NoSQL and ACID
NoSQL’s Motivation

Make it easy to build and deploy applications.

✓ Ease of scaling and operation
✓ Fault tolerance
✓ Many data models
✓ Good price/performance
X ACID transactions
What if we had ACID?

Only good for financial applications?
Big performance hit?
Sacrifice availability?

Nope… When NoSQL has ACID, it opens up a very different path.
The case for ACID in NoSQL
Bugs don’t appear under concurrency

• ACID means *isolation*.
• Reason *locally* rather than *globally*.
  – If every transaction maintains an invariant, then multiple clients running any combination of concurrent transactions also maintain that invariant.
• The impact of each client is *isolated*.
Isolation means strong abstractions

• Example interface:
  – `storeUser(name, SSN)`
  – `getName(SSN)`
  – `getSSN(name)`

• Invariant: \( N == getName(getSSN(N)) \)
  – Always works with single client.
  – Without ACID: Fails with concurrent clients.
  – With ACID: Works with concurrent clients.
Remove/decouple data models

- A NoSQL database with ACID can provide **polyglot** data models and APIs.
  - Key-value, graph, column-oriented, document, relational, publish-subscribe, spatial, blobs, ORMs, analytics, etc…

- Without requiring separate physical databases. **This is a huge ops win.**
Limited ACID

- Compare-and-set
- Partial ACID

Useful, but don’t enable strong abstractions or correct-by-default code
So, why don't we have ACID?

• History.
• It's hard.
History
In 2008, NoSQL doesn’t really exist yet.
The CAP\textsuperscript{2008} theorem

“Pick 2 out of 3”

- Eric Brewer
The CAP\textsubscript{2008} theorem

“Data inconsistency in large-scale reliable distributed systems has to be tolerated … [for performance and to handle faults]”

- Werner Vogles (CTO Amazon.com)
**CAP$_{2008}$ Conclusions?**

- Scaling requires distributed design
- Distributed requires high availability
- Availability requires no C

So, if we want scalability we have to give up C, a cornerstone of ACID, right?
Thinking about \( \text{CAP}_{2008} \)

\[
\text{CAP availability} \neq \text{High availability}
\]
Fast forward to CAP\textsuperscript{2013}

“Why ’2 out of 3’ is misleading”

“CAP prohibits… perfect availability”

- Eric Brewer
Fast forward to CAP\textsubscript{2013}

“Achieving strict consistency can come at a \textbf{cost} in update or read \textbf{latency}, and may result in lower \textbf{throughput}…”

- Werner Vogles (Amazon CTO)
Fast forward to \textit{CAP}_{2013}

“...it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions.“

- Google (Spanner)
ACID is Hard
The ACID NoSQL plan

• Maintain both **scalability** and **fault tolerance**
• Leverage CAP\textsubscript{2013} and deliver a CP system with true **global ACID transactions**
• Enable abstractions and **many data models**
• Deliver high per-node **performance**
Bolt-on approach

Bolt transactions on top of a database without transactions.
Bolt-on approach

Bolt transactions on top of a database without transactions.

• **Upside**: Elegance.

• **Downsides**: 
  – Nerd trap
  – Performance. “…integrating multiple layers has its advantages: integrating concurrency control with replication reduces the cost of commit wait in Spanner, for example” -Google
Transactional building block approach

Use non-scalable transactional DBs as components of a cluster.
Transactional building block approach

Use non-scalable transactional DBs as components of a cluster.

• **Upside**: Local transactions are fast
• **Downside**: Distributed transactions across machines are hard to make fast, and are messy (timeouts required)
Decomposition approach

Decompose the processing pipeline of a traditional ACID DB into *individual stages.*
Decomposition approach

Decompose the processing pipeline of a traditional ACID DB into *individual stages*.

• Stages:
  – Accept client transactions
  – Apply concurrency control
  – Write to transaction logs
  – Update persistent data representation

• **Upside**: Performance
• **Downside**: “Ugly” and complex architecture needs to solve tough problems for each stage
Challenges with ACID
Split brain challenge

• Any consistent database need a fault-tolerance source of “ground truth”
• Must prevent database from splitting into two independent parts

Solution:
• Using thoughtfully chosen Paxos nodes can yield high availability, even for drastic failure scenarios
• Paxos is not required for each transaction
Latency challenge

- Durability costs latency
- Causal consistency costs latency

**Solution:**
- Bundling ops reduces overhead
- ACID costs only needed for ACID guarantees
Correctness challenge

• MaybeDB:
  – Set(key, value) – Might set key to value
  – Get(key) – Get a value that key was set to

Solution:

• The much stronger ACID contract requires vastly more powerful tools for testing
NoSQL + ACID!
ACID results

Jepsen test results

2000 total
2000 acknowledged
2000 survivors

All 2000 writes succeeded. :-D 0% data lost!
Performance results

- A 24-machine FoundationDB cluster processing 100% cross-node transactions saturates its SSDs at 890,000 op/s
Scalability results

Performance of different cluster sizes
A vision for NoSQL

• The next generation should **maintain**
  – Scalability and fault tolerance
  – High performance

• While **adding**
  – ACID transactions
  – Data model flexibility
Thank you