Rethinking Eventual Consistency
A survey of synchronization techniques for replicated distributed databases

Phil Bernstein & Sudipto Das
Microsoft Research
June, 2013

Definition: Eventual Consistency

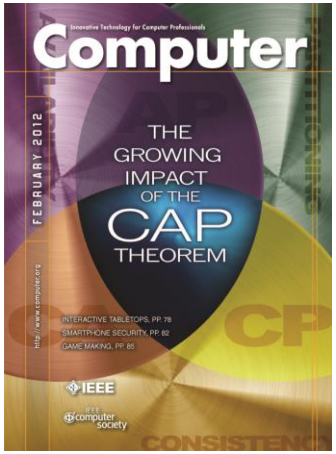
In a replicated database, updates arrive in different orders at different copies of a data item,

but eventually

the copies converge to the same value.

Eventual Consistency is All the Rage

- Origin: Thomas' majority consensus algorithm, published in 1979 (ACM TODS).
- Was used in Grapevine (PARC, early 1980's) and in numerous systems since then.
- Doug Terry et al. coined the term in a 1994 Bayou paper
- Werner Vogels at Amazon promoted it in Dynamo (2007)
- Cover topic of February 2012 IEEE Computer



Despite today's hype

- Most of what we'll say was known in 1995
- There are many published surveys
  - But this talk has a rather different spin
- We'll often cite old references to remind you where the ideas came from

Correctness Goal

- Ideally, replication is transparent

In the world of transactions:

- One-Copy Serializability - The system behaves like a serial processor of transactions on a one-copy database

  [Attar, Bernstein, & Goodman, IEEE TSE 10(6), 1984]

In the world of operations:

- Linearizability - A system behaves like a serial processor of operations on a one-copy database

  [Herlihy & Wing, ACM TOPLAS 12(3), 1990]

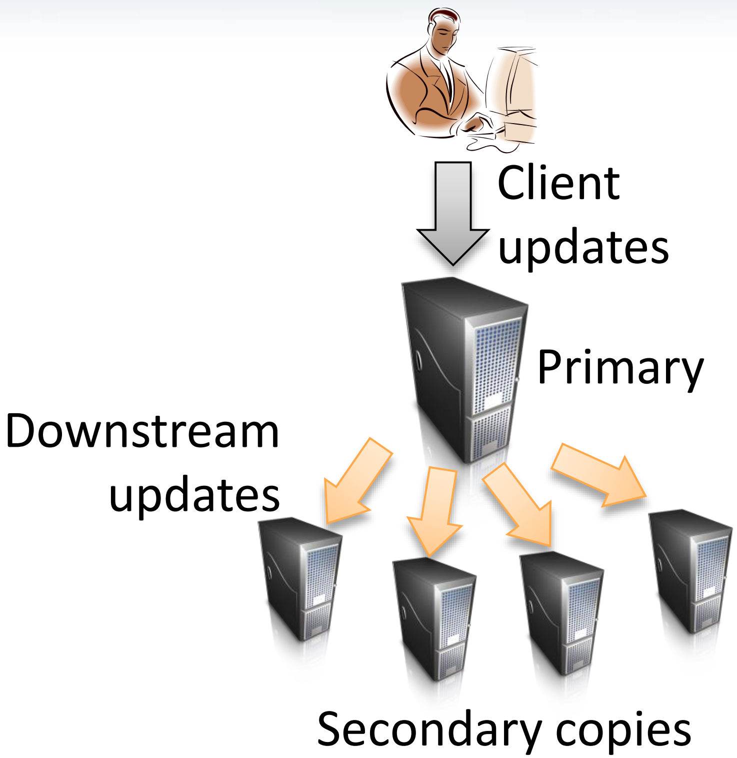
Nice Goal If You Can Get It

- But you can’t in many practical situations

- Let’s review the three main types of solutions
  - Primary Copy
  - Multi-Master
  - Consensus Algorithms

Primary Copy

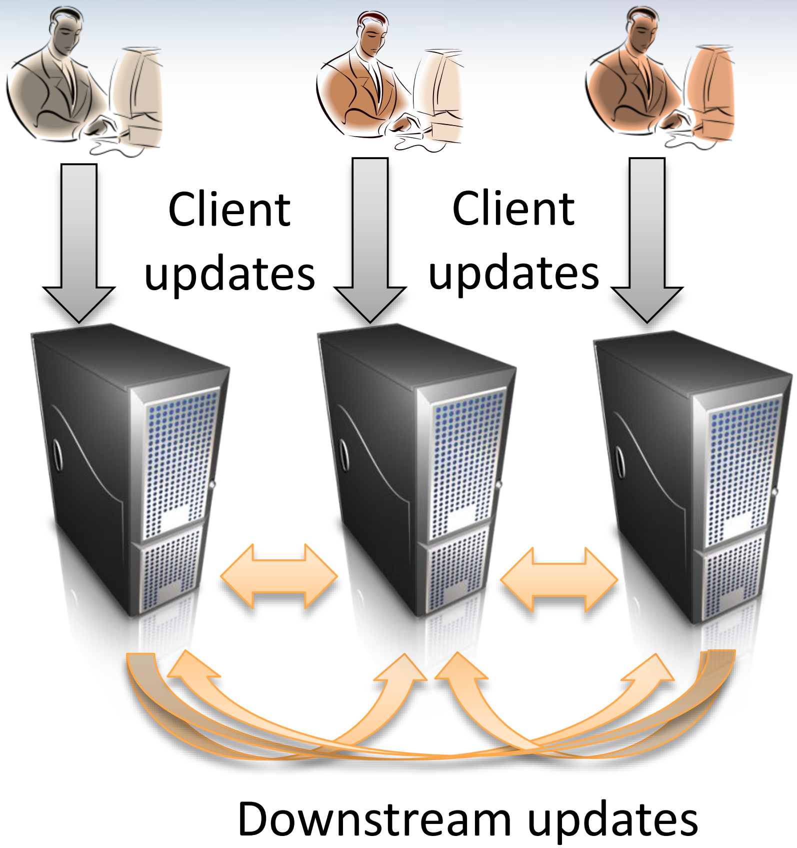
- Only the primary copy is updatable by clients
- Updates to the primary flow downstream to secondaries
- What if there's a network partition?
- Clients that can only access secondaries can't run updates



[Alsberg & Day, ICSE 1976] [Stonebraker & Neuhold, Berkeley Workshop 1979]

