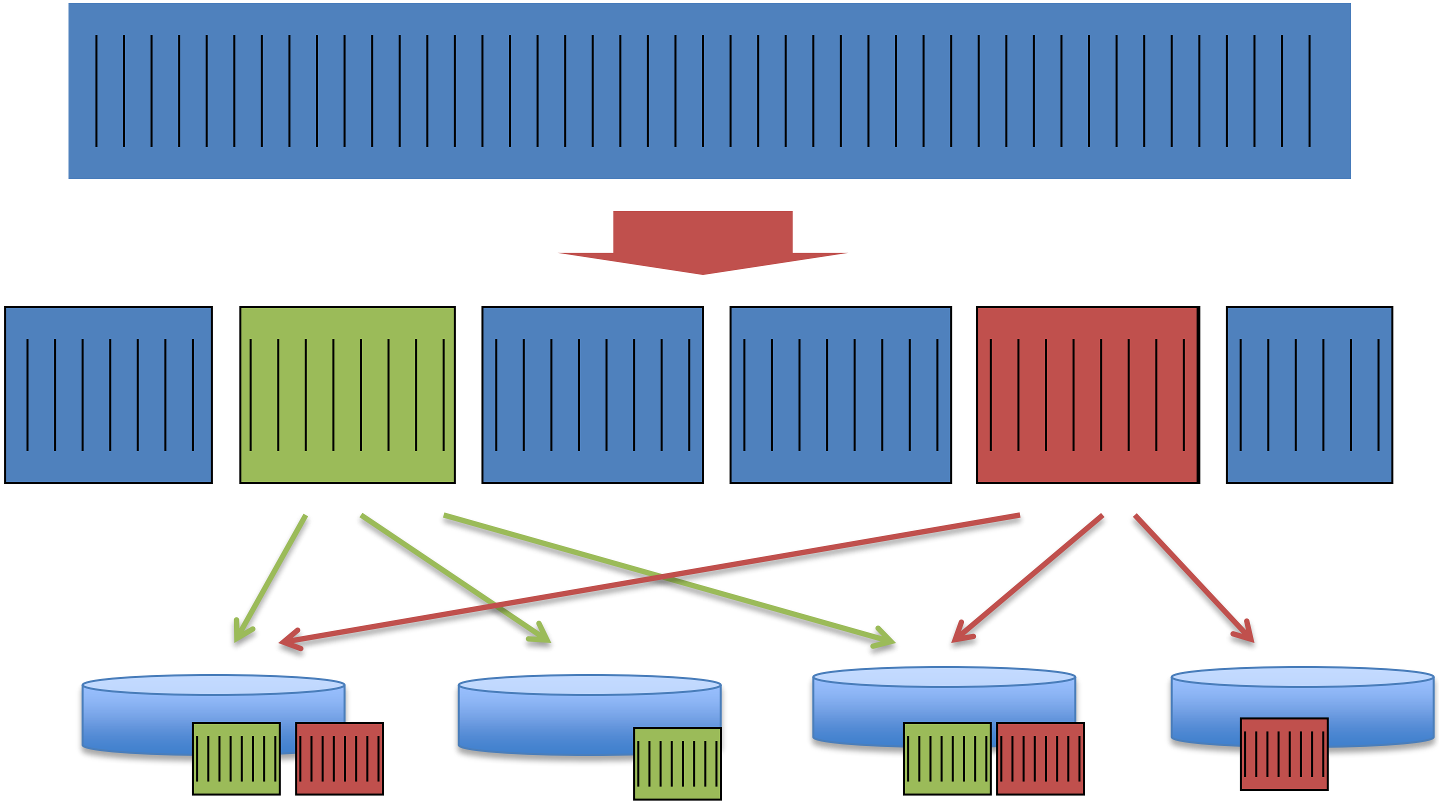


1) We need to ensure high availability

2) We also want to support updates

Example

User: Joe
Friends: Sue, …
Status: “I’m sleepy”
Wall: “…”, “…”

User: Sue
Friends: Joe, Kai, …
Status: “Headed to new Bond flick”
Wall: “…”, “…”

User: Kai
Friends: Sue, …
Status: “Done for tonight”
Wall: “…”, “…”

Write: Update Sue’s status. Who sees the new status, and who sees the old one?

