NoSQL: Past, Present, Future

Eric Brewer
Professor, UC Berkeley
VP Infrastructure, Google

QCon SF
November 8, 2012

Charles Bachman, 1973 Turing Award

Integrated Datastore (IDS)
(very) Early “No SQL” database

“Navigational” Database

- Tight integration between code and data
  - Database = linked groups of records (“CODASYL”)
    - Pointers were physical names, today we hash
  - Programmer as “navigator” through the links
  - Similar to DOM engine, WWW, graph DBs
- Used for its high performance, but…
  - But hard to program, maintain
  - Hard to evolve the schema (embedded in code)

Why Relational? (1970s)

- Need a high-level model (sets)
- Separate the data from the code
  - SQL is the (only) API
- Data outlasts any particular implementation
  - because the model doesn't change
- Goal: implement the top-down model well
  - Led to transactions as a tool
  - Declarative language leaves room for optimization

Also 1970s: Unix

"The most important job of UNIX is to provide a file system"
– original 1974 Unix paper

- Bottom-up world view
  - Few, simple, efficient mechanisms
  - Layers and composition
  - "navigational"
  - Evolution comes from APIs, encapsulation

- NoSQL is in this Unix tradition
  - Examples: dbm (1979 kv), gdbm, Berkeley DB, JDBM

Two Valid World Views

Relational View

- Top Down
  - Clean model,
  - ACID Transactions
- Two kinds of developers
  - DB authors
  - SQL programmers
- Values
  - Clean Semantics
  - Set operations
  - Easy long-term evolution
- Venues: SIGMOD, VLDB

Systems View

- Bottom Up
  - Build on top
  - Evolve modules
- One kind of programmer
  - Integrated use
- Values:
  - Good APIs
  - Flexibility
  - Range of possible programs
- Venues: SOSP, OSDI

NoSQL in Context

- Large reusable storage component
- Systems values:
  - Layered, ideally modular APIs
  - Enable a range of systems and semantics
- Some things to build on top over time:
  - Multi-component transactions
  - Secondary indices
  - Evolution story
  - Returning sets of data, not just values

How did I get here…

- Modern cluster-based server (1995)
  - Scalable, highly available, commodity clusters
  - Inktomi search engine (1996), proxy cache (1998)
- But didn't use a DBMS
  - Informix was 10x slower for the search engine
  - Instead, custom servers on top of file systems
- Led to “ACID vs. BASE” spectrum (1997)
  - **B**asically **A**vailable, **S**oft State, **E**ventual Consistency
  - … but BASE was not well received… (ACID was sacred)

Genesis of the CAP Theorem

- I felt the design choices we made were “right”:
  - *Sufficient* (and faster)
  - *Necessary (consistency hinders performance/availability)*
- Started to notice other systems that made similar decisions: Coda, Bayou

- Developed CAP while teaching in 1998
  - Appears in 1999
  - PODC keynote in 2000, led to Gilbert/Lynch proof
- … but nothing changed (for a while)

CAP Theorem

- Choose at most two for any shared-data system:
  - **C**onsistency (linearizable)
  - **A**vailability (system always accepts *updates*)
  - **P**artition Tolerance
- Partitions are inevitable for the wide area
  - => consistency vs. availability
- I think this was the right phrasing for 2000
  - But probably not for 2010

Things CAP does *NOT* say..

1. Give up on consistency (in the wide area)
   - Inconsistency should be the exception
   - Many projects give up more than needed
2. Give up on transactions (ACID)
   - Need to adjust “C” and “I” expectations (only)
3. Don’t use SQL
   - SQL is appearing in “NoSQL” systems
   - Declarative languages fit well with CAP

CAP & ACID

No partitions => Full ACID

With partitions:

Atomic:

- Partitions should occur between operations (!)
- Each side should use atomic ops

Consistent:

- *Temporarily* violate this (e.g. no duplicates?)

Isolation:

- *Temporarily* lose this by definition

Durable:

- Should never forfeit this (and we need it later)

Single-site transactions

Atomic transaction, but only within one site

No distributed transactions

Google BigTable:

Multi-column row operations are atomic

… but that part of the row always within one site

CAP allows this just fine:

- Modulo no LAN partitions (reasonable)
- Google MegaStore spans multiple sites
  - Slow writes
  - Paxos helps availability, but still subject to partitions