Multi-Master

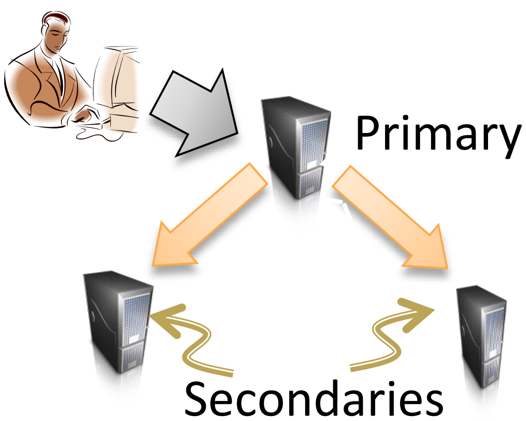
- Copies are independently updatable
- Conflicting updates on different copies are allowed
- Doesn't naturally support 1SR.
- To ensure eventual consistency or linearizability of copies:
  - Either updates are designed to be commutative
  - Or conflicting updates are detected and merged



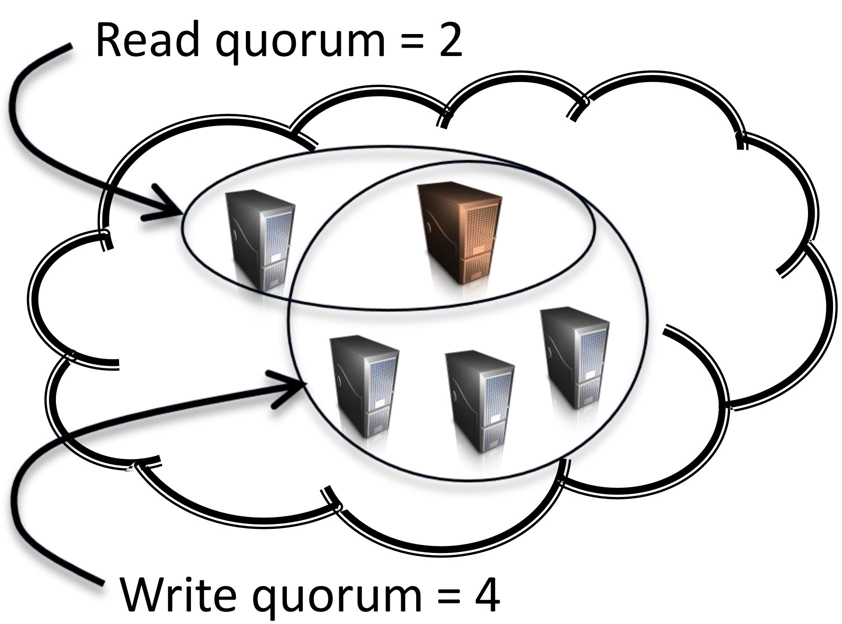
- "The partitioned DB problem" in late 1970's.
- Popularized by Lotus Notes, 1989

Consensus Algorithms

- Copies can be a replicated-state machine
  - Essentially, a serial processor of operations
  - Can be primary-copy or multi-master
- Uses quorum consensus to achieve 1SR or linearizability.
  - Ensures conflicting ops access at least one copy in common



Each downstream update is applied to a quorum of secondaries



The CAP Theorem

- You can have only two of Consistency-of-Replicas, Availability, and Partition-Tolerance
  - Can get C & A, if there's no partition
  - Can get C & P but only one partition can accept updates
  - Can get A & P, but copies in different partitions won't be consistent

Conjecture by [Brewer, PODC 2000]

Proved by [Gilbert & Lynch, SIGACT News 33(3) 2002]

This Isn't Exactly News

- "Partitioning - When communication failures break all connections between two or more active segments of the network ... each isolated segment will continue ... processing updates, but there is no way for the separate pieces to coordinate their activities. Hence ... the database ... will become inconsistent. This divergence is unavoidable if the segments are permitted to continue general updating operations and in many situations it is essential that these updates proceed."
- [Rothnie & Goodman, VLDB 1977]
- So the CAP theorem isn't new, but it does focus attention on the necessary tradeoff

Can we do better than Eventual Consistency?

- There have been many attempts at defining stronger but feasible consistency
  - Parallel snapshot isolation
  - Consistent prefix
  - Monotonic reads
  - Timeline consistency
  - Linearizability
  - Eventually consistent transactions
  - Causal consistency
  - Causal+ consistency
  - Bounded staleness
  - Monotonic writes
  - Read-your-writes
  - Strong consistency

It’s Confusing

- We’ll try to eliminate the confusion by
  - Characterizing consistency criteria
  - Describing mechanisms that support each one
  - And summarizing their strengths and weaknesses

Disclaimer

- There are many excellent surveys of replication
  - We don't claim ours is better, just different

- S.B. Davidson, H. Garcia-Molina, D. Skeen: Consistency in Partitioned Networks. ACM Computing Surveys. Sept. 1985
- S-H Son: Replicated data management in distributed database systems, SIGMOD Record 17(4), 1988.
- Y. Saito, M. Shapiro: Optimistic replication. ACM Comp. Surveys. Jan. 2005
- P. Padmanabhan, et al.: A survey of data replication techniques for mobile ad hoc network databases. VLDB Journal 17(5), 2008
- D.B. Terry: Replicated Data Management for Mobile Computing. Morgan & Claypool Publishers 2008
- B. Kemme, G. Alonso: Database Replication: a Tale of Research across Communities. PVLDB 3(1), 2010
- B. Kemme, R. Jiménez-Peris, M. Patiño-Martínez: Database Replication. Morgan & Claypool Publishers 2010
- P.A. Bernstein, E. Newcomer: Principles of Transaction Processing (2nd edition). Chapter 9 on Replication. Elsevier, 2009.

