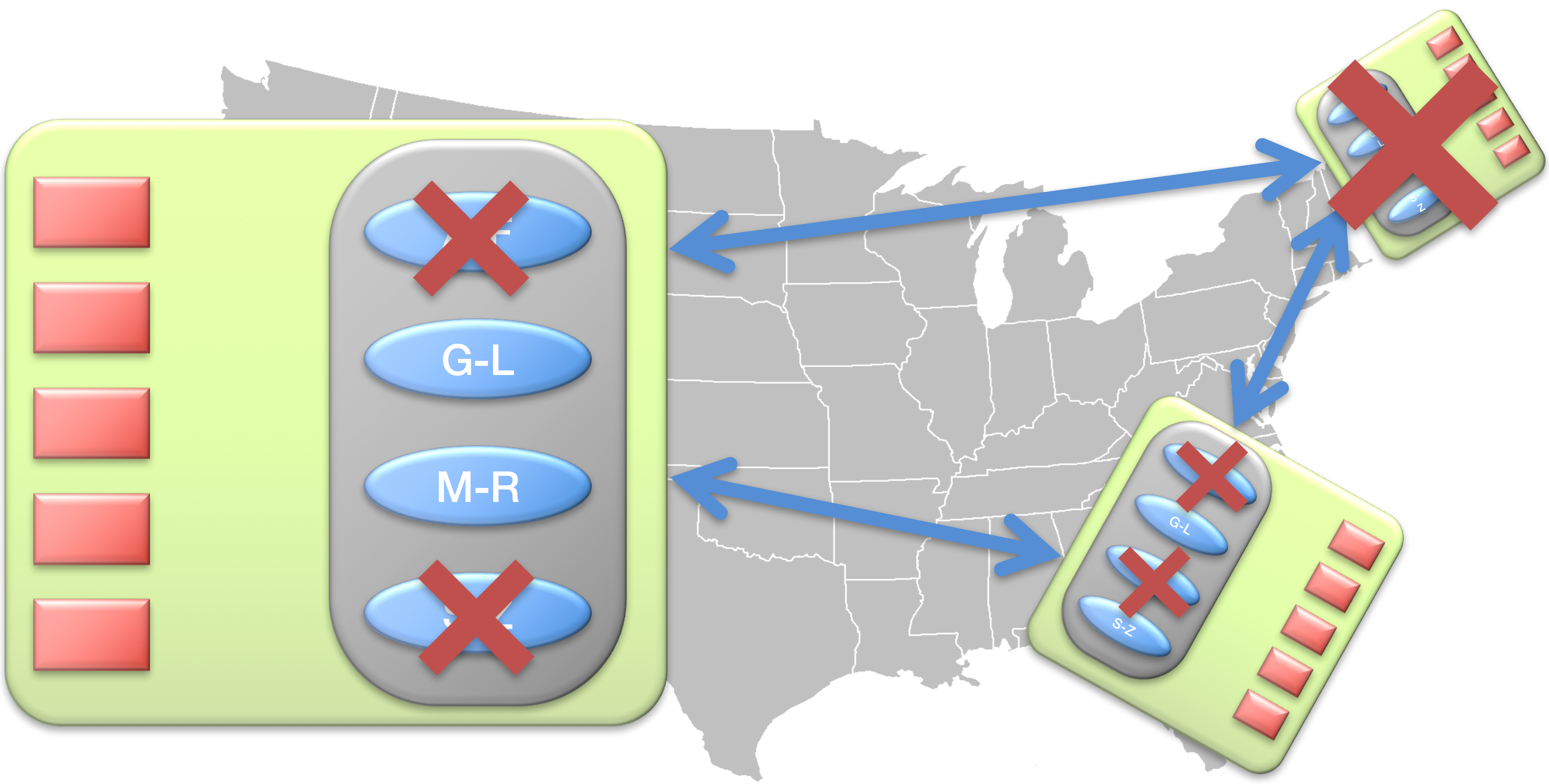
Scalable Storage is Sharded and Geo-Replicated