Focus on partitions

Claim 1: partitions are *temporary*

- Provide degraded service for a while
- Then *RECOVER*

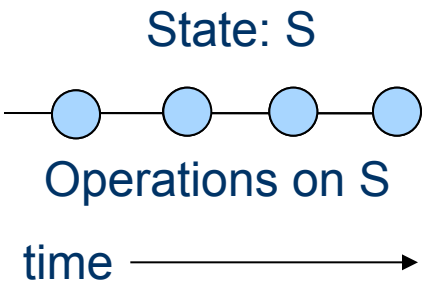
Claim 2: can *detect* “partition mode”

- Timeout => effectively partitioned
  - Commit locally? (A) => partition started
  - Fail? (C)
  - Retry just means postpone the decision a bit

Claim 3: impacts lazy vs. eager consistency

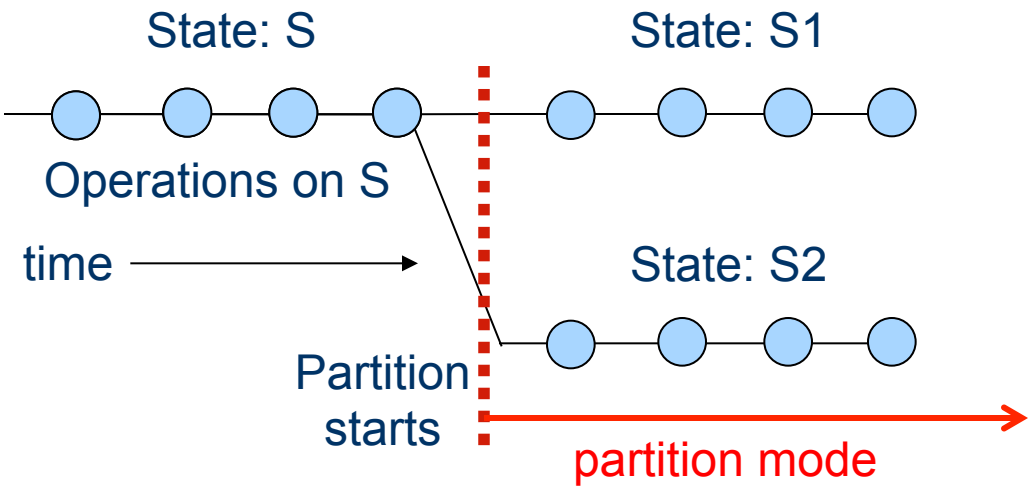
- Lazy => can’t recover consistency during partition
  - Can only choose A in some sense

Life of a Partition



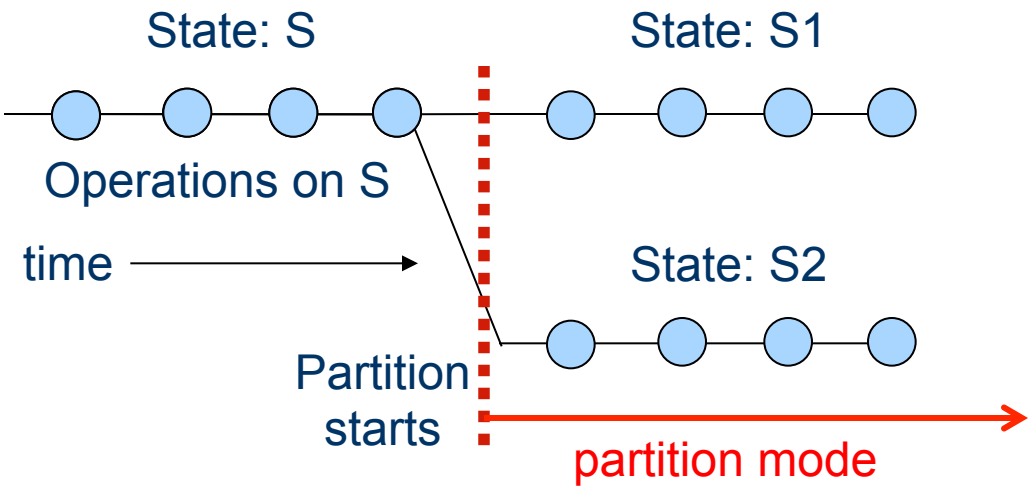
- Serializable operations on state S
- Available (no partitions)

