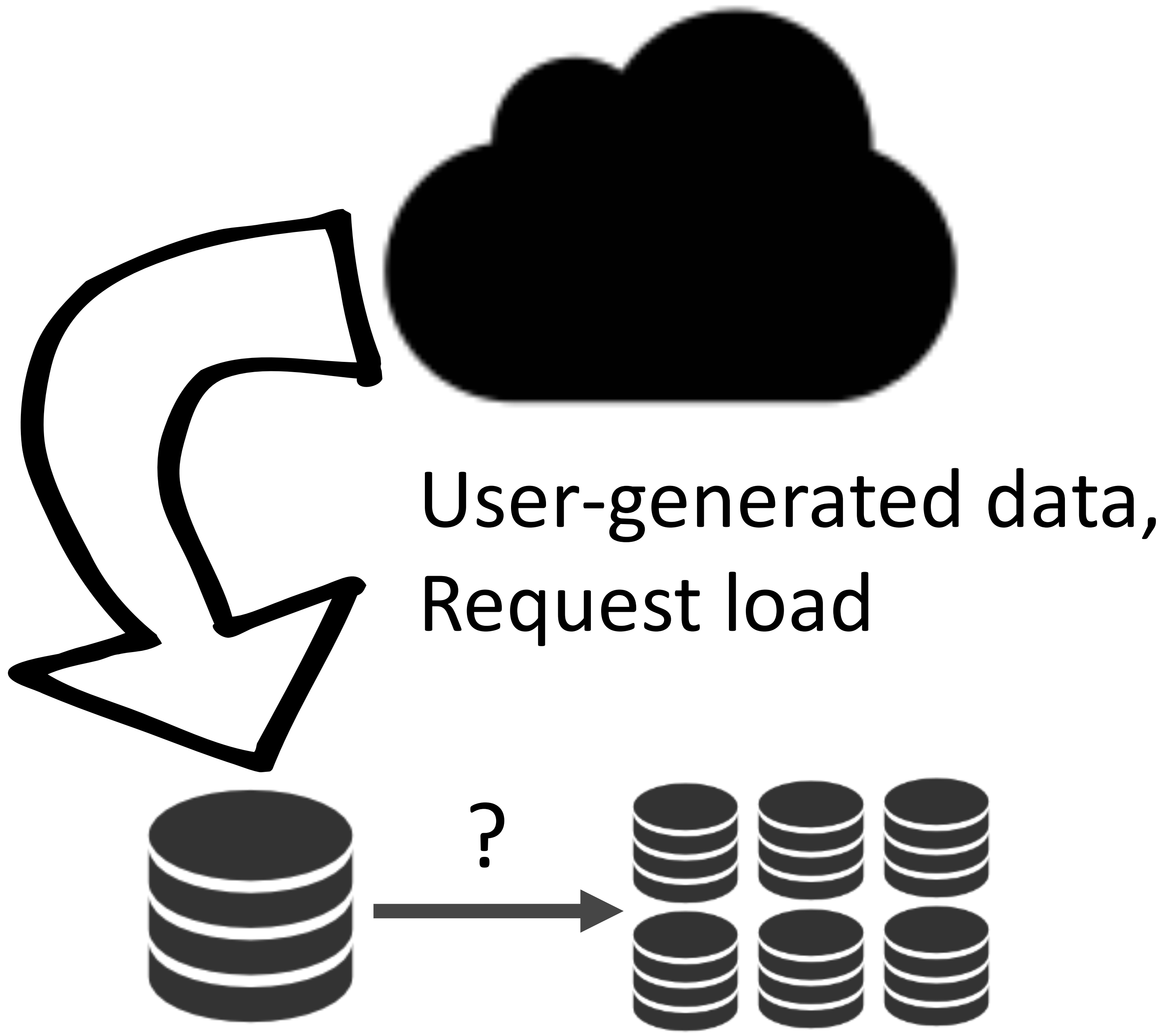
Advanced Data Management (CSCI 640/490)

Scalable Databases

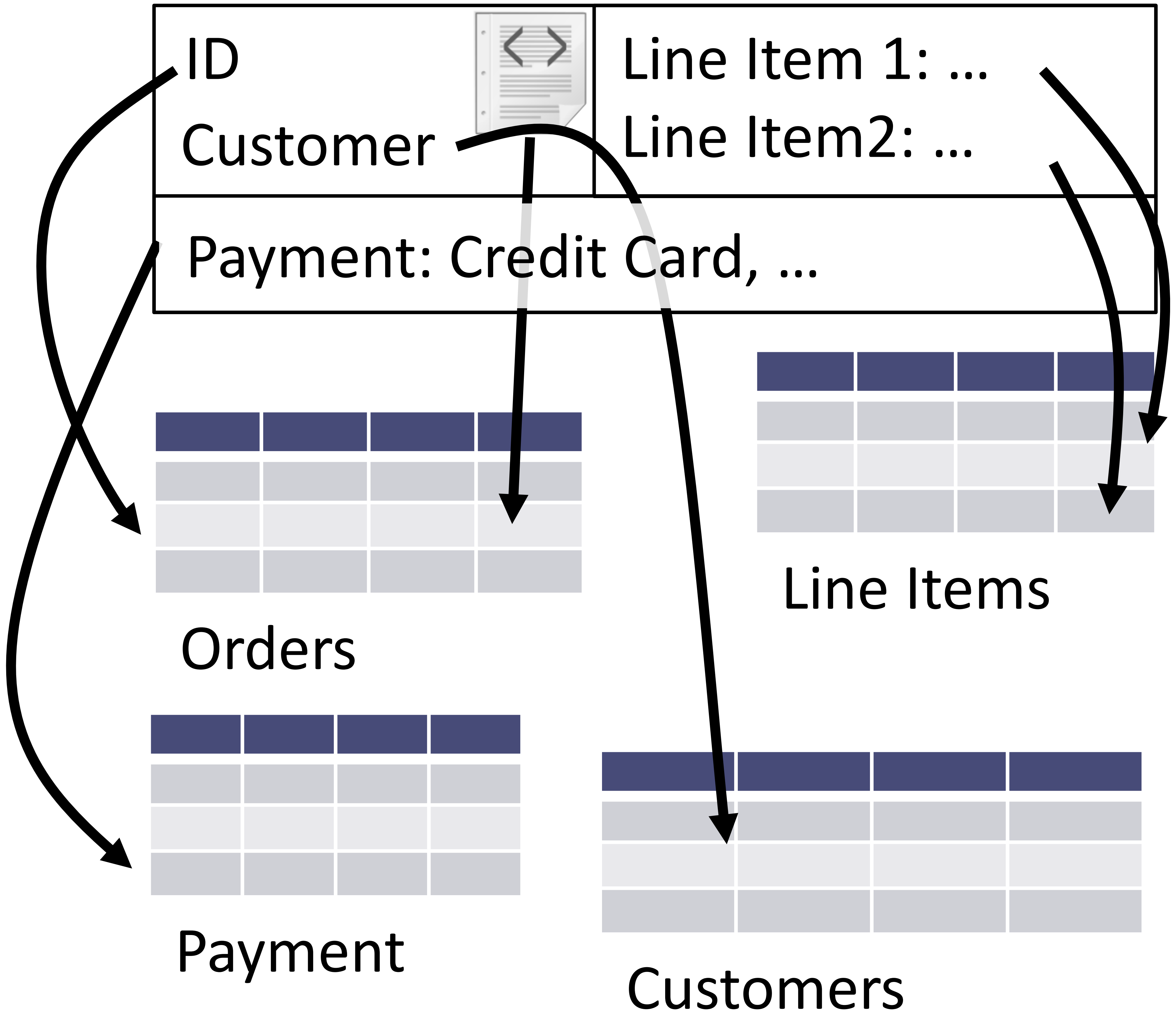
Dr. David Koop

NoSQL Motivation

Scalability



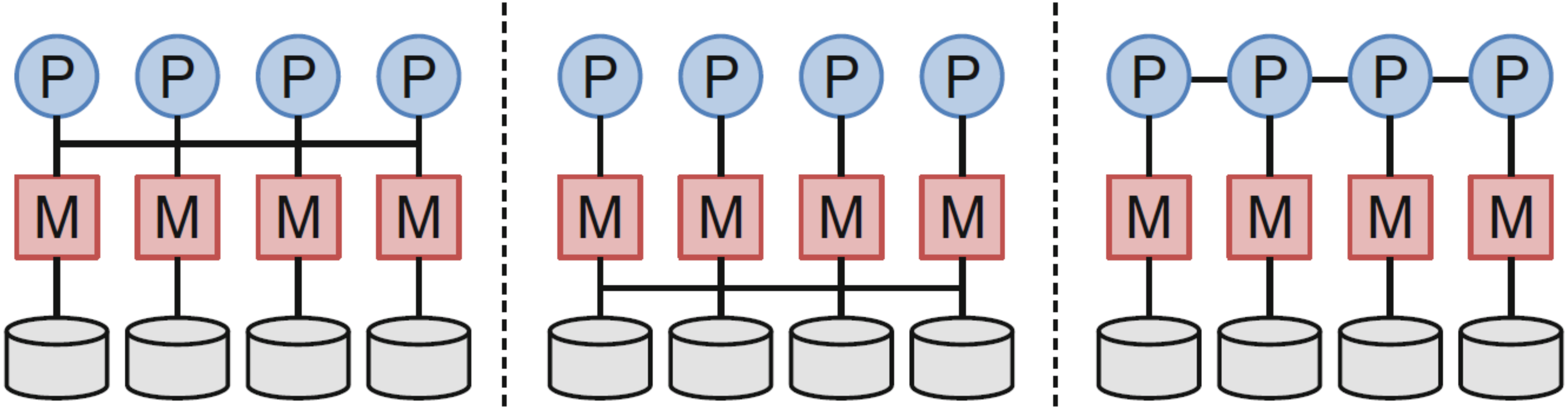
Impedance Mismatch



[F. Gessert et al., 2017]

Shared Nothing Architecture

Shift towards higher distribution & less coordination:



Shared Memory
e.g. "Oracle 11g"

Shared Disk
e.g. "Oracle RAC"

Shared Nothing
e.g. "NoSQL"

[F. Gessert et al., 2017]

Stonebraker: The End of an Architectural Era

- "RDBMSs were designed for the business data processing market, which is their sweet spot"
- "They can be beaten handily in most any other market of significant enough size to warrant the investment in a specialized engine"
- Changes in markets (science), necessary features (scalability), and technology (amount of memory)
- RDBMS Overhead: Logging, Latching, and Locking
- Relational model is not necessarily the answer
- SQL is not necessarily the answer

Horizontal Partitioning vs. Vertical Partitioning

Vertical Partitions

VP1

CUSTOMER ID	FIRST NAME	LAST NAME
1	TAEKO	OHNUKI
2	O.V.	WRIGHT
3	SELDA	BAĞCAN
4	JIM	PEPPER

VP2

CUSTOMER ID	FAVORITE COLOR
1	BLUE
2	GREEN
3	PURPLE
4	AUBERGINE

Original Table

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

Horizontal Partitions

HP1

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN

HP2

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

[M. Drake]