Databases: *“Everyone MUST see the same thing, either old or new, no matter how long it takes.”*

NoSQL: *“For large applications, we can’t afford to wait that long, and maybe it doesn’t matter anyway”*

Example

Friends

Jim, Sue
Sue, Jim
Lin, Joe
Joe, Lin
Jim, Kai
Kai, Jim
Jim, Lin
Lin, Jim

Users

Jim

Sue

…

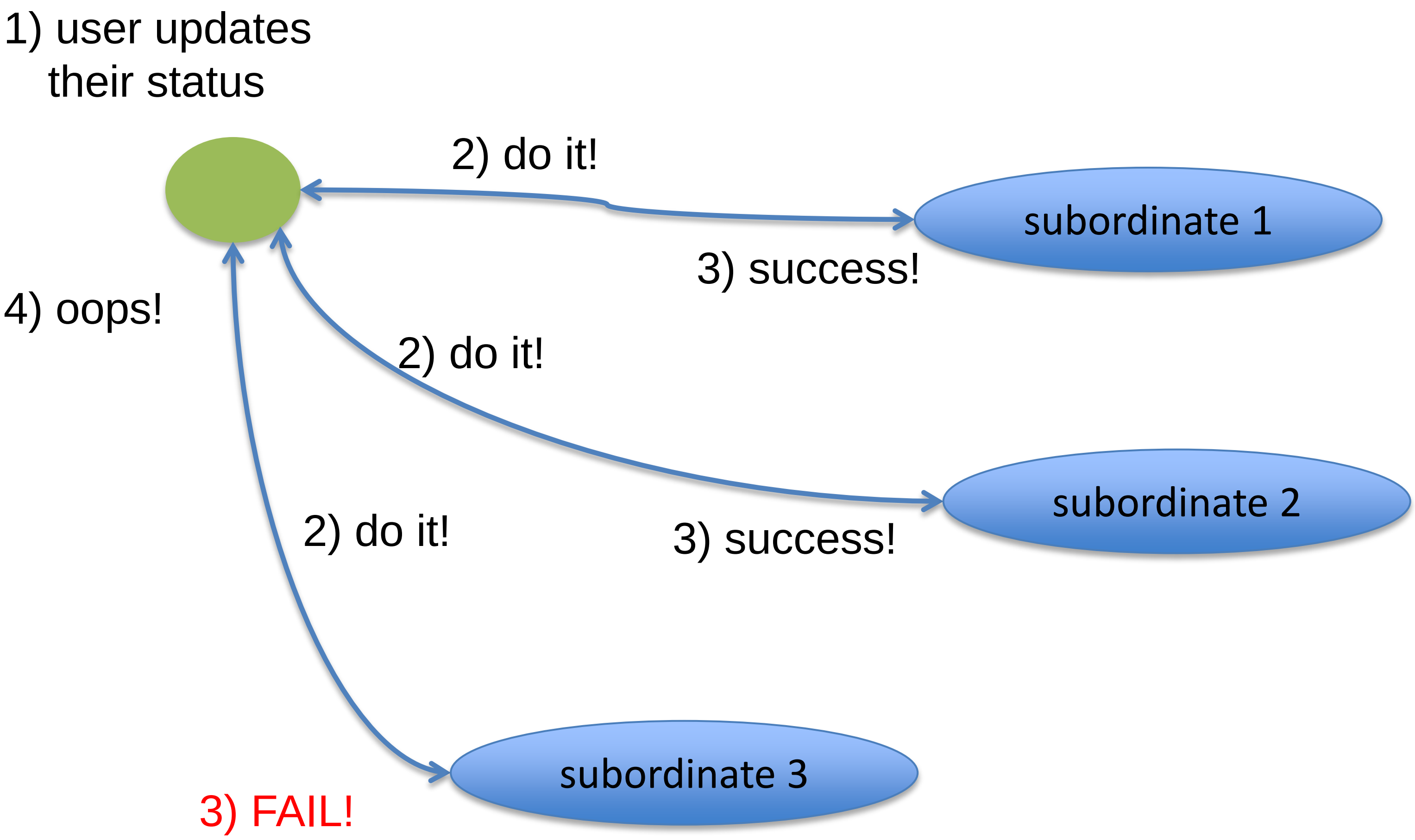
Posts

Sue: “headed to see new Bond flick”