Life of a Partition



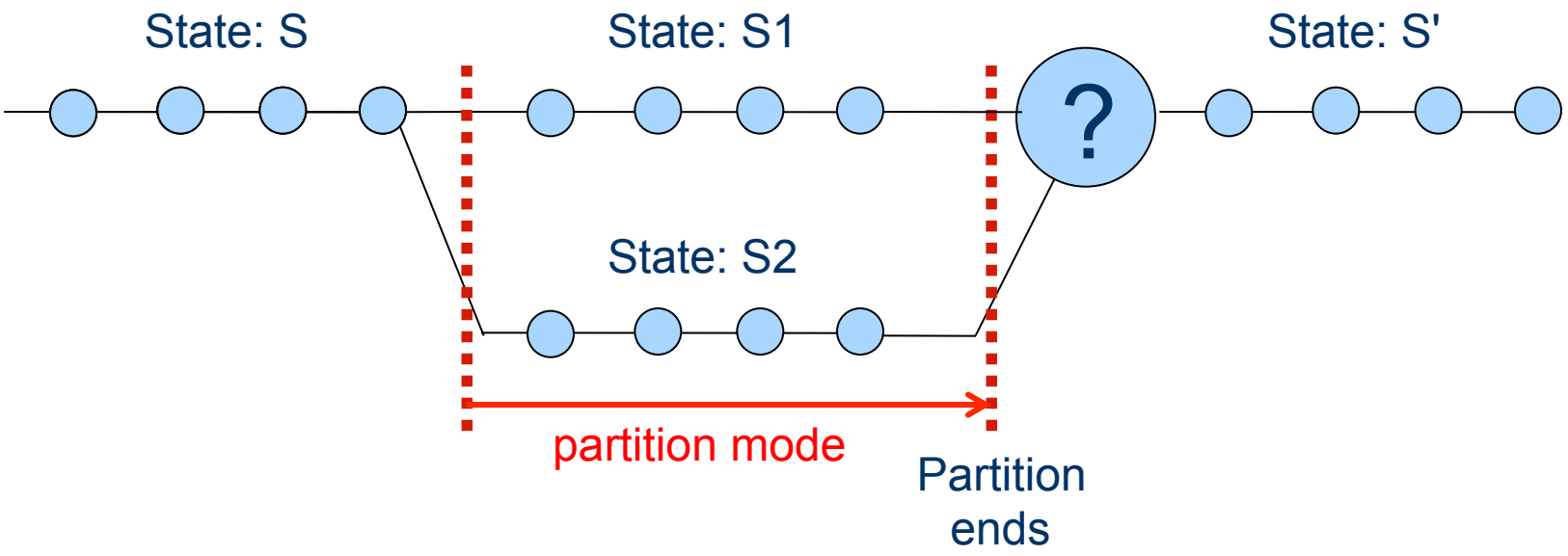
- Both sides available, *locally* linearizable
  … but (maybe) globally inconsistent
- No ACID “I”: concurrent ops on both sides
- No ACID “C” either (only local integrity checks)

Life of a Partition



- Commit locally?
- Externalize output? (A says yes)
- Execute side effects? (launch missile?)

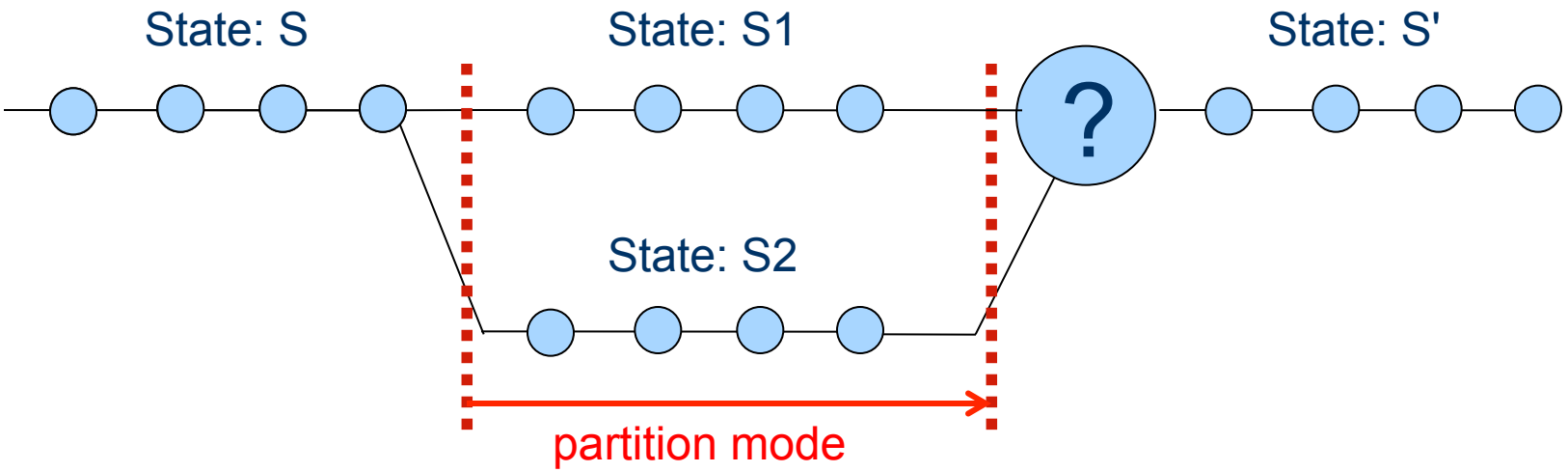
Life of a Partition



Need “Partition Recovery”