NoSQL Classification Criteria

Data Model

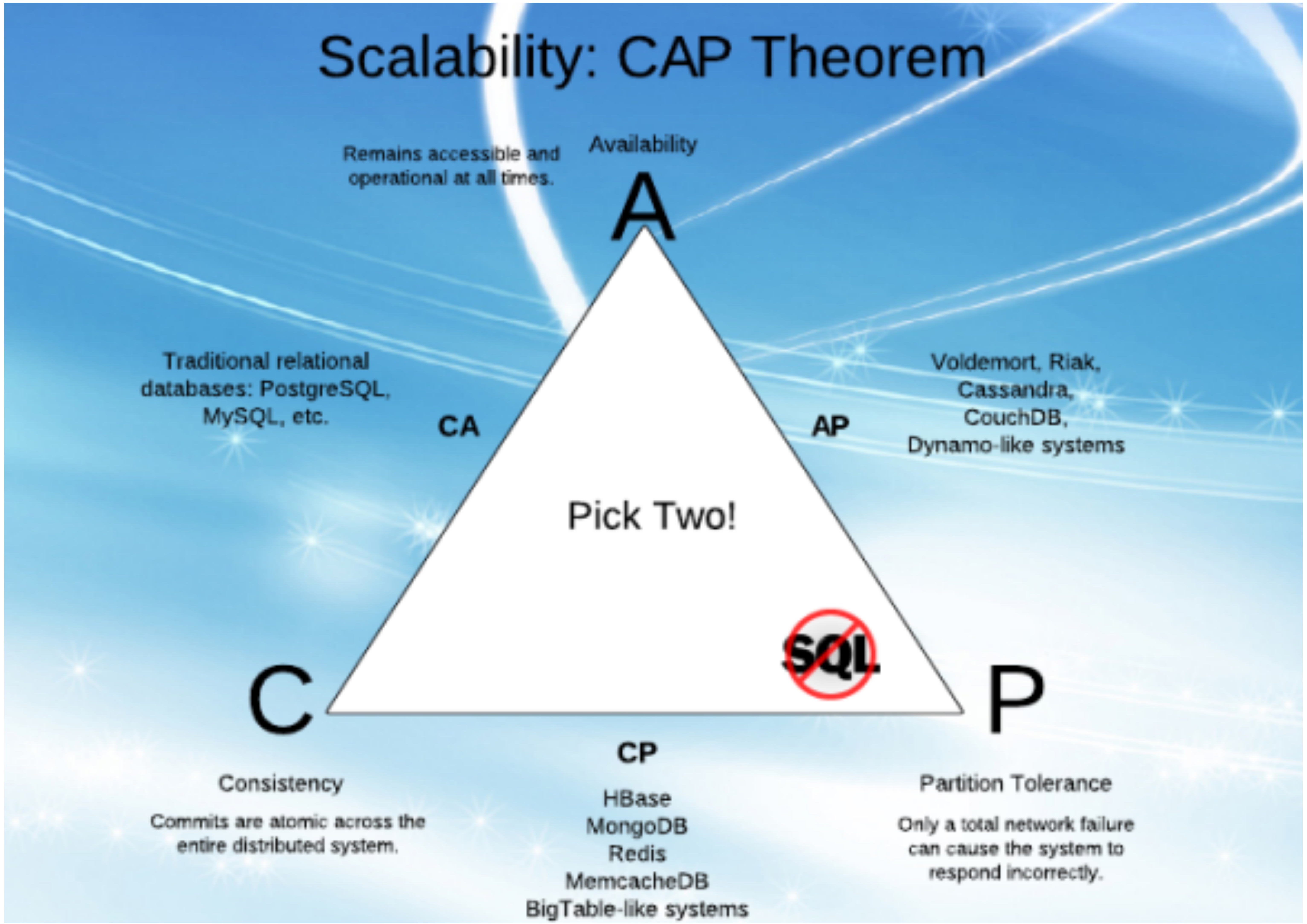
- Key-Value
- Wide-Column
- Document
- Graph

Consistency/Availability Trade-Off

- **AP**: Available & Partition Tolerant
- **CP**: Consistent & Partition Tolerant
- **CA**: Not Partition Tolerant

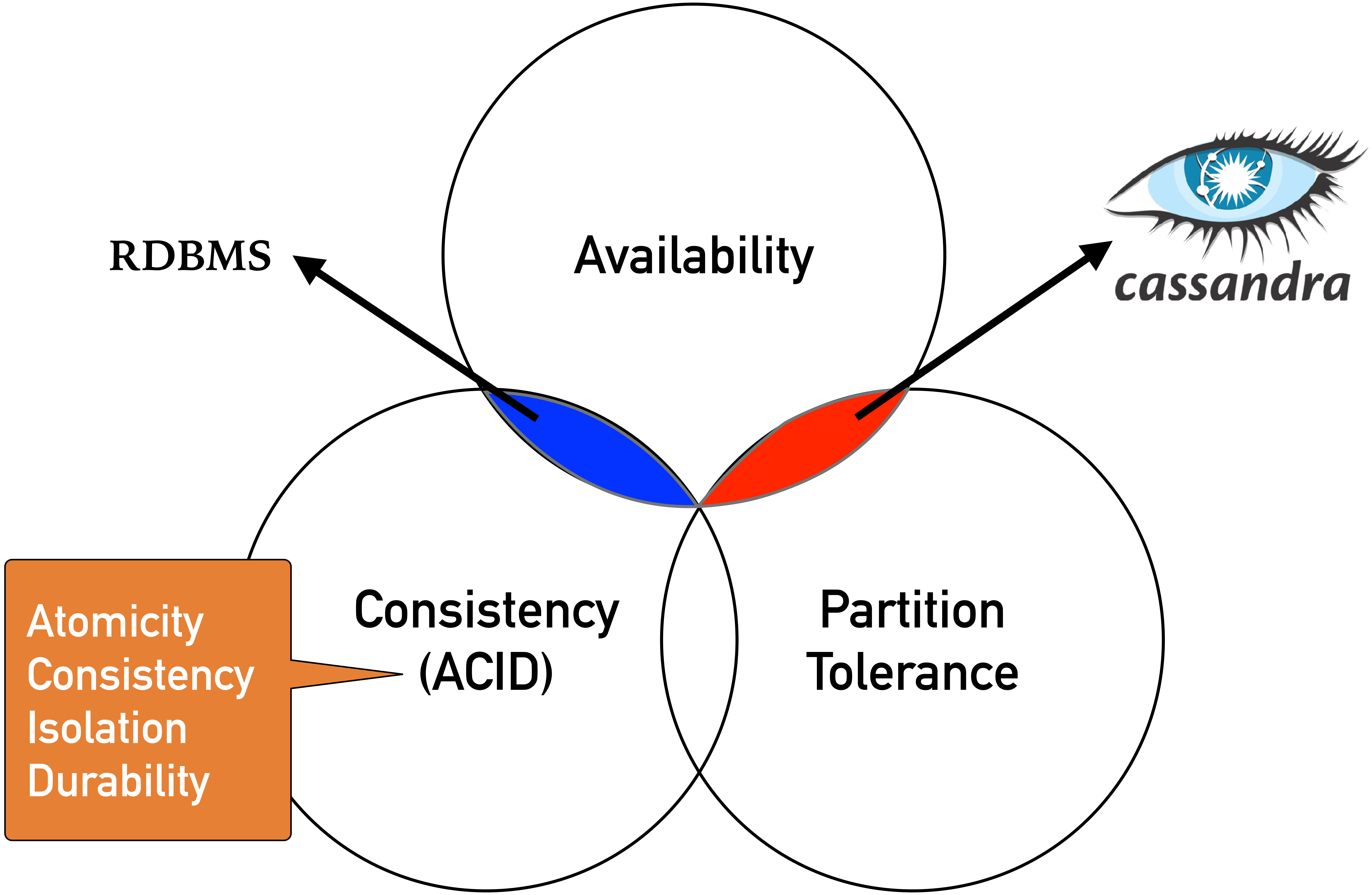
[F. Gessert et al., 2017]

CAP Theorem



[E. Brewer]

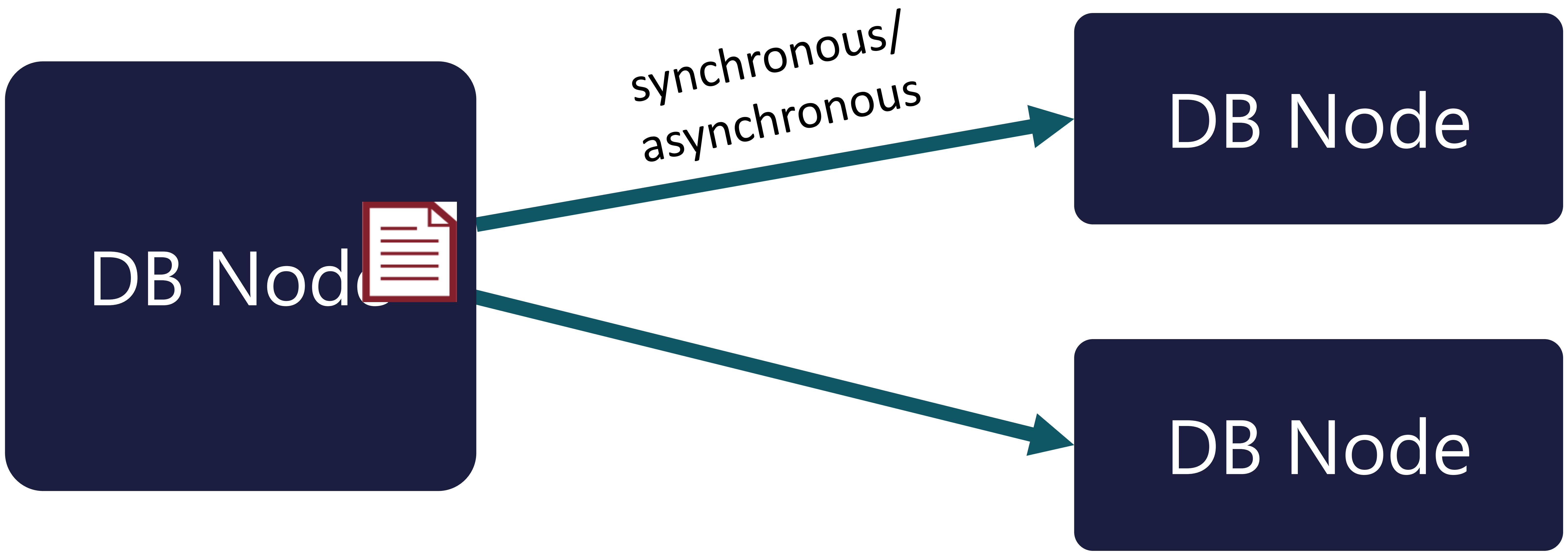
Cassandra and CAP



[G. Atil]

Replication

- Store N copies of each data item
- Consistency model: synchronous vs. asynchronous
- Coordination: Multiple Primary, Primary/Replica



[F. Gessert et al., 2017]

Replication: When

- Asynchronous (lazy)
  - Writes are acknowledged immdediately
  - Performed through log shipping or update propagation
  - Pro: Fast writes, no coordination needed
  - Con: Replica data potentially stale (inconsistent)
- Synchronous (eager)
  - The node accepting writes synchronously propagates updates/transactions before acknowledging
  - Pro: Consistent
  - Con: needs a commit protocol (more roundtrips), unavailable under certain network partitions

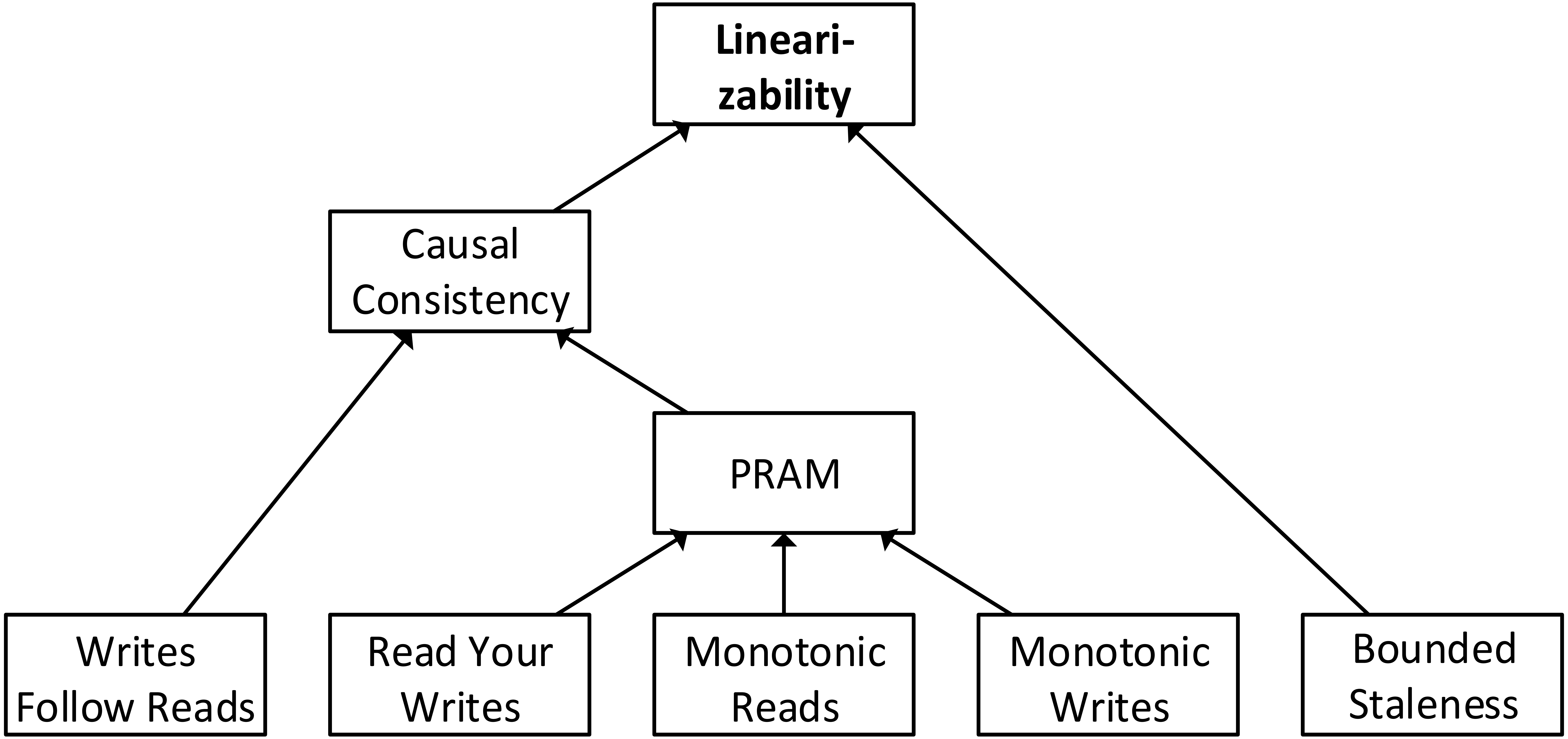
[F. Gessert et al., 2017]