More Disclaimers

- There’s a huge literature on replication.
  - Please tell us if we missed something important
- We’ll cover replication mechanisms in database systems, distributed systems, programming languages, and computer-supported cooperative work
  - We won’t cover mechanisms in computer architecture

Preview of the Design Space

- Multi-master is designed to handle partitions
- With primary copy, during a partition
  - Majority quorum(x) = partition with a quorum of x's copies
  - Majority quorum can run updates and satisfy all correctness criteria
  - Minority quorum can run reads but not updates, unless you give up on consistency
- So an updatable minority quorum is just like multi-master

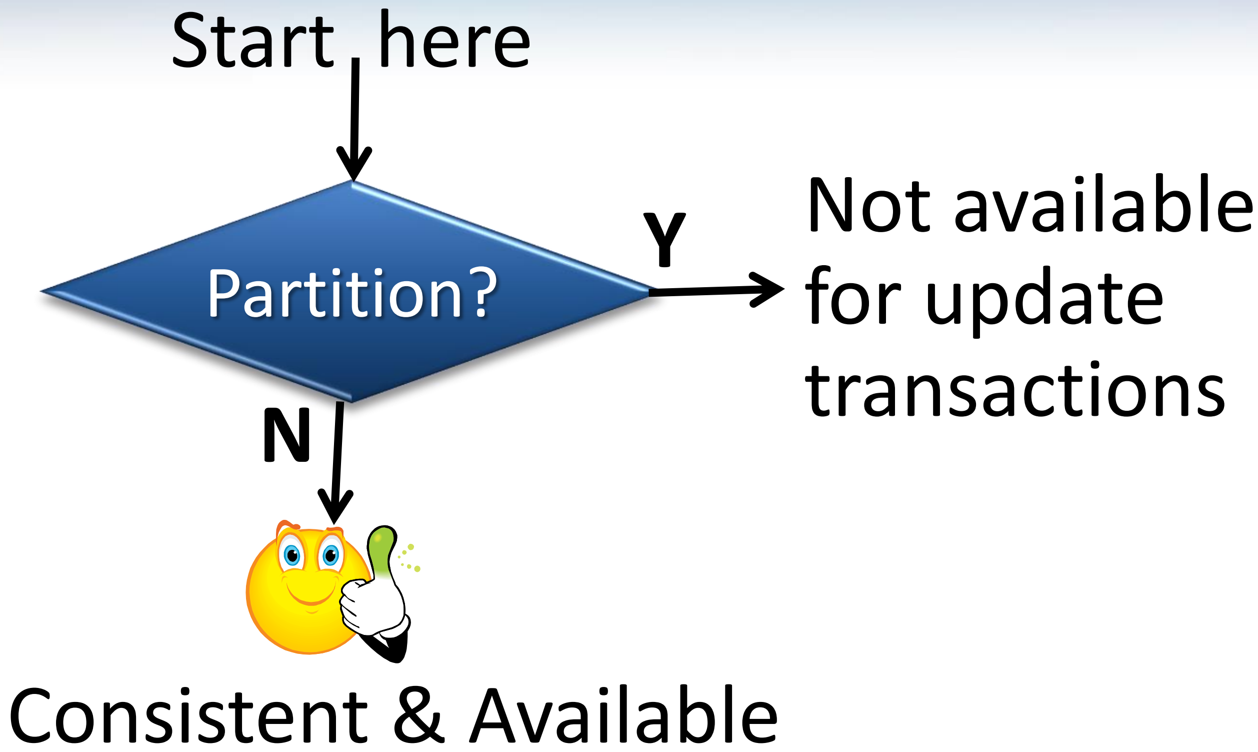
Preview of the Design Space (2)

- Eventual consistency – there are many good ways to achieve it
- For isolation and session goals, the solution space is more complex
  - Strengthens consistency, but complicates programming model
  - Improves availability, but not clear by how much
  - If a client rebinds to another server, ensuring these goals entails more expense, if they're attainable at all.
  - No clear winner

Bottom Line

- App needs to cope with arbitrary states during a partition
- Offer a range of isolation and session guarantees and let the app developer choose among them
  - Possibly worthwhile for distributed systems experts
  - Need something simpler for “ordinary programmers”
- Encapsulate solutions that offer good isolation for common scenarios
  - Use data types with commutative operations
  - Convergent merges of non-commutative operations
  - Scenario-specific classes