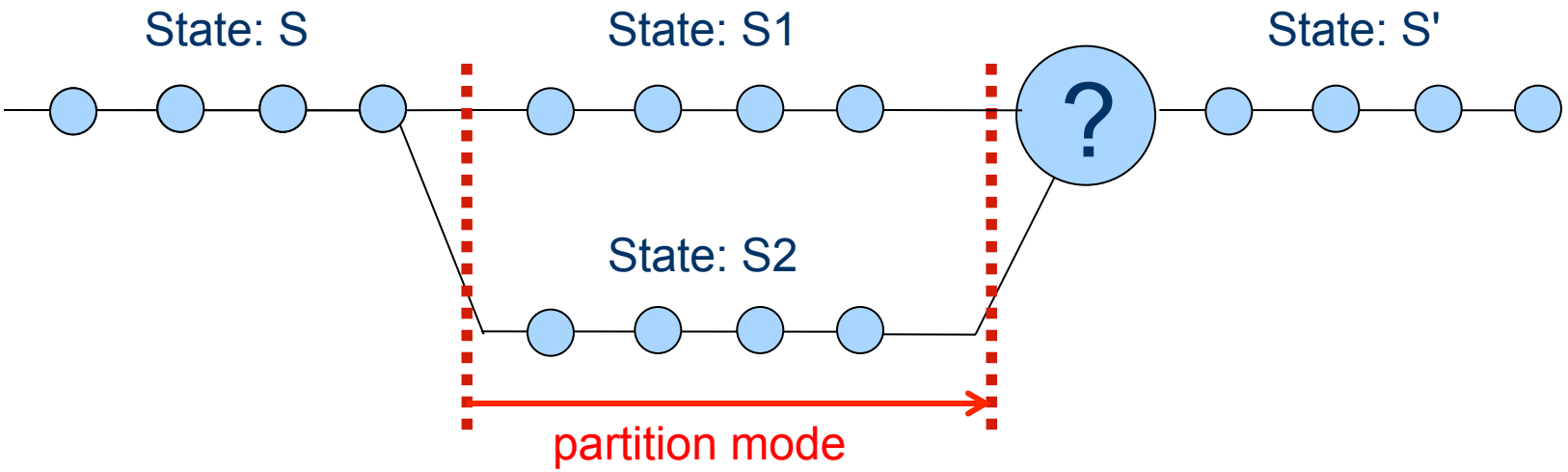
- Goal: restore consistency (ACID)
- Similar to traditional recovery
  - Move to some self-consistent state
  - Roll forward the “log” from each side

Partition Recovery



1) Merge State (S')
   - Easy: last writer wins
   - General: S' = *f*(S1 log, S2 log) // the paths matter
2) Detect bad things that you did
   - Side effects? Incorrect response?
3) Compensate for bad actions

Partition Recovery



Amazon shopping cart:

1) Merge by union of items
2) Only bad action is deleted item reappears

ATM “Stand In” Time

- ATMs have “partition mode”
  - … chooses A over C
  - Commutative atomic ops: incr, decr
  - When partition heals, *the end balance is correct*
- Partition recovery:
  - *Detect*: intermediate wrong decisions
    - Side effects (like “issue cash”) might be wrong
    - *Exceptions are not commutative* (below zero?)
  - *Compensate* via overdraft penalty
- Bound “wrongness” during partition: (less A)
  - Limit deficit to (say) $200
    - When you remove $200, “decr” becomes unavailable

Define your “Partition Strategy”

1) Define detection (start Partition Mode)
2) Partition Mode operation:

   Determine which operations can proceed

   - Can depend on args/access level/state
   - Simple example: no updates, read only
   - ATM: withdrawal allowed only up to $200 total

3) Partition recovery

   - Detect problems via joint logs
   - Execute compensations
     - Every allowed op should have a compensation
   - Calculate merged state (last)