Replication: Where

- Primary-Replica (Primary Copy)
  - Only a dedicated primary is allowed to accept writes, replicas are read-replicas
  - Pro: reads from the primary are consistent
  - Con: primary is a bottleneck and SPOF
- Multi-Primary (Update anywhere)
  - The server node accepting the writes synchronously propagates the update or transaction before acknowledging
  - Pro: fast and highly-available
  - Con: either needs coordination protocols (e.g. Paxos) or is inconsistent

[F. Gessert et al., 2017]

Consistency Levels



[via [F. Gessert et al., 2017]]

Assignment 4

- Work on Data Integration and Data Fusion
- Integrate artist datasets from different institutions (MET, NGA, Chicago Art Institute, …)
- Record Matching:
  - Which artists are the same?
- Data Fusion:
  - Names
  - Birth/death dates
  - Nationality

Slides: Introduction to Cassandra

Robert Stupp

What is Cassandra?

- Fast Distributed (Column Family NoSQL) Database
  - High availability
  - Linear Scalability
  - High Performance
- Fault tolerant on Commodity Hardware
- Multi-Data Center Support
- Easy to operate
- Proven: CERN, Netflix, eBay, GitHub, Instagram, Reddit

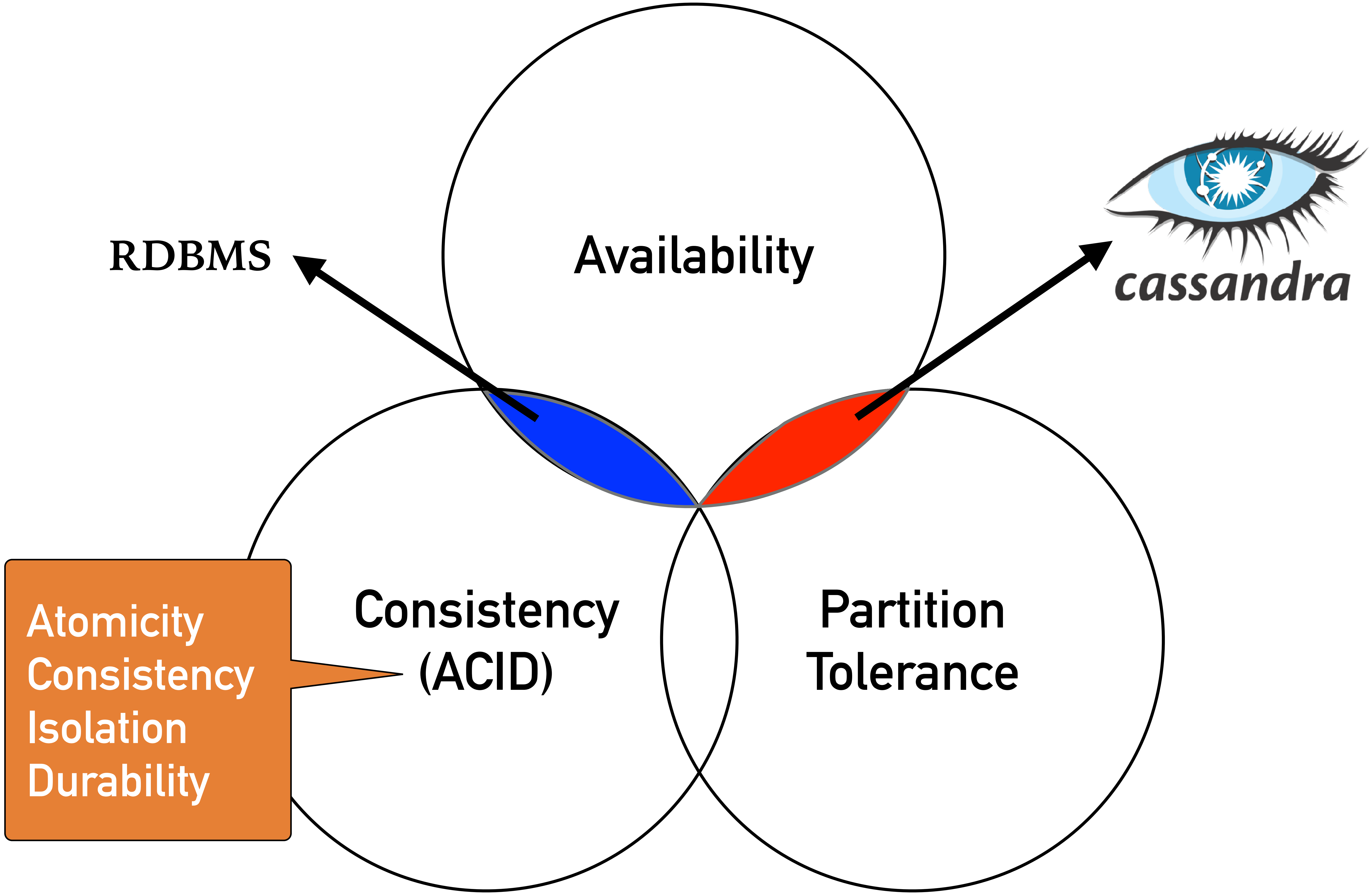
[G. Atil]

Relational Databases vs. Cassandra

Relational Database	**Cassandra**
Handles moderate incoming data velocity	Handles high incoming data velocity
Data arriving from one/few locations	Data arriving from many locations
Manages primarily structured data	Manages all types of data
Supports complex/nested transactions	Supports simple transactions
Single points of failure with failover	No single points of failure; constant uptime
Supports moderate data volumes	Supports very high data volumes
Centralized deployments	Decentralized deployments
Data written in mostly one location	Data written in many locations
Supports read scalability (with consistency sacrifices)	Supports read and write scalability
Deployed in vertical scale up fashion	Deployed in horizontal scale out fashion

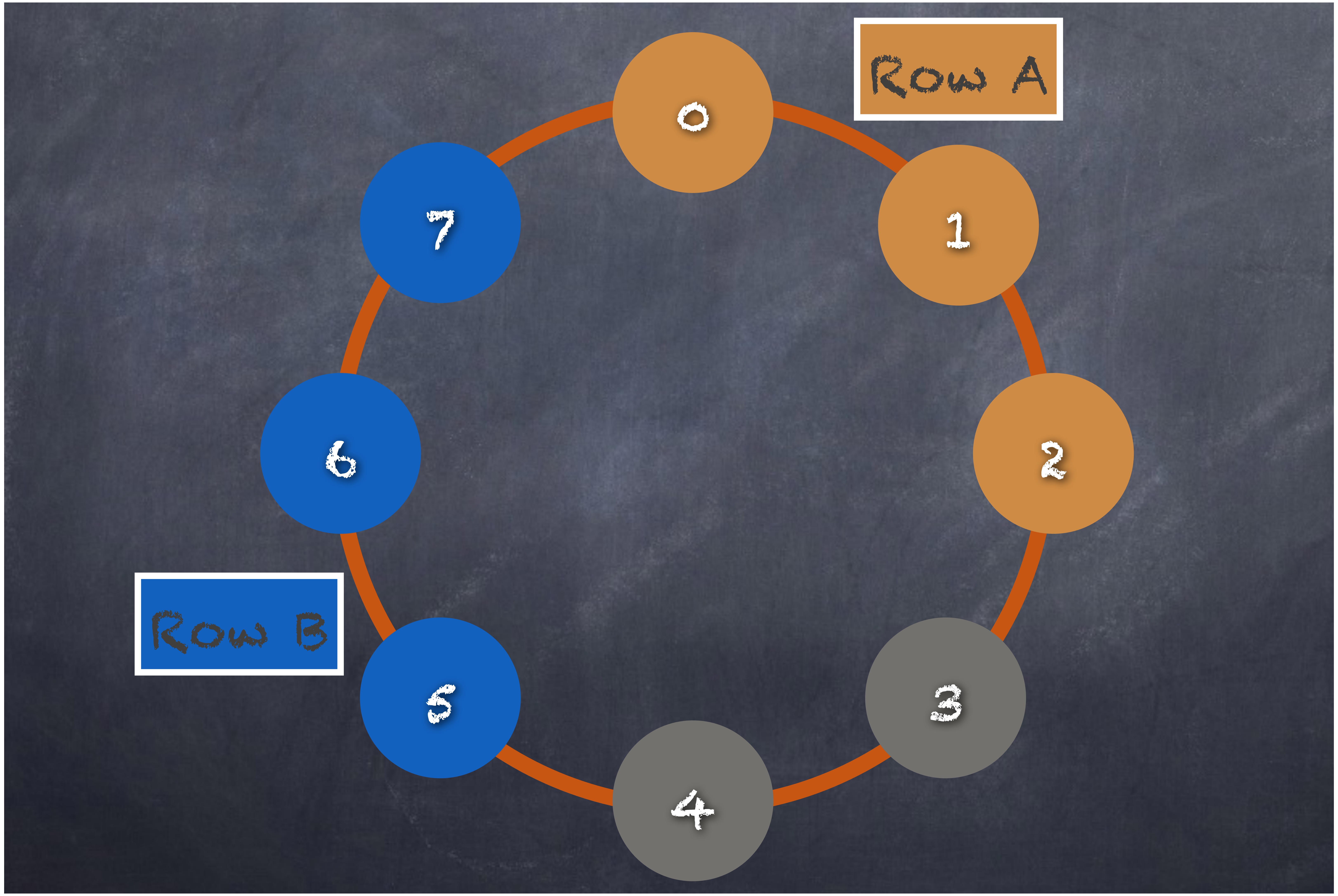
[DataStax]

Cassandra and CAP



[G. Atil]