Sue: “it was ok”

Kai: “I’m hungry”

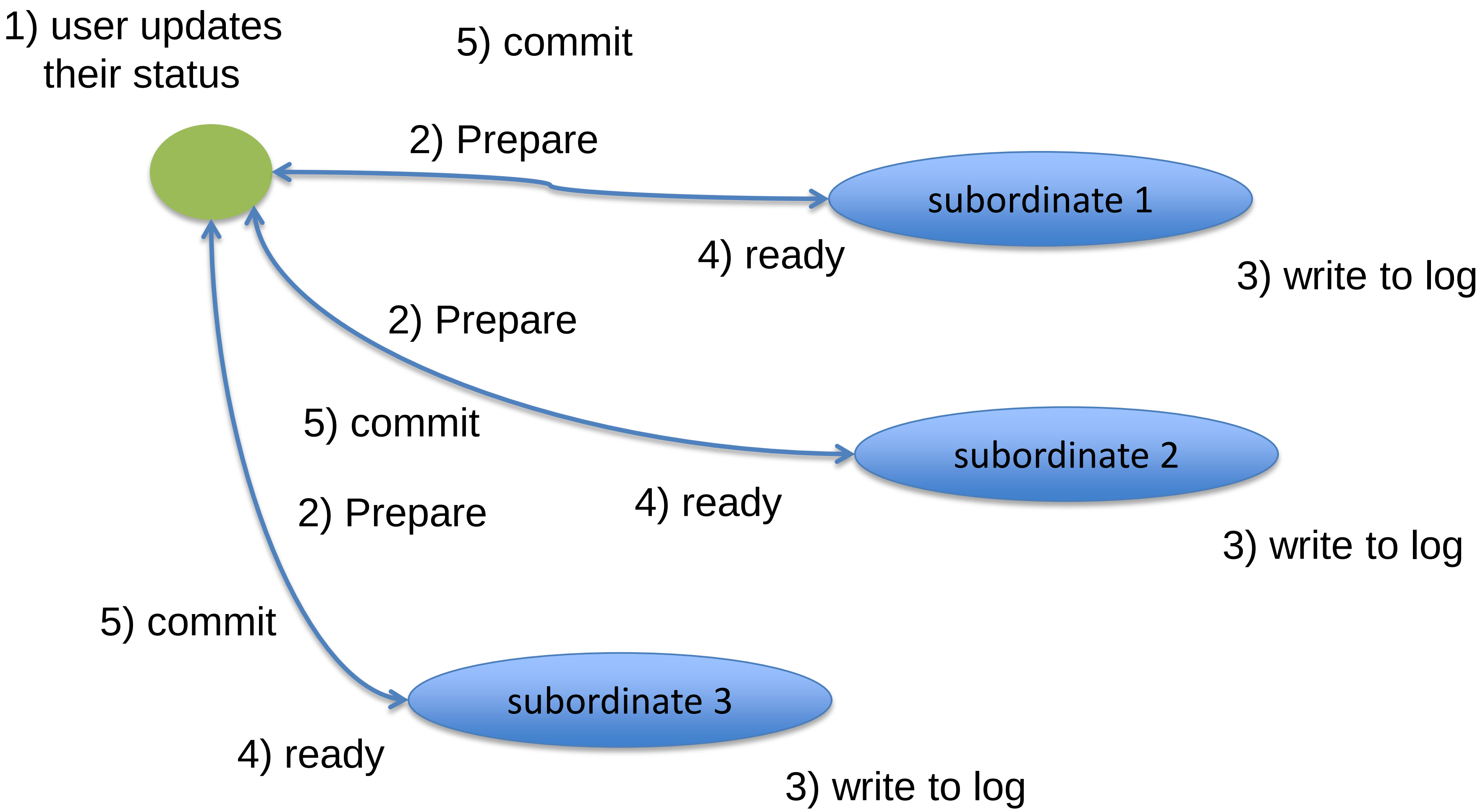
Two-Phase Commit Motivation



Two-Phase Commit

- Phase 1:
  - Coordinator Sends “Prepare to Commit”
  - Subordinates make sure they can do so no matter what
    - Write the action to a log to tolerate failure
  - Subordinates Reply “Ready to Commit”
- Phase 2:
  - If all subordinates ready, send “Commit”
  - If anyone failed, send “Abort”