Compensation Happens

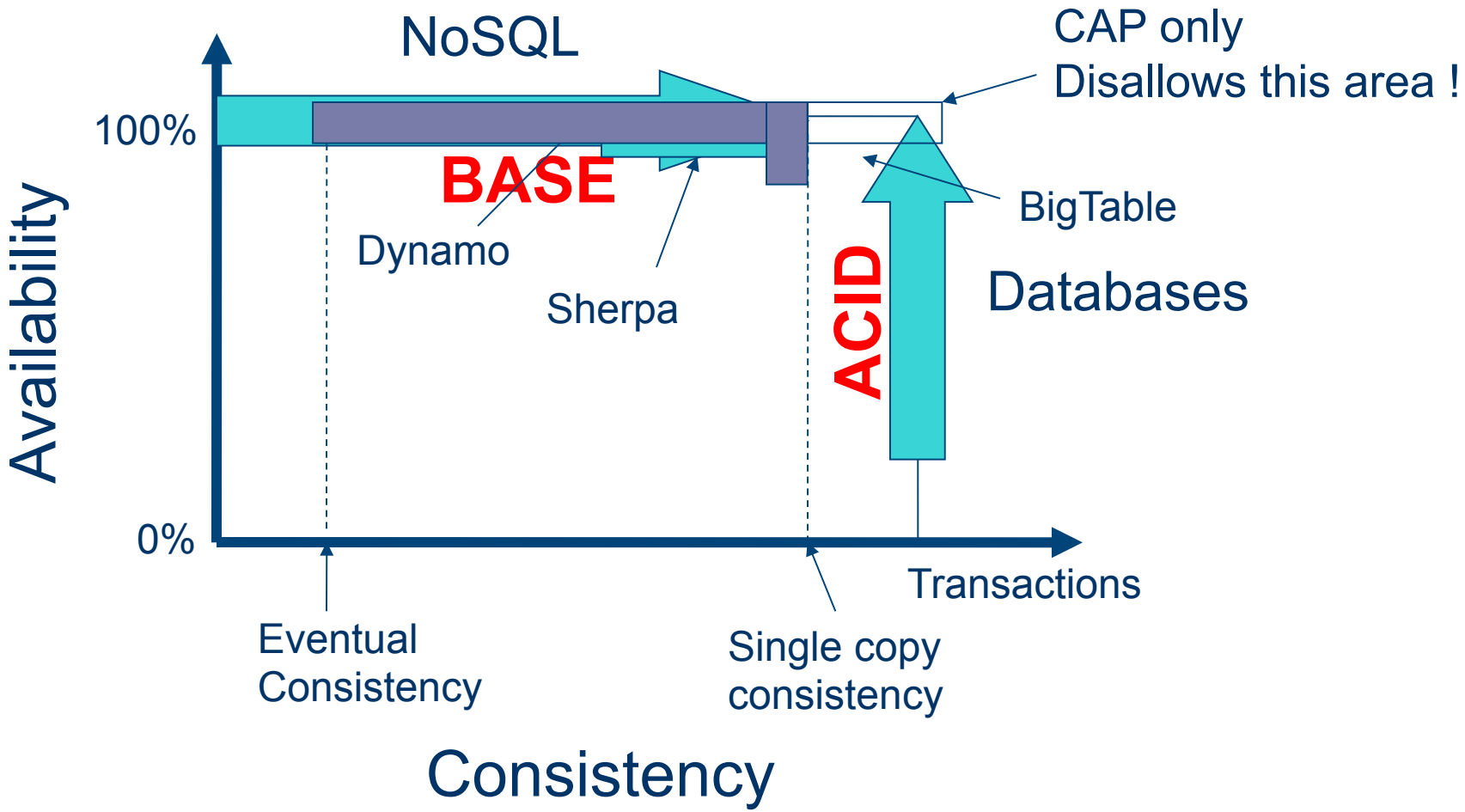
- Claim: Real world =

weak consistency + delayed exceptions + *compensation*

  - Charge you twice => credit your account
  - Overbook an airplane => compensate passengers that miss out

- This concept is missing from wide-area data systems
  - Except for some workflow
- Compensating transactions can be human response
  - “We just realized we sent you two of the same item”
  - Should be logged just like any other xact

CAP 2010


NoSQL
CAP only
Disallows this area !
100%
BASE
Dynamo
Sherpa
BigTable
Databases
ACID
Availability
0%
Transactions
Eventual
Consistency
Single copy
consistency
Consistency

Summary

- Net effect of CAP:
  - Freedom to explore a wide diverse space
  - Merging of systems and DB approaches
- While there are no partitions:
  - Can have both A and C, and full ACID xact
- Choosing A => focus on partition recovery
  - Need a before, during, and after strategy
  - Delayed Exceptions seem promising
  - Applying the ideas of compensation is open