Cassandra: Replication



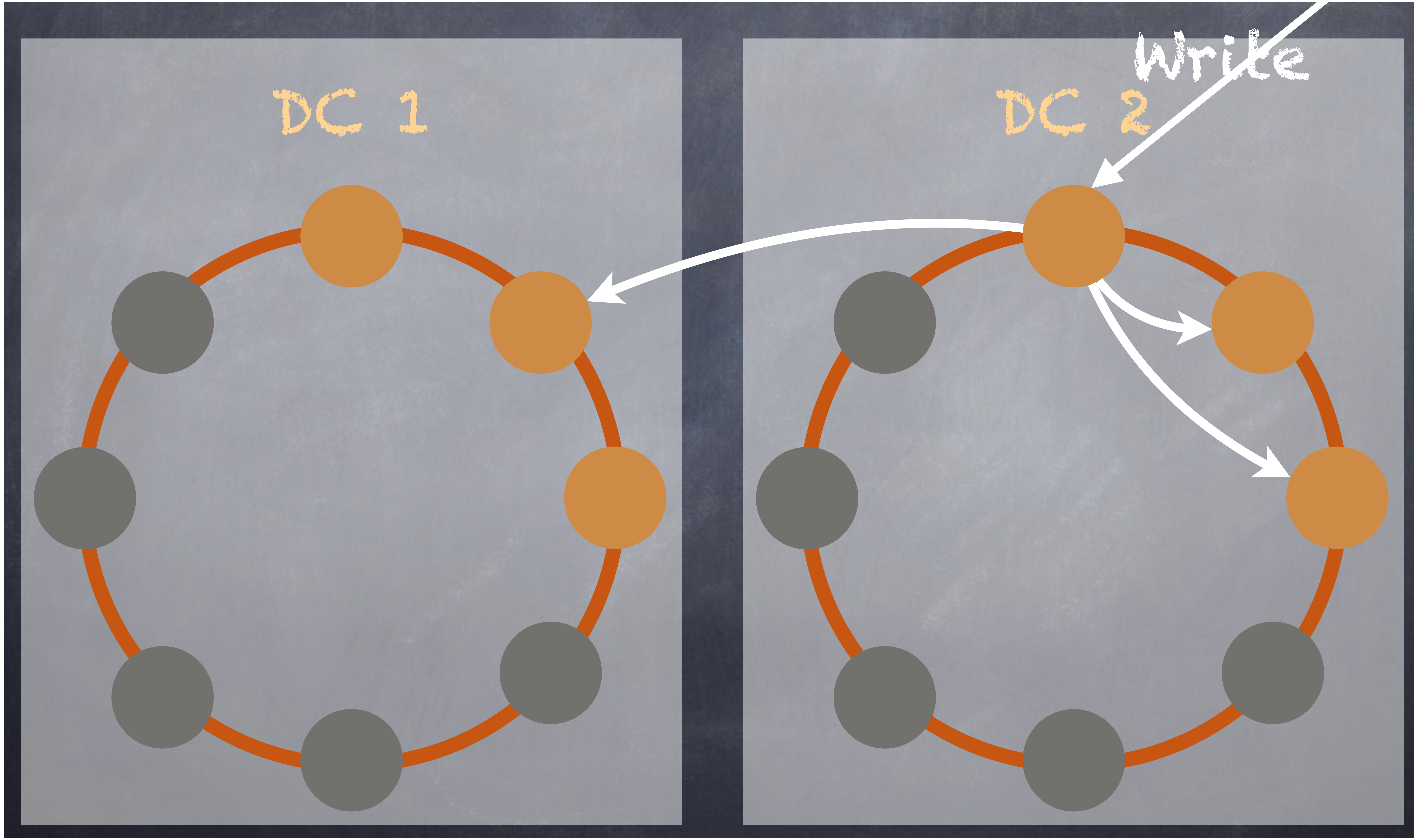
[R. Stupp]

Cassandra: Consistency Levels

- Data is always replicated according to replication factors
- Consistency Levels: ANY (only writes), ONE, LOCAL_ONE, QUORUM, LOCAL_QUORUM
- Consistency levels defines how many replicas must fulfill the request
- LOCAL_* are local to the data center, others go across data centers
- quorum = (sum-of-replication-factors / 2) + 1
  - Each data center may have its own replication factor
- ANY provides lowest consistency but highest availability
- ALL provides the highest consistency and lowest availability (not recommended)

[R. Stupp]

Multiple Data Center Replication



[R. Stupp]

NewSQL

A. Pavlo

Recent History in Databases

- Early 2000s: Commercial DBs dominated, Open-source DBs missing features
- Mid 2000s: MySQL adopted by web companies
- Late 2000s: NoSQL does scale horizontally out of the box
- Early 2010s: New DBMSs that can scale across multiple machines natively and provide ACID guarantees

[A. Pavlo]

NewSQL

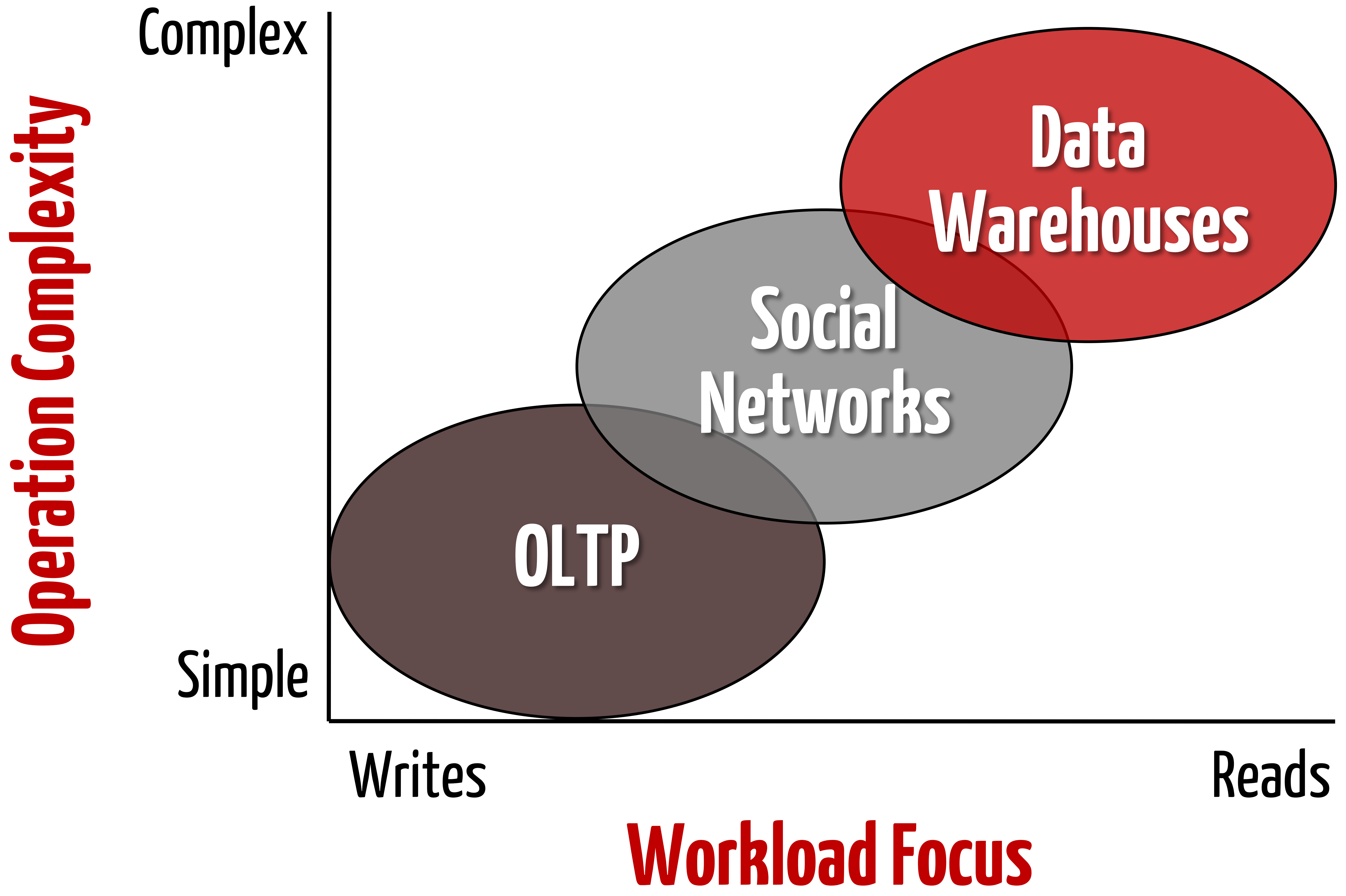
[A. Pavlo]

NewSQL

- 451 Group's Definition:
  - A DBMS that delivers the scalability and flexibility promised by NoSQL while retaining the support for SQL queries and/or ACID, or to improve performance for appropriate workloads.
- Stonebraker's Definition:
  - SQL as the primary interface
  - ACID support for transactions
  - Non-locking concurrency control
  - High per-node performance
  - Parallel, shared-nothing architecture

[A. Pavlo]

OLTP Workload



[A. Pavlo]

Ideal OLTP System

- Main Memory Only
- No Multi-processor Overhead
- High Scalability
- High Availability
- Autonomic Configuration

[A. Pavlo]

What's Really New with NewSQL?

A. Pavlo & M. Aslett

The Official Ten-Year Retrospective of NewSQL

A. Pavlo

Three Types of NewSQL Systems

- New Architectures
  - New codebase without architectural baggage of legacy systems
  - Examples: VoltDB, Spanner, Clustrix
- Transparent Sharding Middleware:
  - Transparent data sharding & query redirecting over cluster of single-node DBMSs
  - Examples: citusdata, ScaleArc (usually support MySQL/postgres wire)
- Database-as-a-Service:
  - Distributed architecture designed specifically for cloud-native deployment
  - Examples: xeround, GenieDB, FathomDB (usually based on MySQL)

[A. Pavlo]

What went wrong?

- Almost every NewSQL company from the last decade has closed, sold for scraps, or pivoted to other markets
- Why?
  - Selling an OLTP Database System is hard
  - Startup cost of a relational system is harder than NoSQL
  - Existing DBMS Systems (MySQL, postgresql) are Good
  - Cloud Disruption
    - Can't sell on-premises
    - Can't complete on cost with cloud vendors
  - Lack of Open Source

[A. Pavlo]

NewSQL is dead, Long live Distributed SQL

- E.g., Cockroach
- Core concepts are similar to earlier systems

[A. Pavlo]

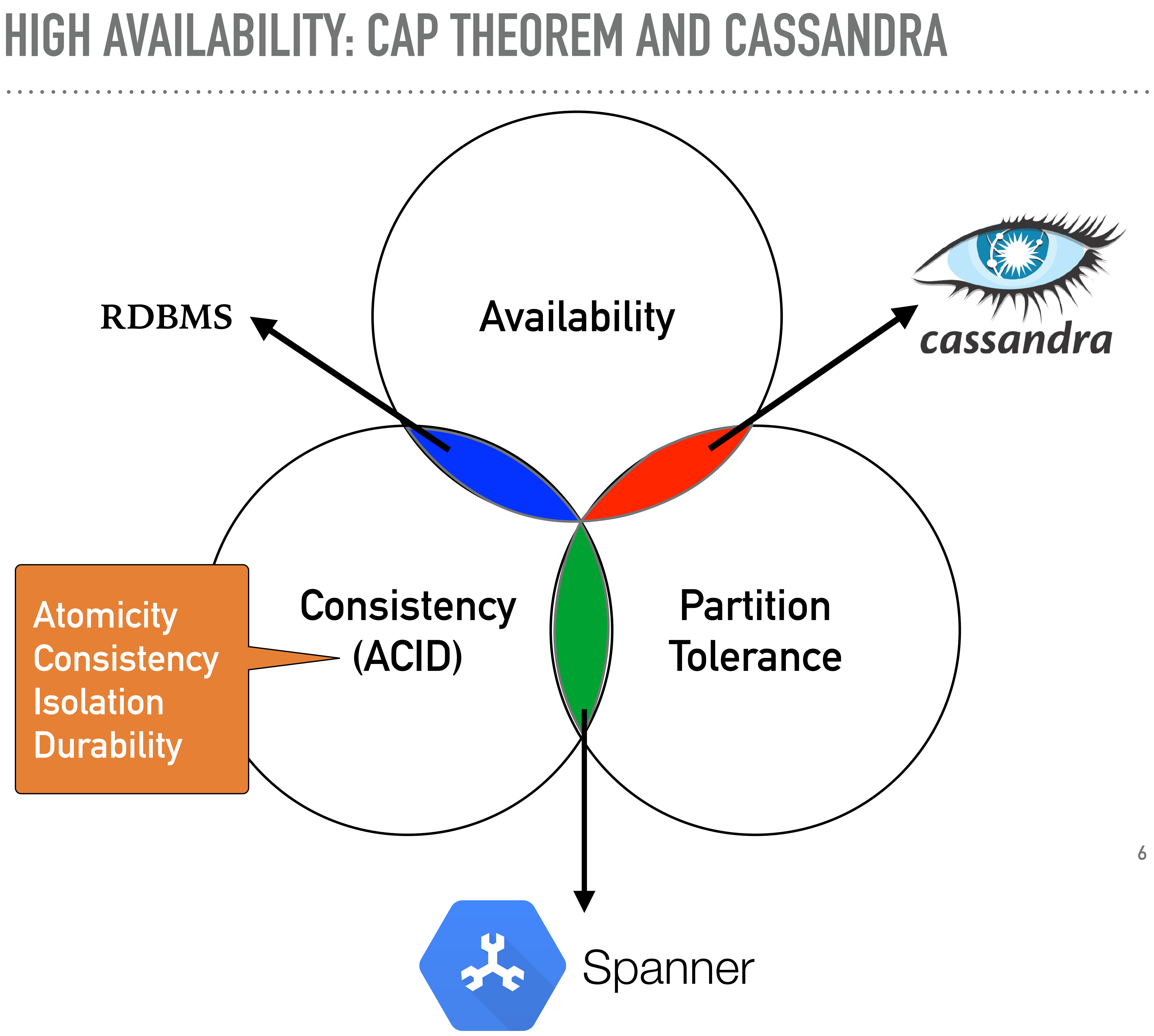
Spanner: Google's Globally-Distributed Database