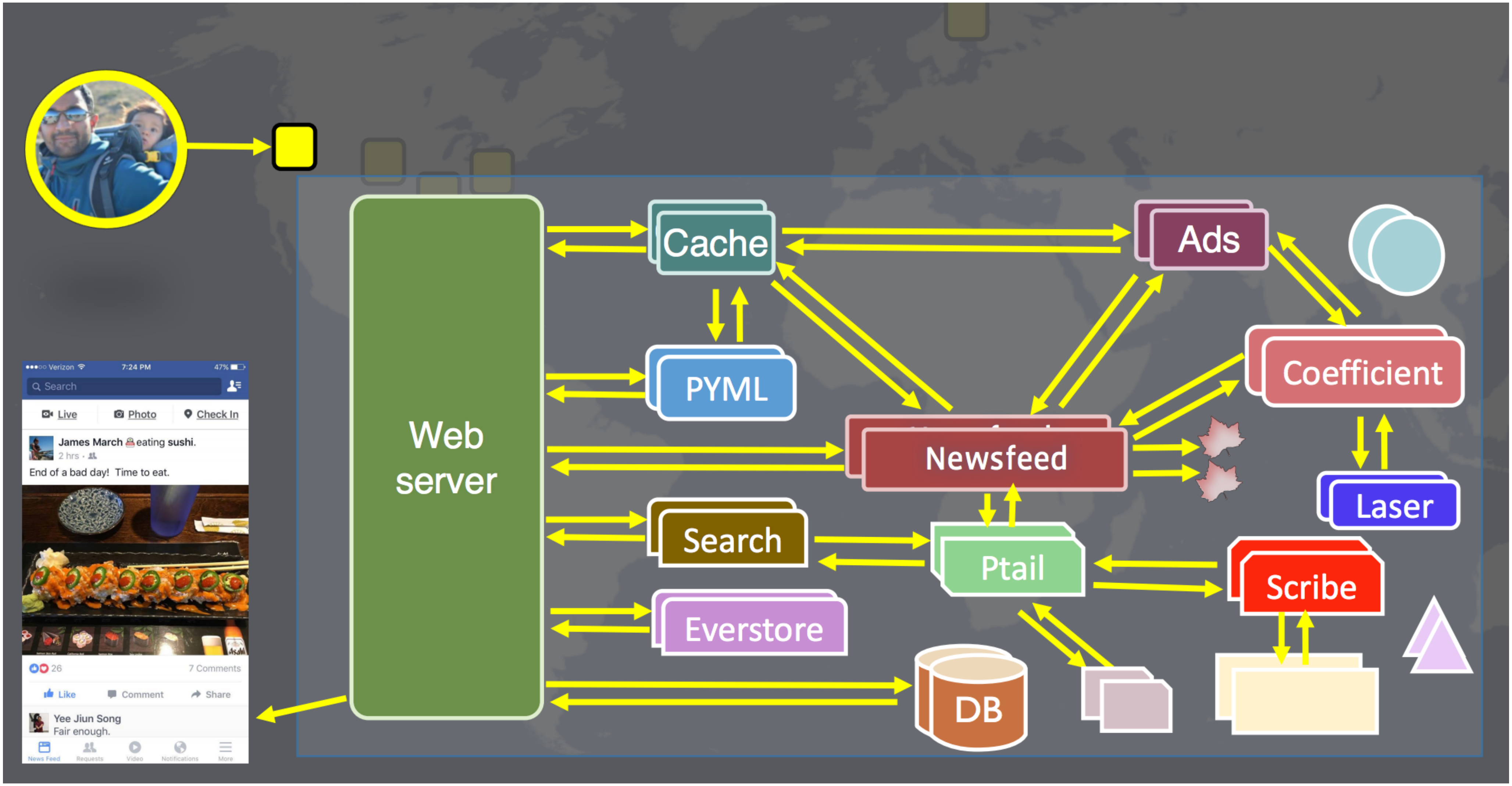
So Much Concurrency!



So Many Failures!