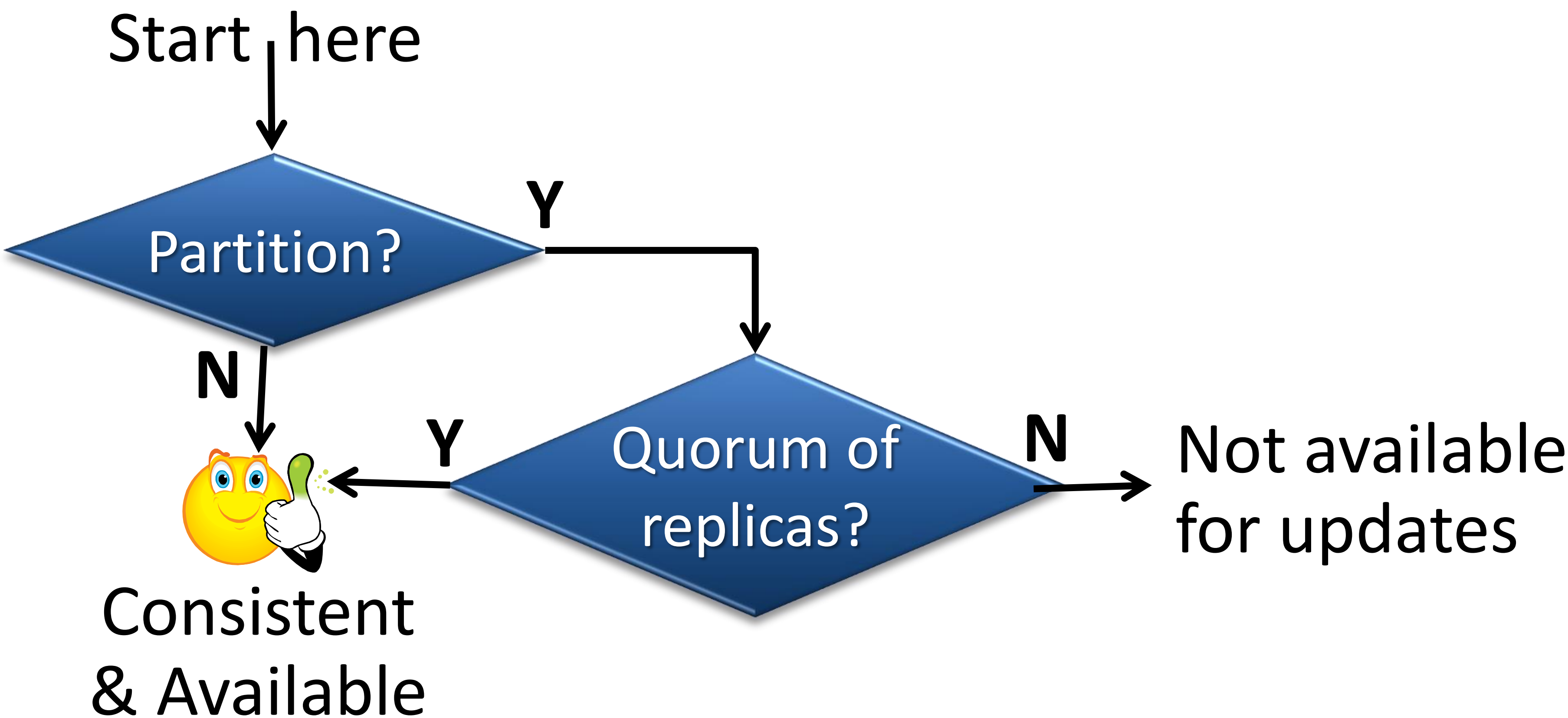
Organize Taxonomy by a Flowchart



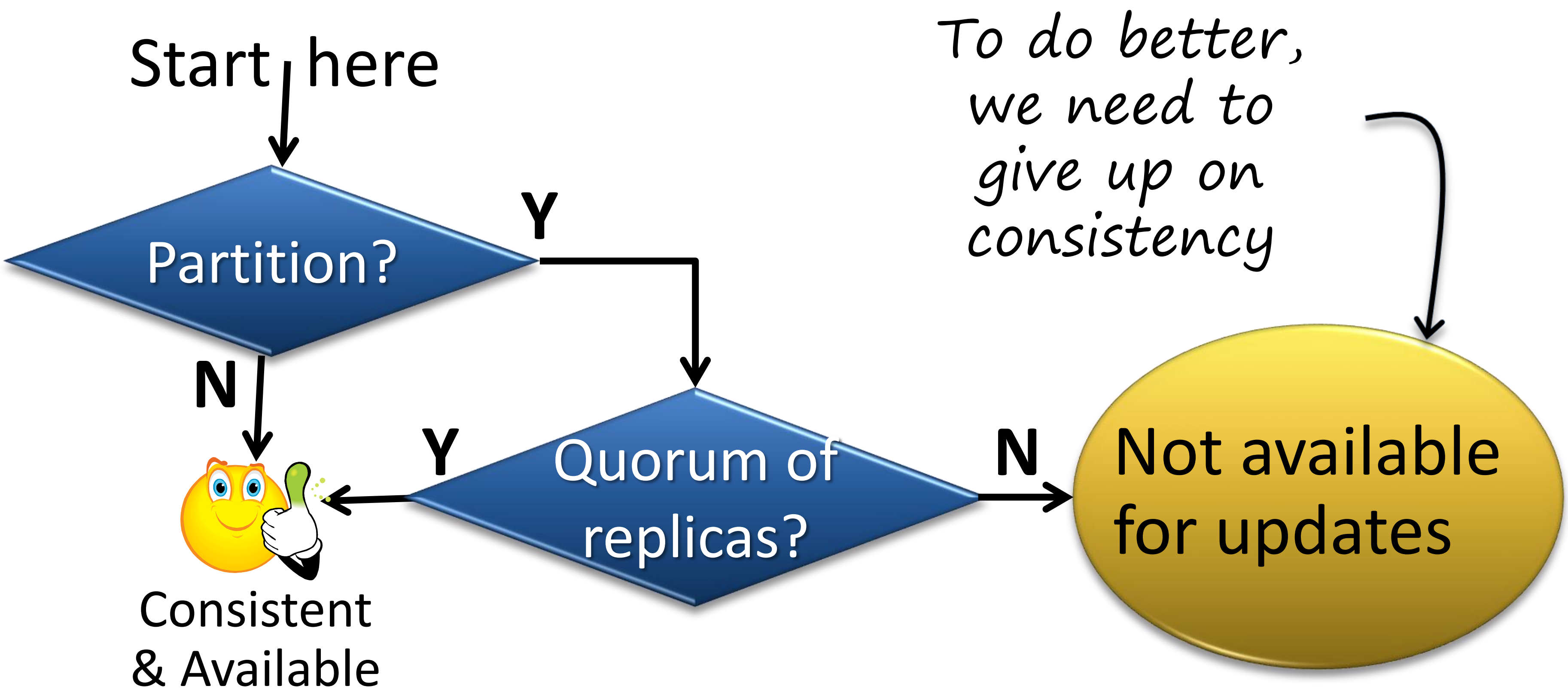
- We'll start with the world of operations, and then look at the world of transactions

Can Have One Writable Partition

- The partition with a quorum of replicas can run writes



What to do about the bad case?



Eventual Consistency

- Eventual consistency is one popular proposal
  - The copies will be identical … someday
  - App still needs to handle arbitrary intermediate states
- How to get it
  - Commutative downstream operations
  - Mergeable operations
  - Vector clocks

Commutative Downstream Updates

Thomas' Write Rule:

- [Thomas, ACM TODS 4(2), 1979]
- Assign a timestamp to each client write operation
- Each copy of *x* stores timestamp(last-write-applied)
- <u>Apply</u> downstream-write(*x*) only if downstream-write(*x*).timestamp > *x*.timestamp
- So highest-timestamp wins at every copy

Downstream writes arrive in this order

W(X=40), TS:1
W(X=70), TS:5
W(X=30), TS:3

Final value:
X=70, TS:5

Commutative Downstream Operations (2)

Pros

- Updates can be applied anywhere, anytime
- Downstream updates can be applied in any order after a partition is repaired

Cons

- Doesn't solve the problem of ordering reads & updates
- For fairness, requires loosely-synchronized clocks

Commutative Downstream Updates (3)

Convergent & Commutative Replicated Data Types

- [Shapiro et al., INRIA Tech. Report, Jan 2011]
- Set operations add/remove don't commute,
- [add(E), add(E), remove(E)] $\not\equiv$ [add(E), remove(E), add(E)]
- But for a counting set, they do commute
  - Each element E in set S has an associated count
  - Add(set S, element E) increments the count for E in S.
  - Remove(S, E) decrements the count

Commutative Downstream Operations (4)

Pros

- Updates can be applied anywhere, anytime
- Downstream updates can be applied in any order after a partition is repaired

Cons

- Constrained, unfamiliar programming model
  - Doesn't solve the problem of ordering reads & updates
- Some app functions need non-commutative updates

Custom Merge Operations

- Custom merge procedures for downstream operations whose client operations were not totally ordered.
  - Takes two versions of an object and creates a new one
- For eventual consistency, merge must be commutative and associative
- Notation: $M(O_2, O_1)$ merges the effect of O_2 into O_1
- Commutative: $O_1 \cdot M(O_2, O_1) \equiv O_2 \cdot M(O_1, O_2)$
- Associative: $M(O_3, O_1 \cdot M(O_2, O_1)) \equiv M(M(O_3, O_2) \cdot O_1)$
- [Ellis & Gibbs, SIGMOD 1989]

Custom Merge Operations (cont'd)