J. C. Corbett et al.

Spanner Overview

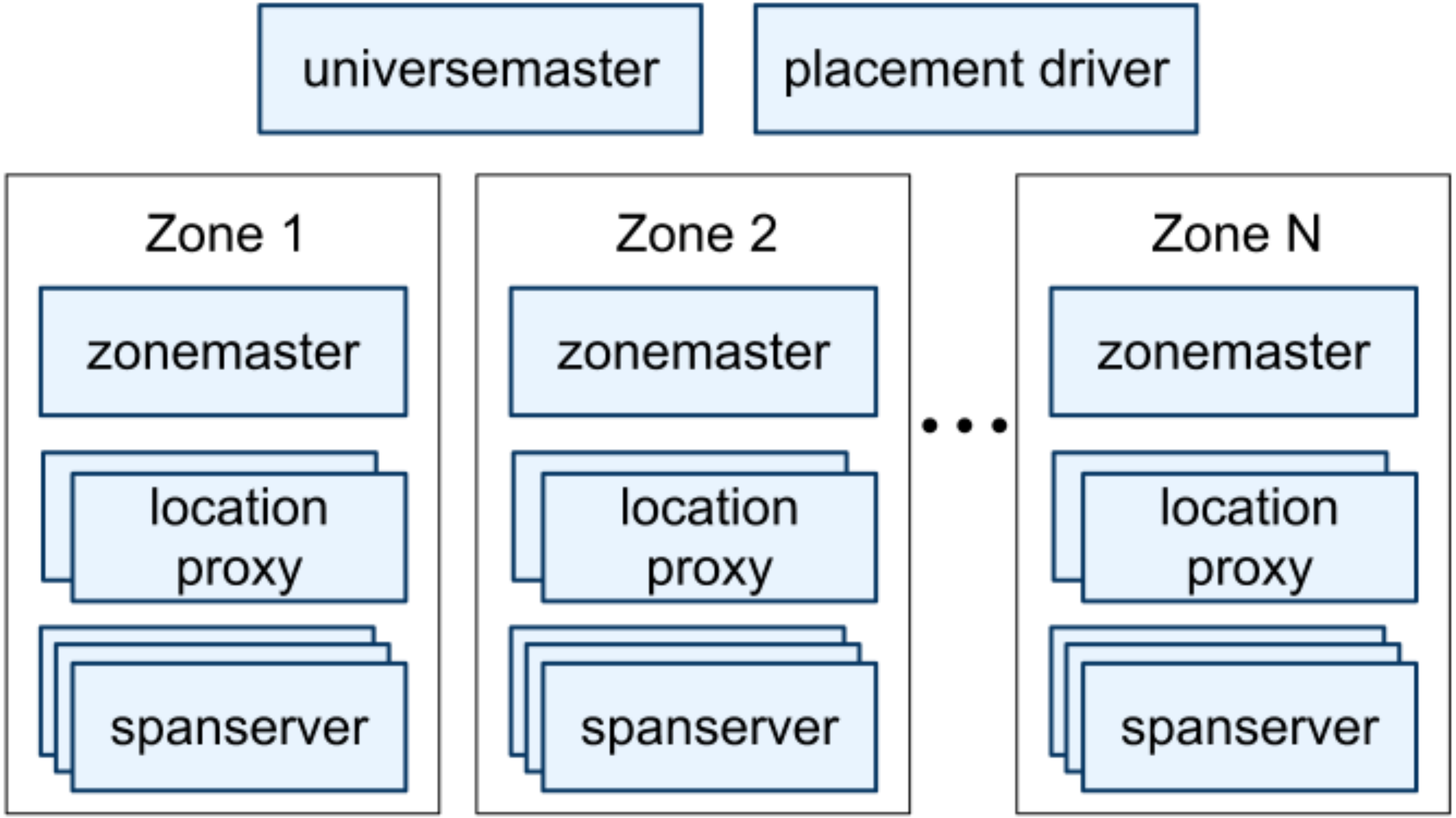
- Focus on scaling databases focused on OLTP (not OLAP)
- Since OLTP, focus is on sharding **rows**
- Tries to satisfy CAP (which is impossible per CAP Theorem) by not worrying about 100% availability
- External consistency using multi-version concurrency control through timestamps
- ACID is important
- Structured: universe with zones with zone masters and then spans with span masters
- SQL-like (updates allow SQL to be used with Spanner)

Spanner and the CAP Theorem



- Which type of system is Spanner?
  - C: consistency, which implies a single value for shared data
  - A: 100% availability, for both reads and updates
  - P: tolerance to network partitions
- Which two?
  - CA: close, but not totally available
  - So actually **CP**

Spanner Server Organization

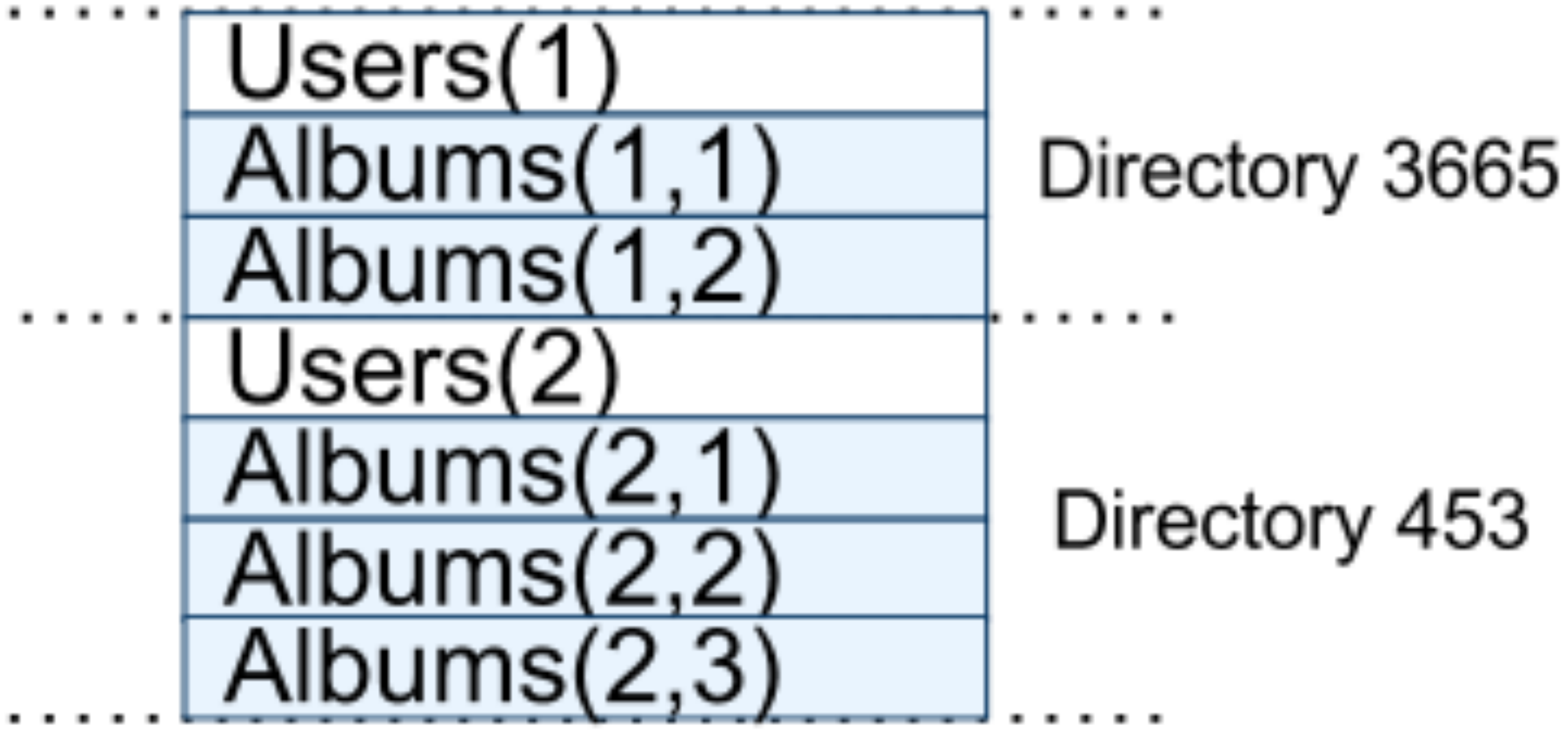


[Corbett et al., 2012]