Not Just One Backend System



[Diagram from Kaushik Veeraraghavan's OSDI '16 Talk]

Each Backend System is a Distributed System

- But with different tradeoffs and designs depending on use

- LIKE count?
  - Eventually consistent storage system

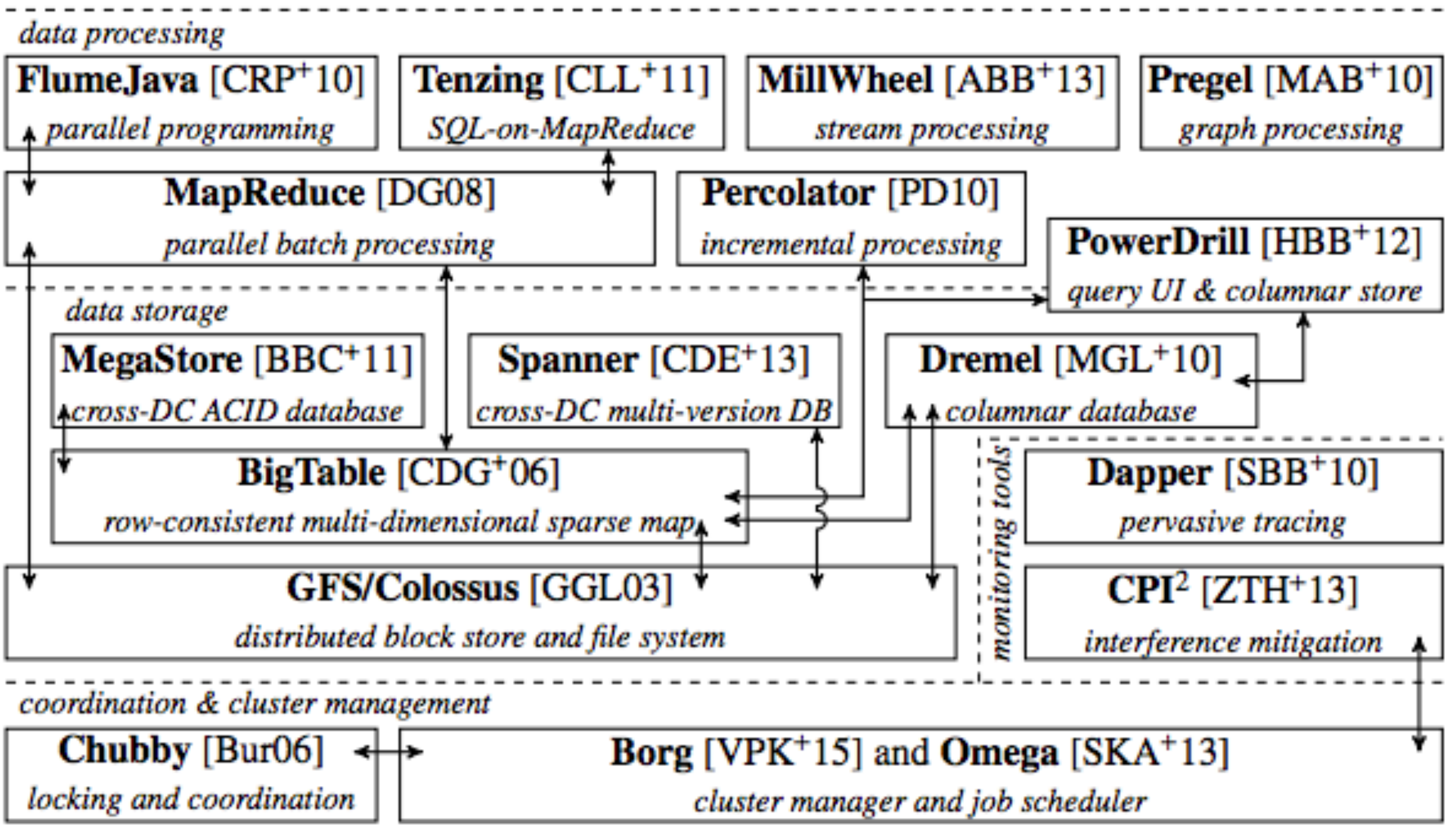
- User Password?
  - Strongly consistent storage system

Each Backend System is a Distributed System

- Search results
  - Use precomputed index, precomputed with MapReduce, or a more efficient, specialized system

- Trending hashtags
  - Use a stream processing system to continuously update computation about what is most popular

Distributed Systems on Distributed Systems on …



[Diagram from Malte Schwarzkopf PhD Thesis 2015]

More Systems in the Fall?!

- COS 461 – Computer Networks
    - Kyle Jamieson
- COS 318 – Operating Systems
    - JP Singh
- COS 375 – Computer Architecture
    - David August

Thanks!