Two-Phase Commit



1) user updates their status

5) commit

2) Prepare

subordinate 1

4) ready

3) write to log

2) Prepare

5) commit

subordinate 2

2) Prepare

4) ready

3) write to log

5) commit

subordinate 3

4) ready

3) write to log

“Eventual Consistency”

- Write conflicts will eventually propagate throughout the system
  - D. Terry et al., “Managing Update Conflicts in Bayou,a Weakly Connected Replicated Storage System”, SOSP 1995

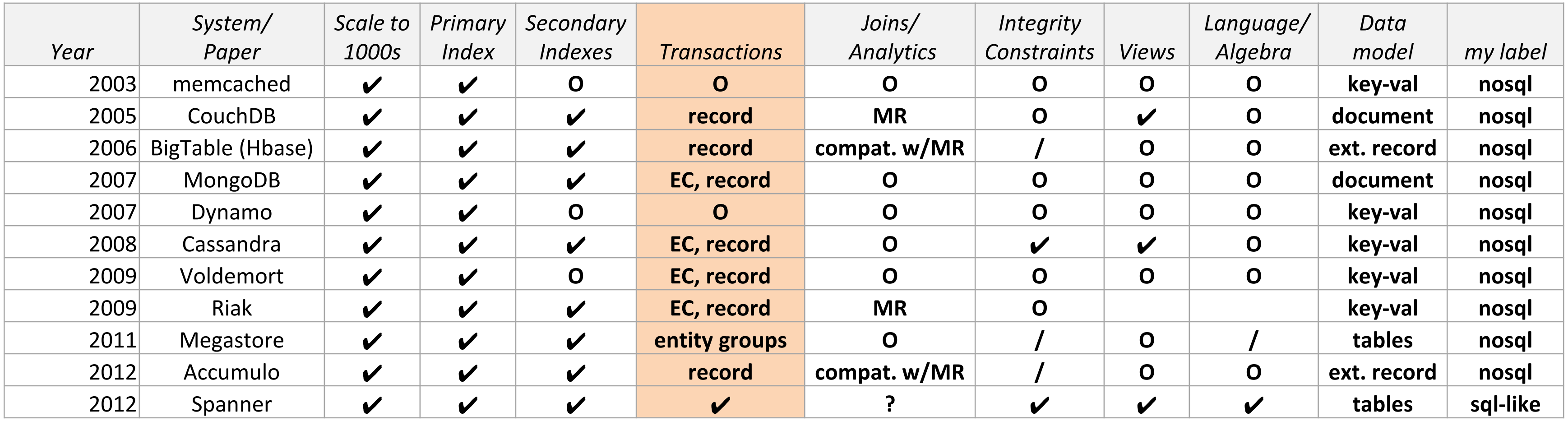
“We believe that applications must be aware that they may read weakly consistent data and also that their write operations may conflict with those of other users and applications.”

“Moreover, applications must be revolved m the detection and resolution of conflicts since these naturally depend on the semantics of the application.”