Interleaved Schema

```
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```



[Corbett et al., 2012]

External Consistency

- Traditional DB solution: **two-phase locking**—no writes while client reads
- "The system behaves as if all transactions were executed sequentially, even though Spanner actually runs them across multiple servers (and possibly in multiple datacenters) for higher performance and availability" [Google]
- Semantically indistinguishable from a single-machine database
- Uses multi-version concurrency control (MVCC) using **timestamps**
- Spanner uses **TrueTime** to generate monotonically increasing timestamps across all nodes of the system

TrueTime

- API to try to keep computers on a globally-consistent clock
- Uses GPS and Atomic Clocks!
- Time masters per datacenter (usually with GPS)
- Each machine runs a timeslave daemon
- Armageddon masters have atomic clocks
- API:

Method	Returns
$TT.now()$	$TTinterval$: $[earliest, latest]$
$TT.after(t)$	true if t has definitely passed
$TT.before(t)$	true if t has definitely not arrived

[Corbett et al., 2012]

Concurrency Control

- Use TrueTime to implement concurrency control
- Types of reads and writes:

Operation	Timestamp Discussion	Concurrency Control	Replica Required
Read-Write Transaction	§ 4.1.2	pessimistic	leader
Read-Only Transaction	§ 4.1.4	lock-free	leader for timestamp; any for read, subject to § 4.1.3
Snapshot Read, client-provided timestamp	—	lock-free	any, subject to § 4.1.3
Snapshot Read, client-provided bound	§ 4.1.3	lock-free	any, subject to § 4.1.3

- Use Two-Phase Commits (2PC)

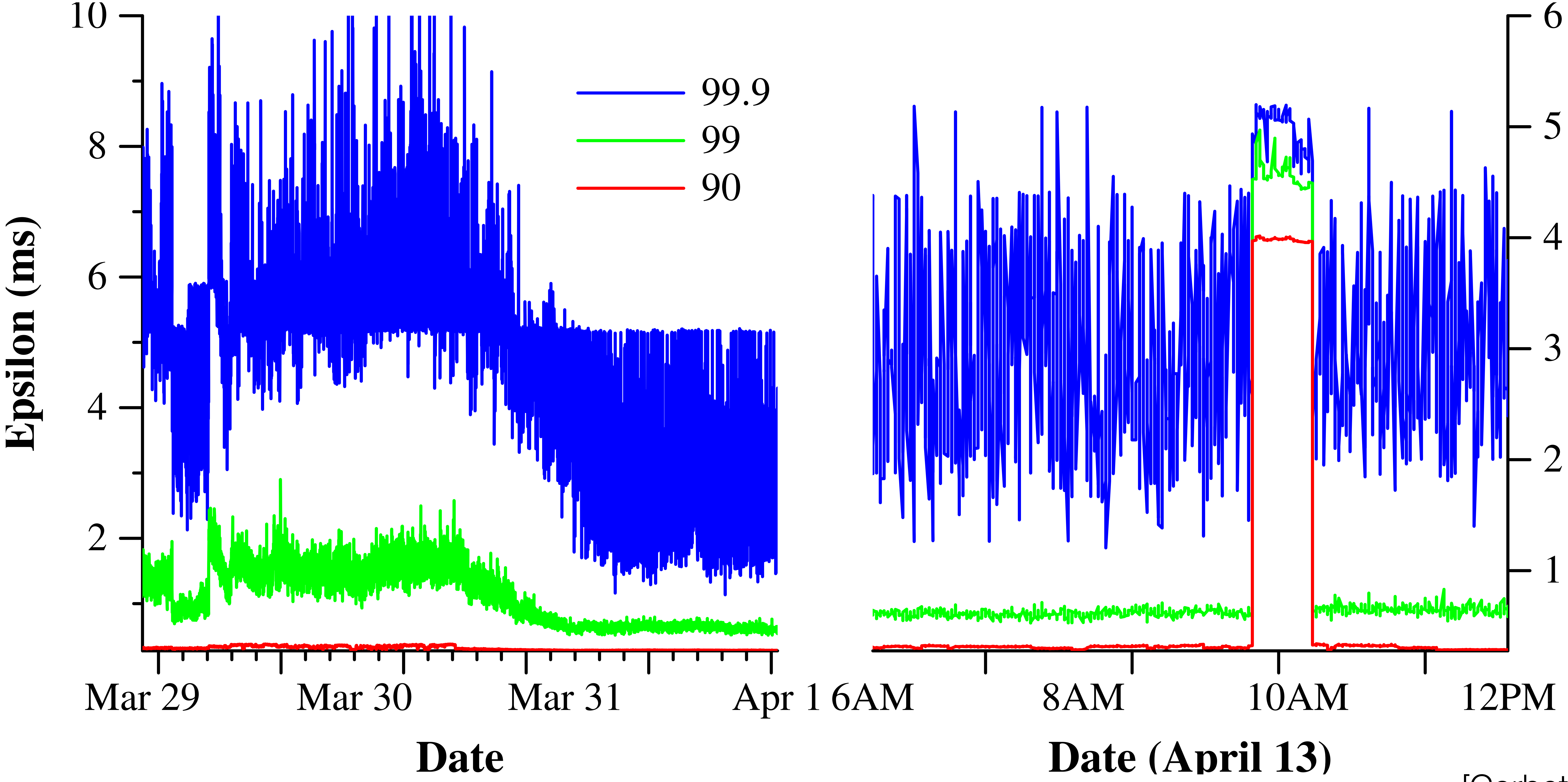
[Corbett et al., 2012]

Two-Phase Commit Scalability

participants	latency (ms)	
	mean	99th percentile
1	17.0 ±1.4	75.0 ±34.9
2	24.5 ±2.5	87.6 ±35.9
5	31.5 ±6.2	104.5 ±52.2
10	30.0 ±3.7	95.6 ±25.4
25	35.5 ±5.6	100.4 ±42.7
50	42.7 ±4.1	93.7 ±22.9
100	71.4 ±7.6	131.2 ±17.6
200	150.5 ±11.0	320.3 ±35.1

[Corbett et al., 2012]

Distribution of TrueTime Epsilons



[Corbett et al., 2012]

F1: A Distributed SQL Database That Scales

J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey,
E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner,
J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte

F1: OLTP and OLAP Together

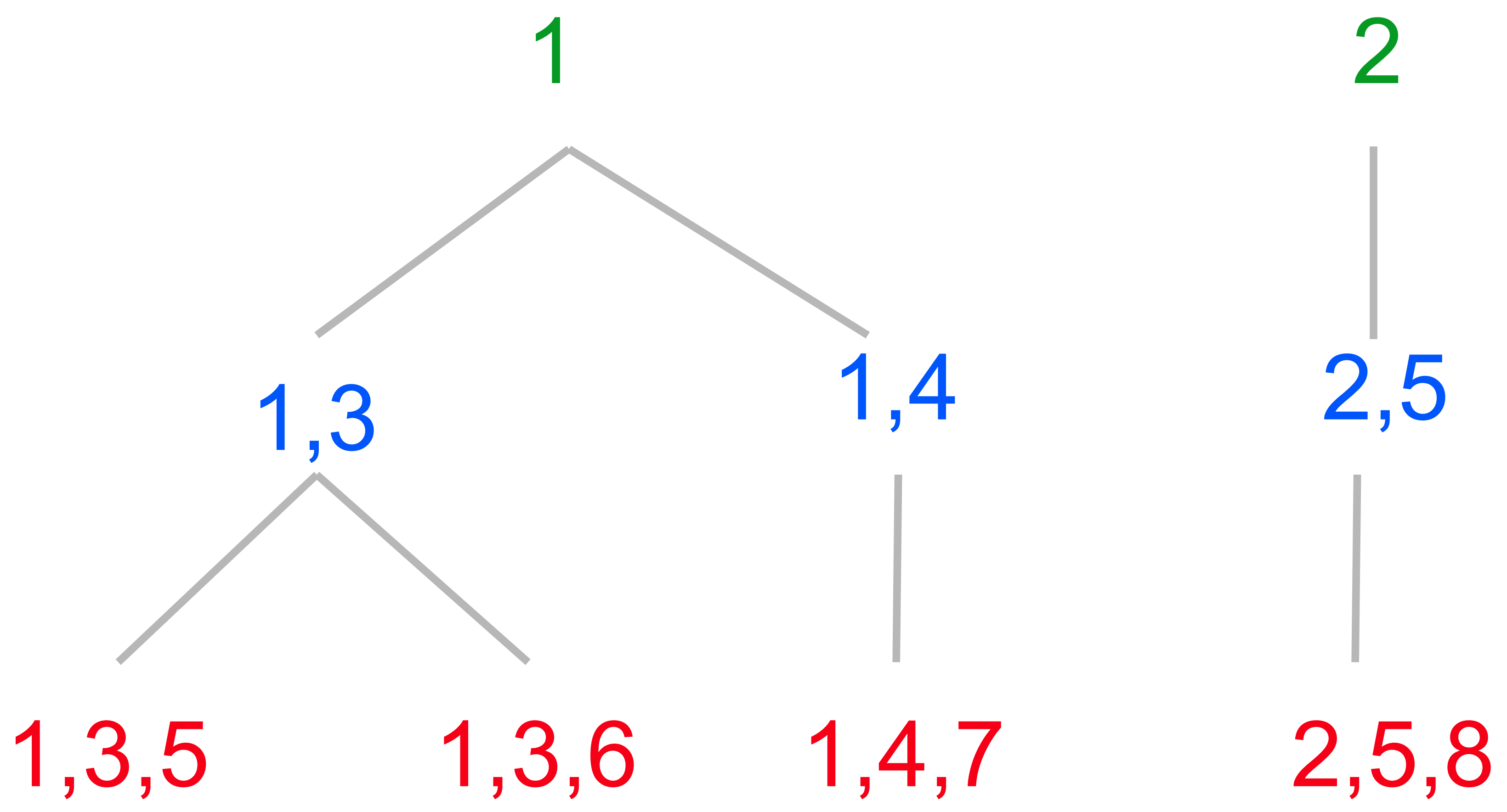
- Distributed data storage: data is not stored at one central location
- Need to keep data and schemas in sync
- Hierarchical schemas keep data that is likely to be accessed at the same time together
- Optimistic Transactions: Long reads that keep track of timestamps and don't lock the database until the write happens
- Change History: Keep track of history with database, also helps with caching
- DIY Object-Relational Mapping: don't automatically join or implicitly traverse relationships
- Protocol buffers as a way to store application data without translation + support for queries

Hierarchical Schema

Explicit table hierarchies. Example:

- Customer (root table): PK (CustomerId)
- Campaign (child): PK (CustomerId, CampaignId)
- AdGroup (child): PK (CustomerId, CampaignId, AdGroupId)

Rows and PKs



Storage Layout

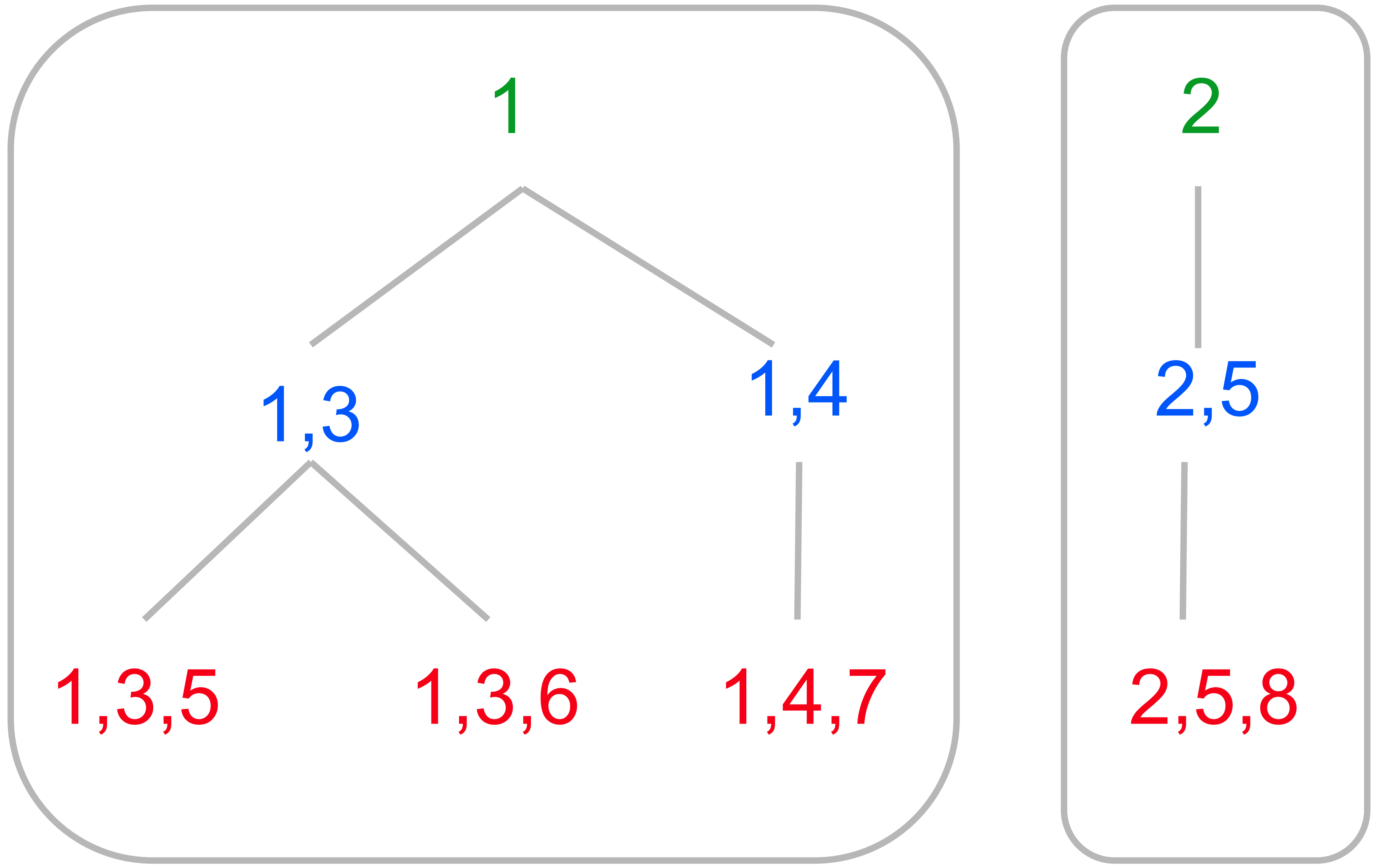
```
Customer (1)
Campaign (1,3)
AdGroup  (1,3,5)
AdGroup  (1,3,6)
Campaign (1,4)
AdGroup  (1,4,7)
Customer (2)
Campaign (2,5)
AdGroup  (2,5,8)
```

[Shute et al., 2012]

Clustered Storage

- Child rows under one root row form a **cluster**
- Cluster stored on one machine (unless huge)
- Transactions within one cluster are most efficient
- Very efficient joins inside clusters (can merge with no sorting)

Rows and PKs



Storage Layout

```
Customer (1)
Campaign (1,3)
AdGroup  (1,3,5)
AdGroup  (1,3,6)
Campaign (1,4)
AdGroup  (1,4,7)
```