Pros

- Enables concurrent execution of conflicting operations without the synchronization expense of total-ordering

Cons

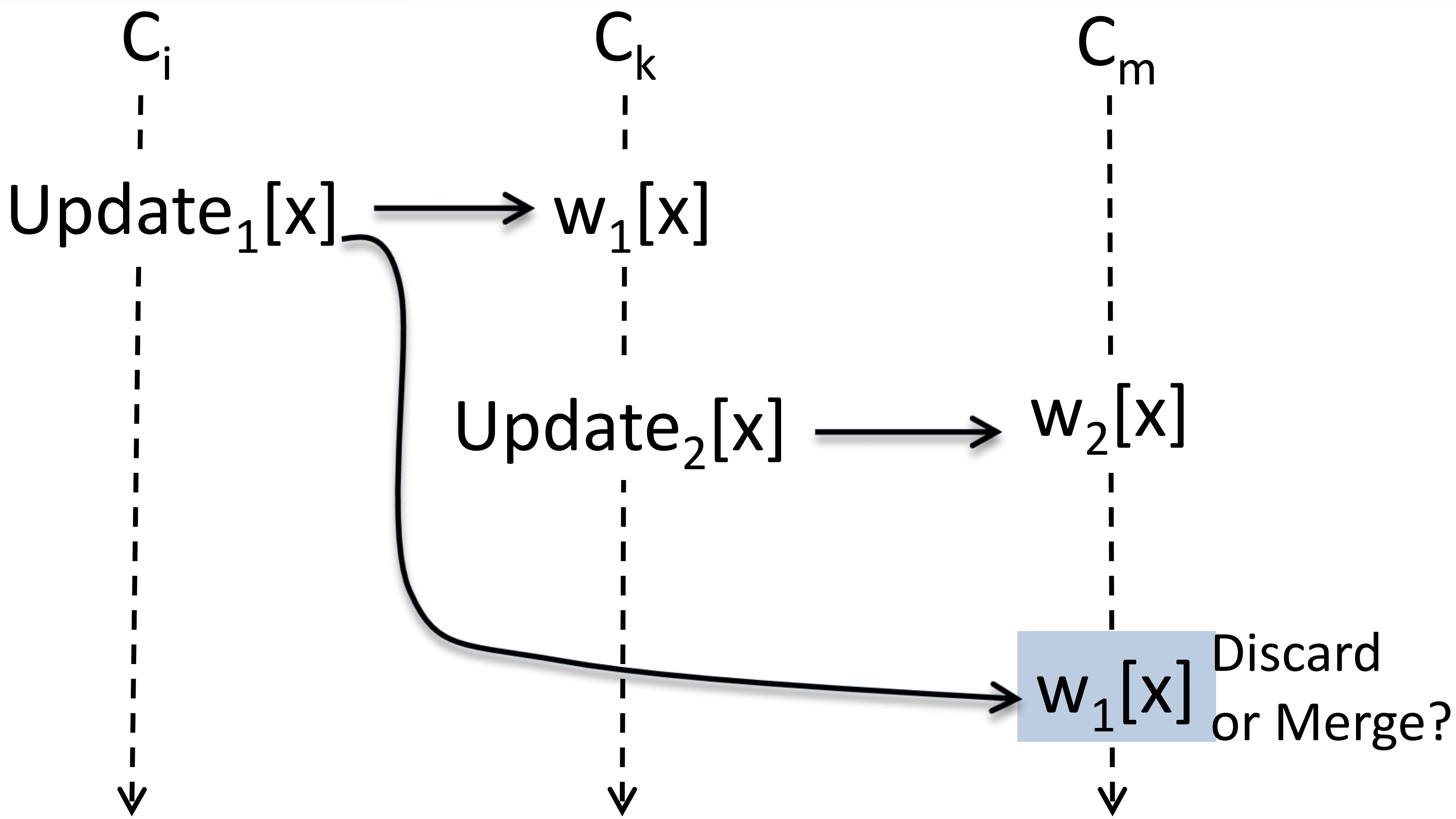
- Requires application-specific logic that's hard to generalize

Vector Clocks tell us the merging order

- In multi-master, each copy assigns a monotonically increasing version number to each client update
- Vector clock is an array of version numbers, one per copy
  - Identifies the set of updates received or applied
- Use it to identify the state that a client update depends on and hence overwrote
  - If two updates conflict but don't depend on one another, then merge them.

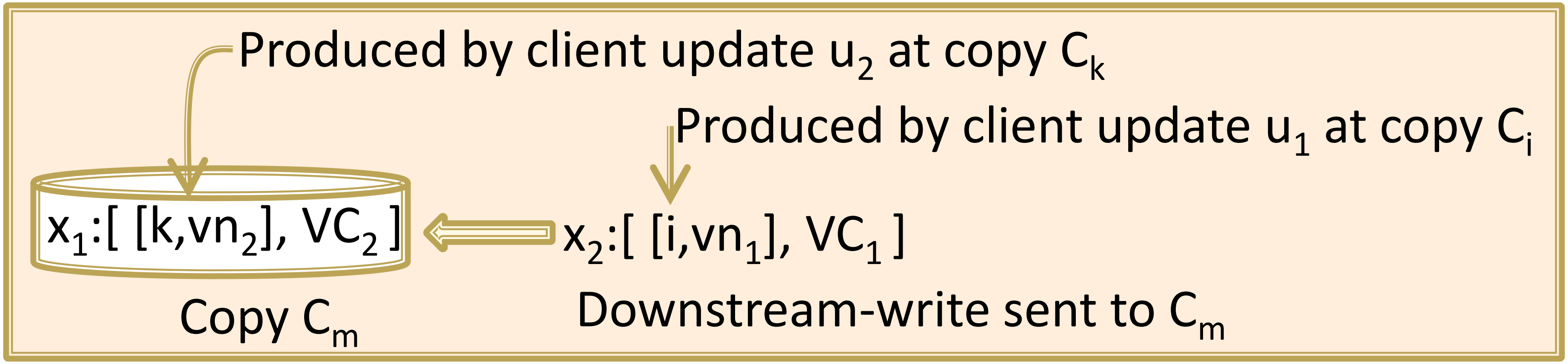
- [Fischer & Michael, PODS 1982]
- [Parker et al., IEE TSE 1983]
- [Wuu & Bernstein, PODC 1984]

Problem: Discard or Merge?