Eventual Consistency

- What the application sees in the meantime is sensitive to replication mechanics and difficult to predict

- Contrast with RDBMS, Paxos: Immediate (or “strong”) consistency, but there may be deadlocks

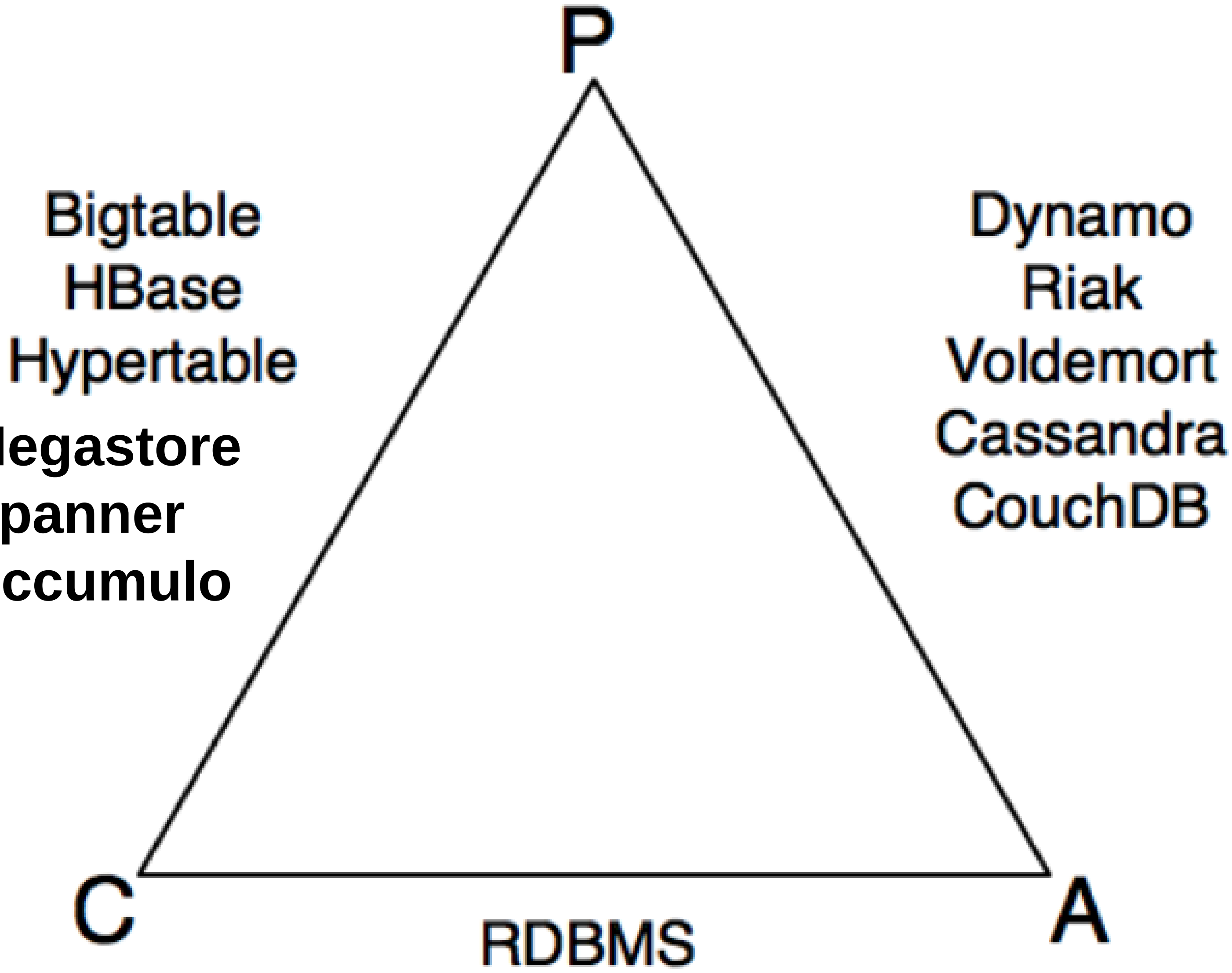


Year	*System/ Paper*	*Scale to 1000s*	*Primary Index*	*Secondary Indexes*	*Transactions*	*Joins/ Analytics*	*Integrity Constraints*	*Views*	*Language/ Algebra*	*Data model*	*my label*
2003	memcached	✔	✔	**O**	**O**	**O**	**O**	**O**	**O**	**key-val**	**nosql**
2005	CouchDB	✔	✔	✔	**record**	**MR**	**O**	✔	**O**	**document**	**nosql**
2006	BigTable (Hbase)	✔	✔	✔	**record**	**compat. w/MR**	**/**	**O**	**O**	**ext. record**	**nosql**
2007	MongoDB	✔	✔	✔	**EC, record**	**O**	**O**	**O**	**O**	**document**	**nosql**
2007	Dynamo	✔	✔	**O**	**O**	**O**	**O**	**O**	**O**	**key-val**	**nosql**
2008	Cassandra	✔	✔	✔	**EC, record**	**O**	✔	✔	**O**	**key-val**	**nosql**
2009	Voldemort	✔	✔	**O**	**EC, record**	**O**	**O**	**O**	**O**	**key-val**	**nosql**
2009	Riak	✔	✔	✔	**EC, record**	**MR**	**O**			**key-val**	**nosql**
2011	Megastore	✔	✔	✔	**entity groups**	**O**	**/**	**O**	**/**	**tables**	**nosql**
2012	Accumulo	✔	✔	✔	**record**	**compat. w/MR**	**/**	**O**	**O**	**ext. record**	**nosql**
2012	Spanner	✔	✔	✔	✔	**?**	✔	✔	✔	**tables**	**sql-like**

CAP Theorem [Brewer 2000, Lynch 2002]

- Consistency
  - Do all applications see all the same data?
- Availability
  - If some nodes fail, does everything still work?
- Partitioning
  - If two sections of your system cannot talk to each other, can they make forward progress on their own?
    - If not, you sacrifice Availability
    - If so, you might have to sacrific Consistency – can't have everything