```
Customer (2)
Campaign (2,5)
AdGroup  (2,5,8)
```

[Shute et al., 2012]

F1 Notes

- Schema changes: allow two different schemas
- Transaction types: Snapshot, Pessimistic, Optimistic
- Change History and application to caching
- Disk latency or network latency?

Google Cloud Spanner

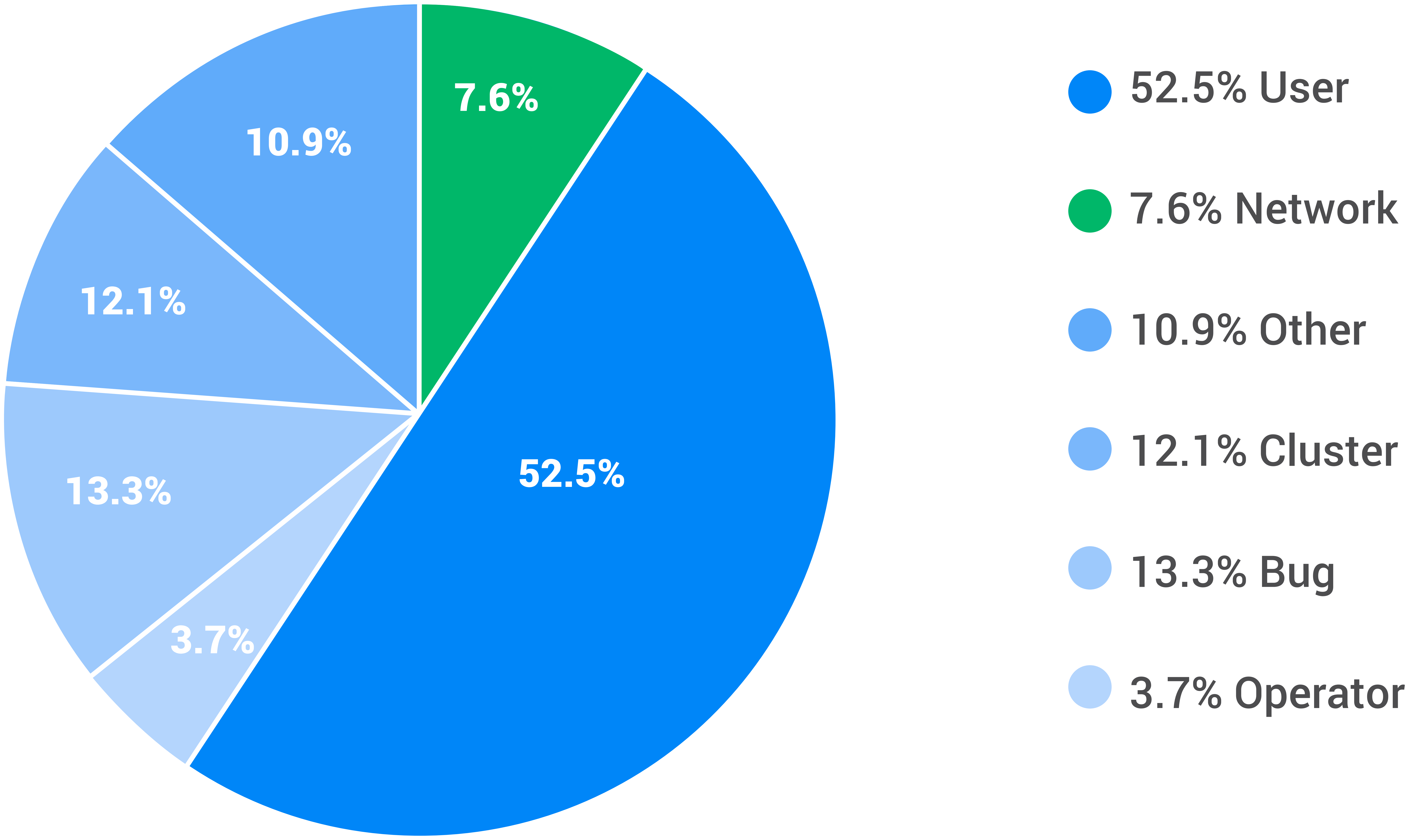
- https://cloud.google.com/spanner/
- Features:
  - Global Scale: thousands of nodes across regions / data centers
  - Fully Managed: replication and maintenance are automatic
  - Transactional Consistency: global transaction consistency
  - Relational Support: Schemas, ACID Transactions, SQL Queries
  - Security
  - Highly Available

Google Cloud Spanner: NewSQL

	CLOUD SPANNER	TRADITIONAL RELATIONAL	TRADITIONAL NON-RELATIONAL
Schema	✓ **Yes**	✓ Yes	✗ No
SQL	✓ **Yes**	✓ Yes	✗ No
Consistency	✓ **Strong**	✓ Strong	✗ Eventual
Availability	✓ **High**	✗ Failover	✓ High
Scalability	✓ **Horizontal**	✗ Vertical	✓ Horizontal
Replication	✓ **Automatic**	Configurable	Configurable

[https://cloud.google.com/spanner/]

Causes of Spanner Availability Incidents



[E. Brewer, 2017]

Causes of Spanner Incidents

- User: overload or misconfiguration (specific to one user)
- Cluster: non-network problems, e.g. servers and power
- Operator: misconfiguration by people
- Bug: software error that caused some problem
- Other: most are one-offs
- Network: individual data centers/regions cut off and under-provisioned bandwidth, uni-directional traffic

[E. Brewer, 2017]

Spanner as "Effectively CA"

- Criteria for being "effectively CA"
  1. At a minimum it must have very high availability in practice (so that users can ignore exceptions), and
  2. As this is about partitions it should also have a low fraction of those outages due to partitions.
- Spanner meets both of these criteria
- Spanner relies on Google's **network** (private links between data centers)
- TrueTime helps create **consistent snapshots**, sometimes have a commit wait

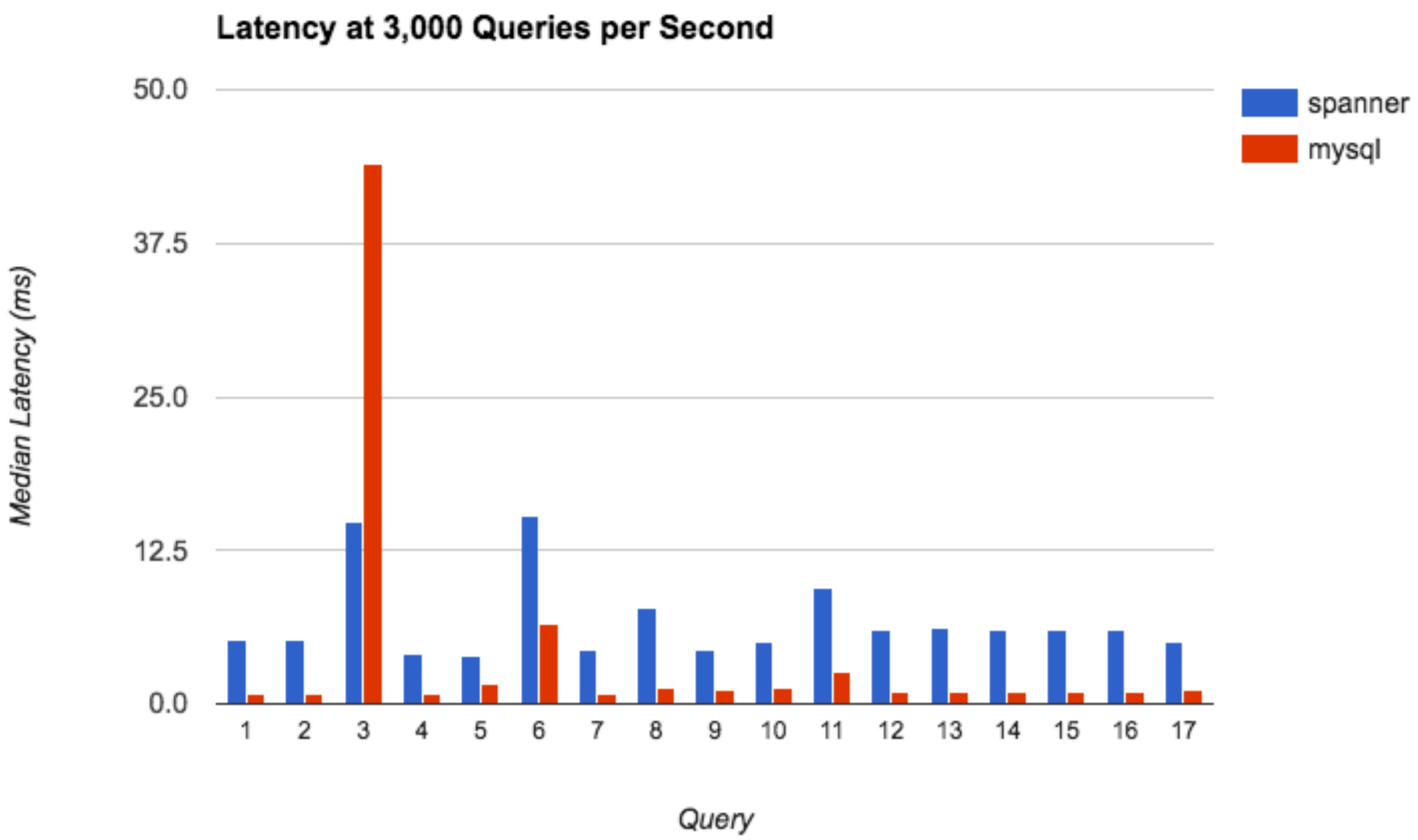
[E. Brewer, 2017]

More Recent Tests: Spanner vs. MySQL