Vector Clocks(2)

- A vector clock can be used to identify the state that a client update depends on ("made-with knowledge")



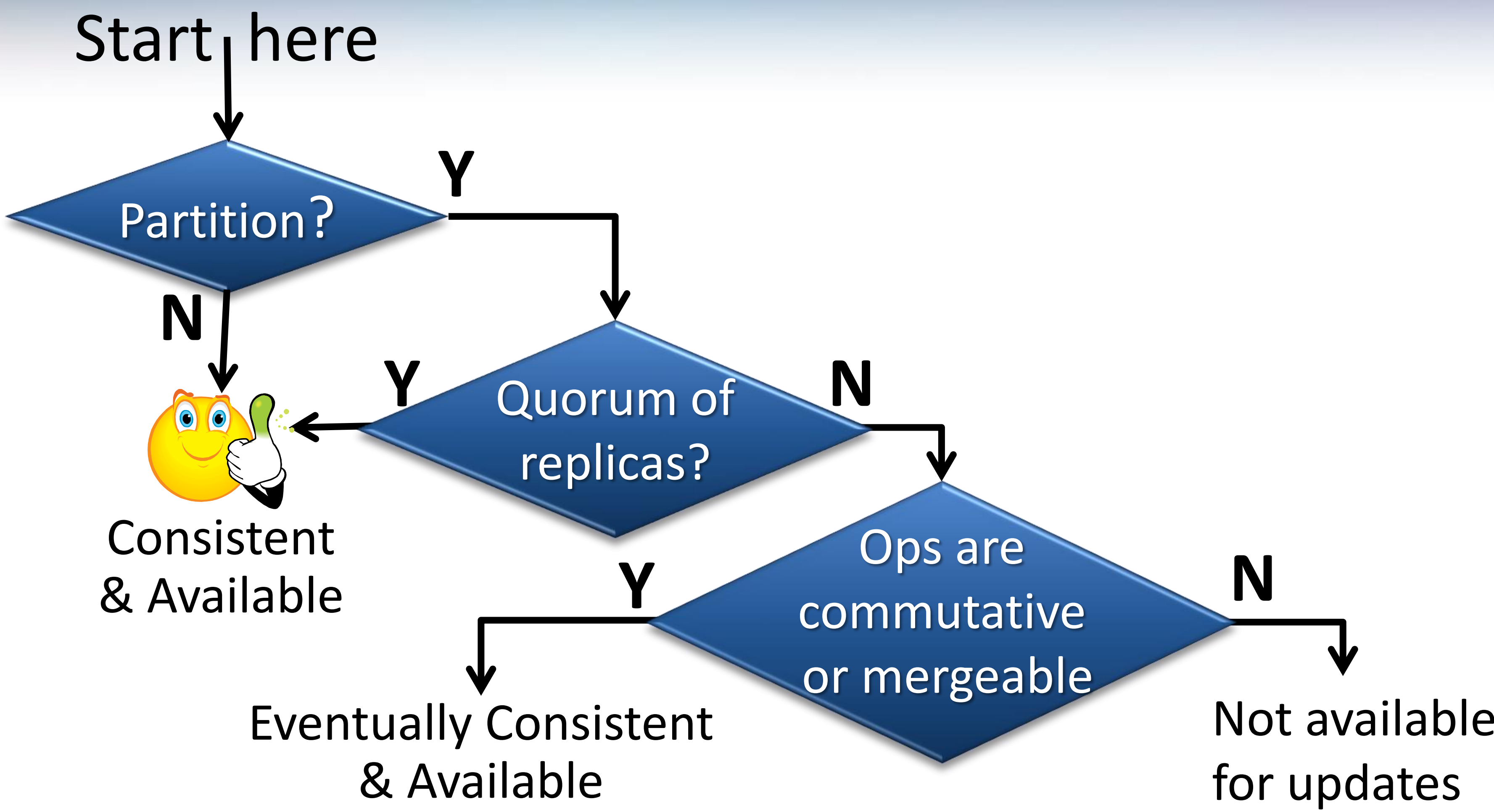
  – If $VC_1[k] \geq vn_2$, then x_2 was "made from" x_1 & should overwrite it
  – If $VC_2[i] \geq vn_1$, then x_1 was "made from" x_2, so discard x_2
  – Else the updates should be reconciled

[Ladin et al., TOCS, 1992]
[Malkhi & Terry, Dist. Comp. 20(3), 2007]

Another Use of Vector Clocks

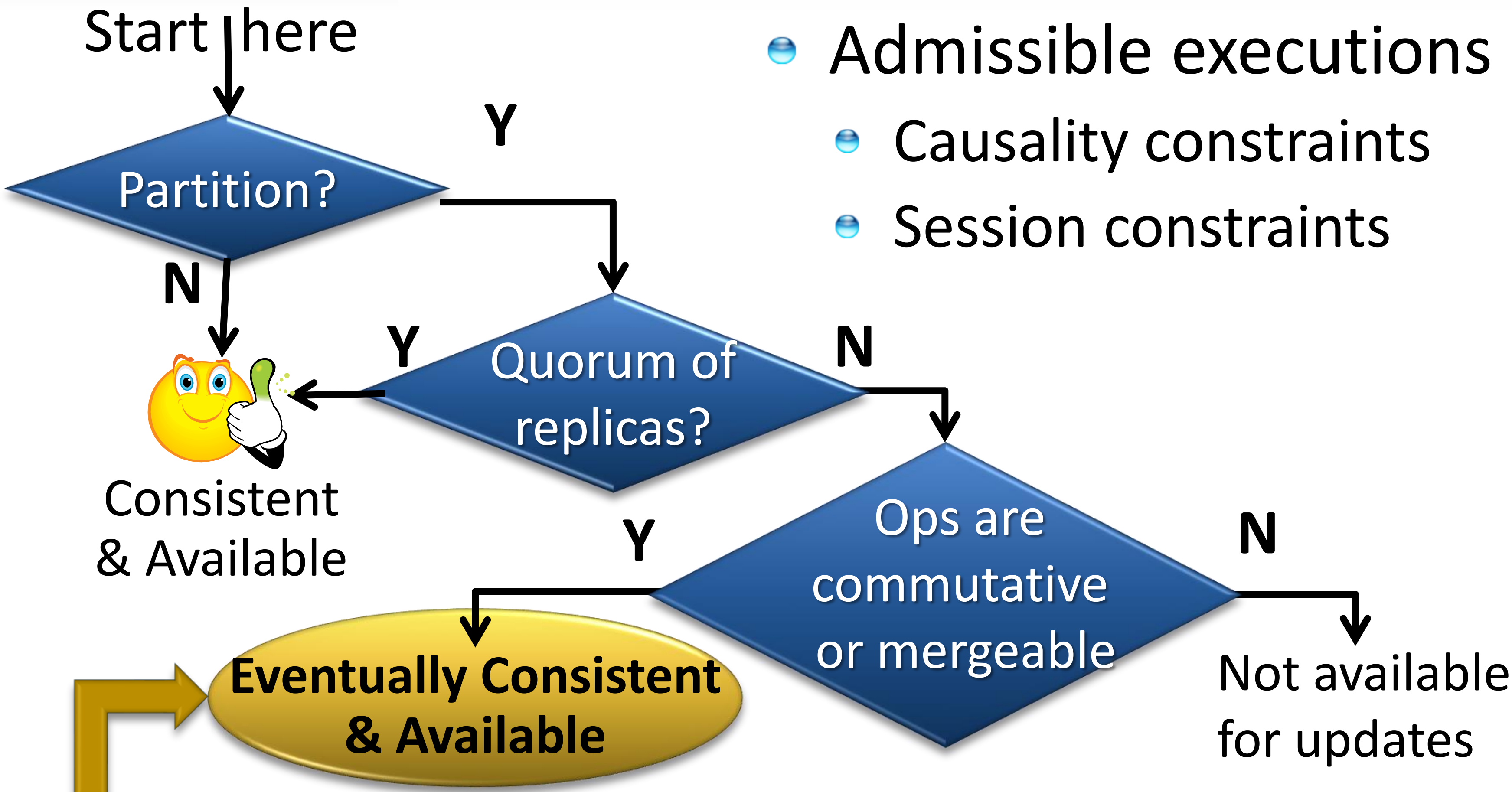
- A copy can use it to identify the updates it has received
  - When it syncs with another copy, they exchange vector clocks to tell each other which updates they already have.
- Avoids shipping updates the recipient has already seen
- Enables a copy to discard updates that it knows <u>all</u> other copies have seen