- Conventional databases assume no partitioning – clusters were assumed to be small and local
- NoSQL systems may sacrifice consistency



Megastore
Spanner
Accumulo

src: Shashank Tiwari

Year	*System/ Paper*	*Scale to 1000s*	*Primary Index*	*Secondary Indexes*	*Transactions*	*Joins/ Analytics*	*Integrity Constraints*	*Views*	*Language/ Algebra*	*Data model*	*my label*
1971	RDBMS	O	✔	✔	✔	✔	✔	✔	✔	**tables**	**sql-like**
2003	memcached	✔	✔	O	O	O	O	O	O	**key-val**	**nosql**
2004	MapReduce	✔	O	O	O	✔	O	O	O	**key-val**	**batch**
2005	CouchDB	✔	✔	✔	**record**	**MR**	O	✔	O	**document**	**nosql**
2006	BigTable (Hbase)	✔	✔	✔	**record**	**compat. w/MR**	/	O	O	**ext. record**	**nosql**
2007	MongoDB	✔	✔	✔	**EC, record**	O	O	O	O	**document**	**nosql**
2007	Dynamo	✔	✔	O	O	O	O	O	O	**key-val**	**nosql**
2008	Pig	✔	O	O	O	✔	/	O	✔	**tables**	**sql-like**
2008	HIVE	✔	O	O	O	✔	✔	O	✔	**tables**	**sql-like**
2008	Cassandra	✔	✔	✔	**EC, record**	O	✔	✔	O	**key-val**	**nosql**
2009	Voldemort	✔	✔	O	**EC, record**	O	O	O	O	**key-val**	**nosql**
2009	Riak	✔	✔	✔	**EC, record**	**MR**	O			**key-val**	**nosql**
2010	Dremel	✔	O	O	O	/	✔	O	✔	**tables**	**sql-like**
2011	Megastore	✔	✔	✔	**entity groups**	O	/	O	/	**tables**	**nosql**
2011	Tenzing	✔	O	O	O	O	✔	✔	✔	**tables**	**sql-like**
2011	Spark/Shark	✔	O	O	O	✔	✔	O	✔	**tables**	**sql-like**
2012	Spanner	✔	✔	✔	✔	**?**	✔	✔	✔	**tables**	**sql-like**
2012	Accumulo	✔	✔	✔	**record**	**compat. w/MR**	/	O	O	**ext. record**	**nosql**
2013	Impala	✔	O	O	O	✔	✔	O	✔	**tables**	**sql-like**

Rick Cattel’s clustering from
“Scalable SQL and NoSQL Data Stores”
SIGMOD Record, 2010

extensible record stores

document stores

key-value stores