	Frequency	Query
1	0.30%	INSERT INTO `terms` (`term`, `rank`, `set_id`, `last_modified`) VALUES (?,?,?,?),(?,?,?,?)
2	0.25%	INSERT INTO `terms` (`term`, `rank`, `set_id`, `last_modified`, `definition`) VALUES (?,?,?,?,?),(?,?,?,?,?),(?,?,?,?,?),...
3	4.22%	INSERT INTO `terms` (`term`,`rank`,`set_id`,`last_modified`) VALUES (?,?,?,?)
4	1.88%	INSERT INTO `terms` (`term`,`rank`,`set_id`,`last_modified`,`definition`) VALUES (?,?,?,?,?)
5	3.28%	SELECT * FROM `terms` WHERE (`is_deleted` = 0) AND (`set_id` IN (??)) AND (`rank` IN (0,1,2,3)) AND (`term` != '')
6	14.13%	SELECT `set_id`, COUNT(*) FROM `terms` WHERE (`is_deleted` = 0) AND (`set_id` = ?) GROUP BY `set_id`
7	12.56%	SELECT * FROM `terms` WHERE (`id` = ?)
8	0.49%	SELECT * FROM `terms` WHERE (`id` IN (??) AND `set_id` IN (??))
9	4.11%	SELECT `id`, `set_id` FROM `terms` WHERE (`set_id` = ?) LIMIT 20000
10	0.43%	SELECT `id`, `set_id` FROM `terms` WHERE (`set_id` IN (??)) LIMIT 20000
11	0.59%	SELECT * FROM `terms` WHERE (`id` IN (??))
12	36.76%	SELECT * FROM `terms` WHERE (`set_id` = ?)
13	0.61%	SELECT * FROM `terms` WHERE (`set_id` IN (??))
14	6.10%	UPDATE `terms` SET `definition`=?, `last_modified`=? WHERE `id`=? AND `set_id`=?
15	0.33%	UPDATE `terms` SET `is_deleted`=?, `last_modified`=? WHERE `id` IN (??) AND `set_id`=??
16	12.56%	UPDATE `terms` SET `rank`=?, `last_modified`=? WHERE `id`=? AND `set_id`=?
17	1.06%	UPDATE `terms` SET `word`=?, `last_modified`=? WHERE `id`=? AND `set_id`=?
18	0.32%	UPDATE `terms` SET `definition`=?, `word`=?, `last_modified`=? WHERE `id`=? AND `set_id`=?

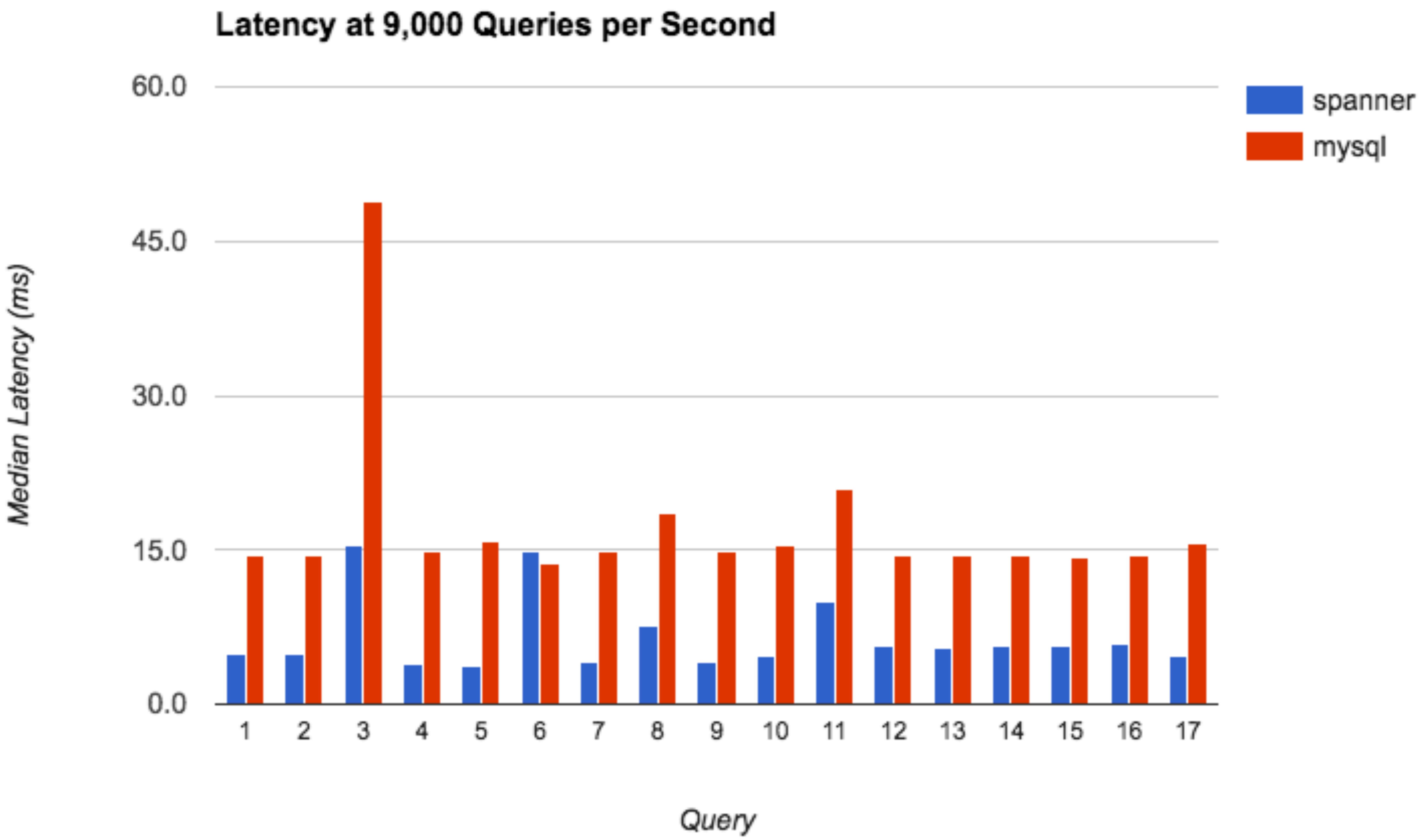
[P. Bakkum and D. Cepeda, 2017]

Latency: Spanner vs. MySQL



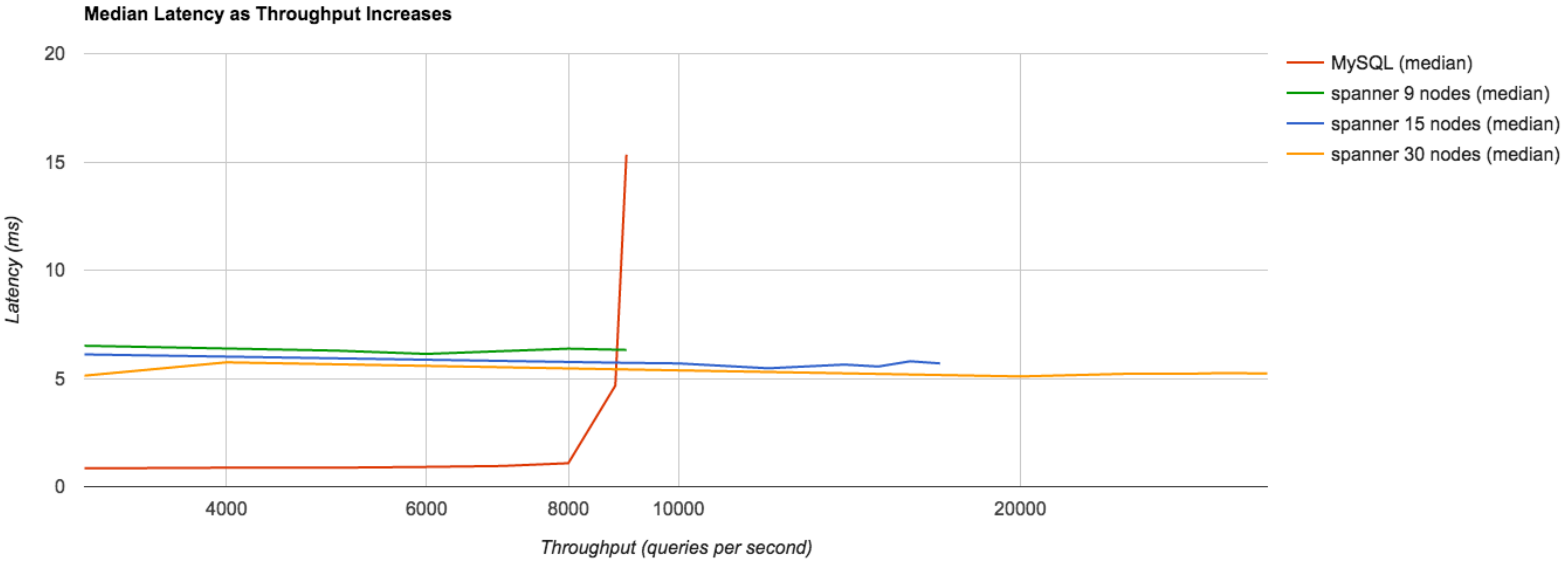
[P. Bakkum and D. Cepeda, 2017]

Latency: Spanner vs. MySQL



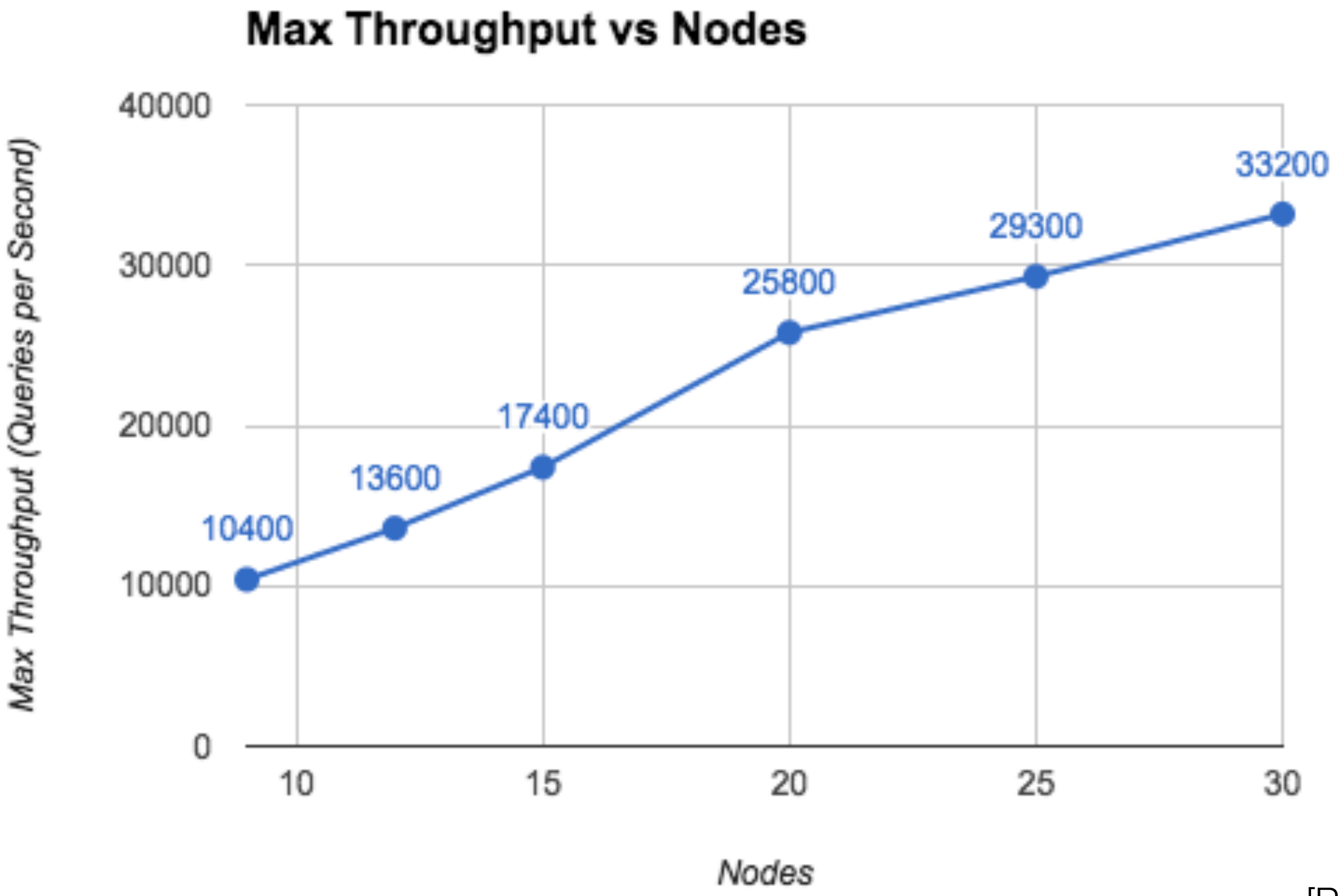
[P. Bakkum and D. Cepeda, 2017]

Throughput: Spanner vs. MySQL



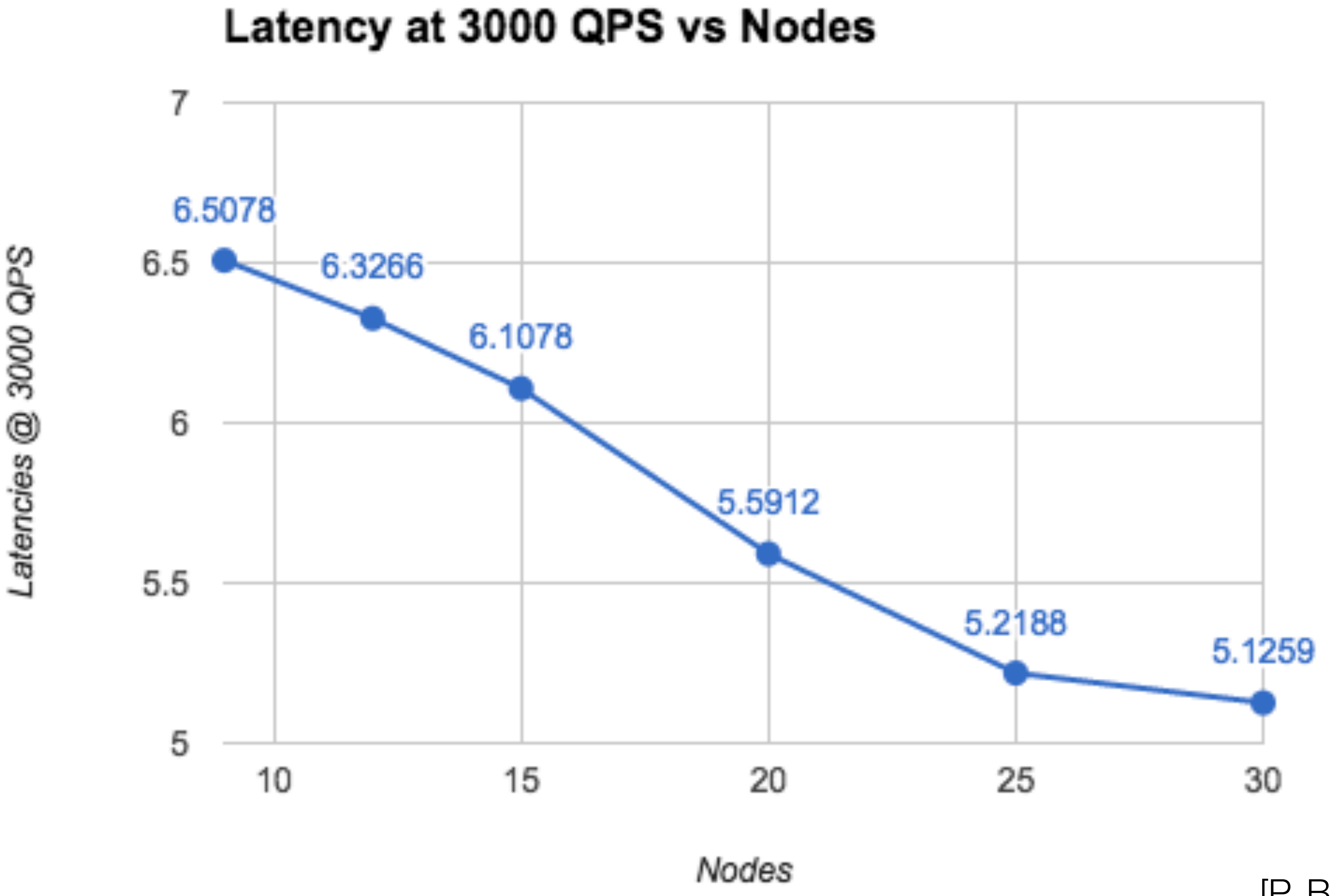
[P. Bakkum and D. Cepeda, 2017]

Max Throughput vs. Nodes



[P. Bakkum and D. Cepeda, 2017]

Spanner: Latency vs. Nodes



[P. Bakkum and D. Cepeda, 2017]