In the Operation World



Strengthening Eventual Consistency

- Admissible executions
  - Causality constraints
  - Session constraints

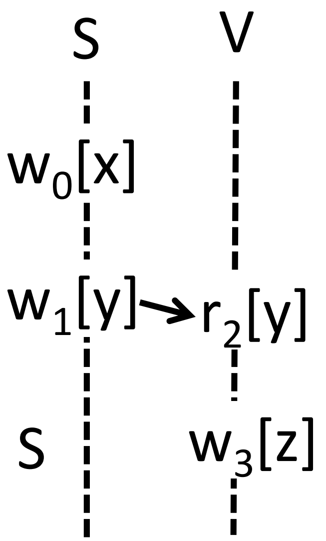


The case we can strengthen

Causal Consistency

<u>Definition</u> – The sequence of operations on each replica is consistent with session order and reads-from order.

- Example: User 1 stores a photo P and a link L to it.
  If user 2 reads the link, then she'll see the photo.
- Causality imposes write-write orders



Causal relationships:

- **WW Session order:** $w_1[y]$ executes after $w_0[x]$ in session S
- **WR Session order**: $w_3[z]$ executes after $r_2[y]$ in session V
- **Reads-from order**: $r_2[y]$ in session V reads from $w_1[y]$ in session S
- **Causality is transitive:** Hence, $w_0[x]$ causally precedes $w_3[z]$

[Lamport, CACM 21(7), 1978]

Causal Consistency (2)

- If all atomic operations preserve database integrity, then causal consistency with eventual consistency may be good enough
  - Store an object, then a pointer to the object
  - Assemble an order and then place it
  - Record a payment (or any atomically-updatable state)
- Scenarios where causal consistency isn't enough
  - Exchanging items: Purchasing or bartering require each party to be credited and debited atomically
  - Maintaining referential integrity: One session deletes an object O while another inserts a reference to O

Implementing Causal Consistency

- Enforce it using dependency tracking and vector clocks
- COPS: Causality with convergent merge [Lloyd et al., SOSP 2011]
  - Assumes multi-master replication
  - Session context (dependency info) = <data item, version#> of the last items read or of the last item written.
  - Each downstream write includes its dependent operations.
  - A write is applied to a copy after its dependencies are satisfied
  - Merge uses version vectors
  - With additional dependency info, it can support snapshot reads
  - Limitation: No causal consistency if a client rebinds to another replica due to a partition

Session Constraints

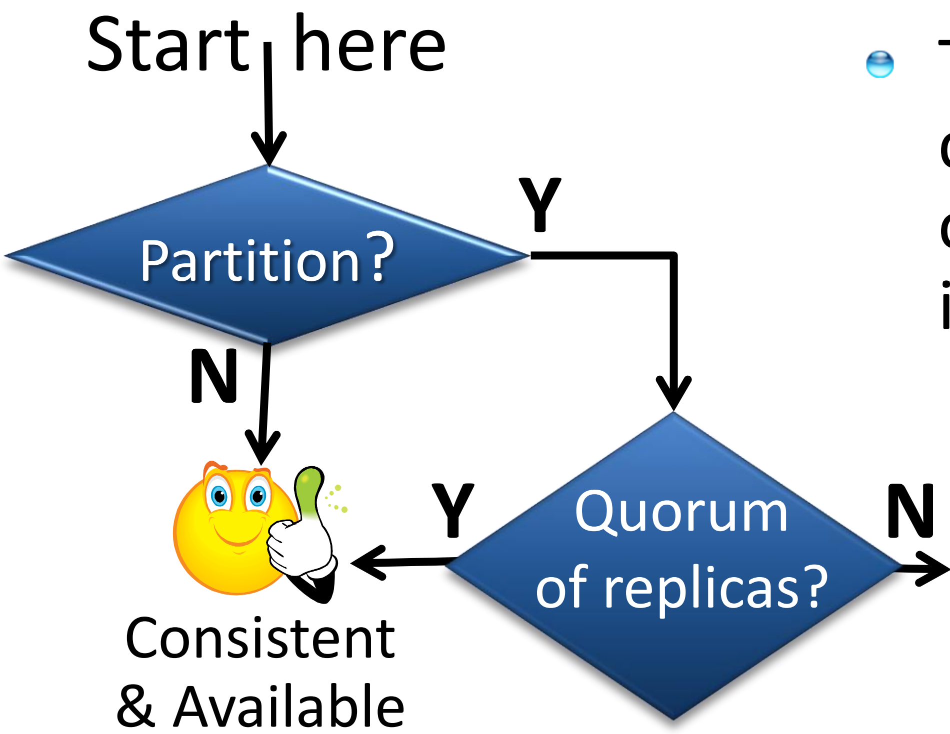
- **Read your writes** – a read sees all previous writes
- **Monotonic reads** – reads see progressively later states
- **Monotonic writes** – writes from a session are applied in the same order on all copies
- **Consistent prefix** – a copy's state only reflects writes that represent a prefix of the entire write history
- **Bounded staleness** – a read gets a version that was current at time *t* or later

[Terry et al., PDIS 1994]

Mechanisms for Session Constraints

- Client session maintains IDs of reads and writes
  - ✓ Accurate representation of the constraints
  - ☹ High overhead per-operation
- Client session maintains vector clocks for the last item read or written
  - ✓ Compact representation of the constraints
  - ☹ Conservative

In the Transaction World



- The operation world ignores transaction isolation
- To get the benefits of commutative or mergeable operations, need a weaker isolation level

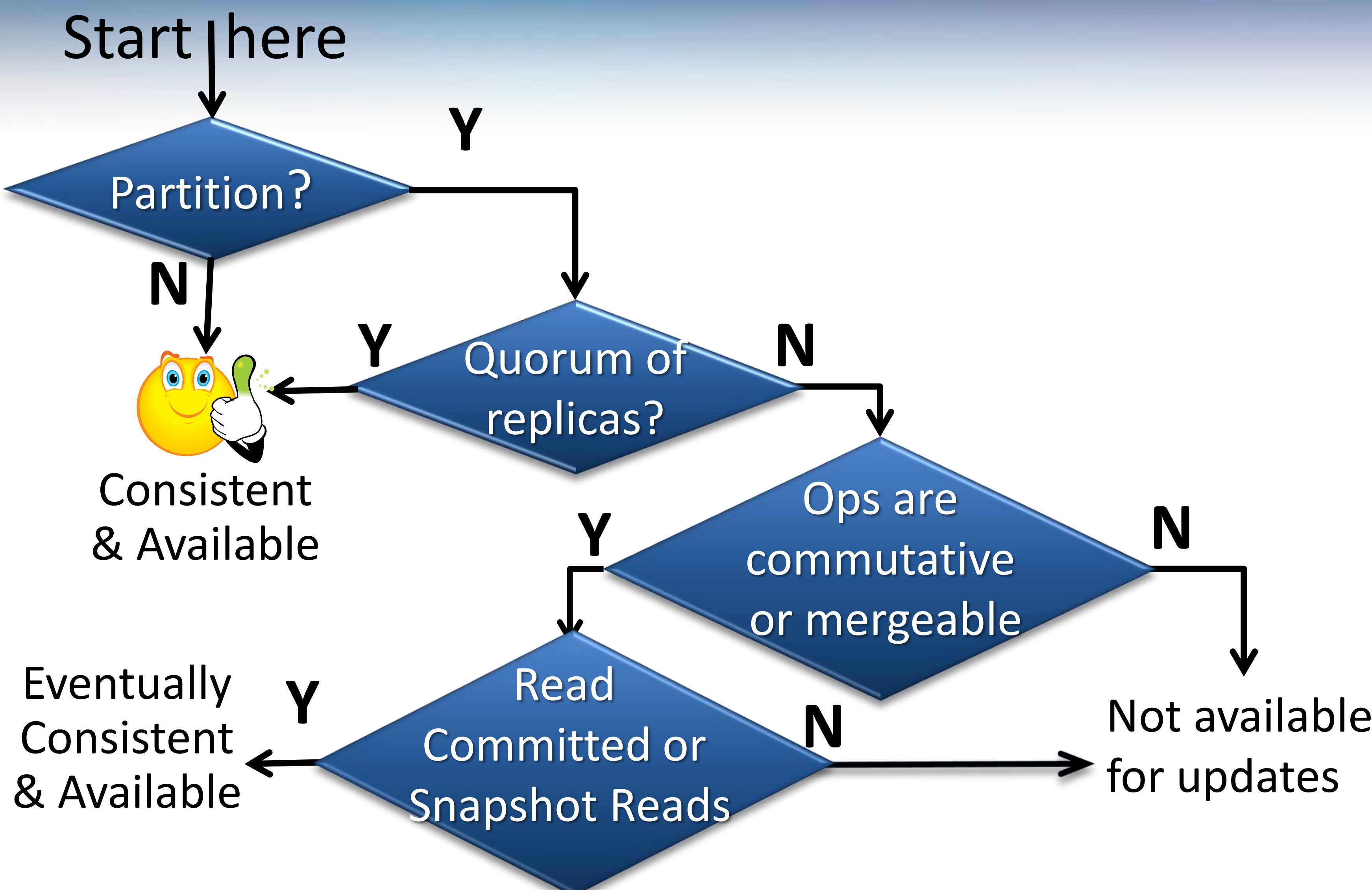
Common weaker isolation levels

- Read committed
  - Transaction reads committed values
- Snapshot reads
  - Transaction reads committed values that were produced by a set of committed transactions
  - All of a transaction’s updates must be installed atomically to ensure the writeset is consistent in the minority partition

Is Weaker Isolation Acceptable?

- People do it all the time for better performance
  - Throughput of Read-Committed is 2.5x to 3x that of Serializable
- Weaker isolation produces errors. Why is this OK?
- No one knows, but here are some guesses:
  - DB's are inconsistent for many other reasons.
    - Bad data entry, bugs, duplicate txn requests, disk errors, ….
  - Maybe errors due to weaker isolation levels are infrequent
  - When DB consistency matters a lot, there are external controls.
    - People look closely at their paychecks
    - Financial information is audited
    - Retailers take inventory periodically

In the Transaction World



Other Admissibility Constraints

- Admissible executions
  - Causality constraints
  - Session constraints
  - Isolation constraints
    - **RedBlue Consistency [Li et al., OSDI 2012]**
    - **1-SR, Read-committed, Snapshot Isolation**
    - **Parallel Snapshot Isolation [Sovran et al, SOSP 2011]**
    - **Concurrent Revisions [Burckhardt et al., ESOP 2012]**

RedBlue Consistency

- *Blue* operations commute with all other operations and can run in different orders on different copies.
- *Red* ones must run in the same order on all copies.
- Use a side-effect-free *generator* operation to transform a red operation to a blue one that is valid in all states
- Example
  - Deposit(acct, amt): acct.total = acct.total + amt
  - EarnInterest(acct): acct.total = acct.total * 1.02
  - Deposit is blue, EarnInterest is red
  - Transform EarnInterest into:
    - Interest = acct.total * 1.02 // runs locally at acct's copy
    - Deposit(acct, Interest) // blue operation runs at all copies

[Li et al., OSDI 2012]

Snapshot Isolation (SI)

- The history is equivalent to one of this form:

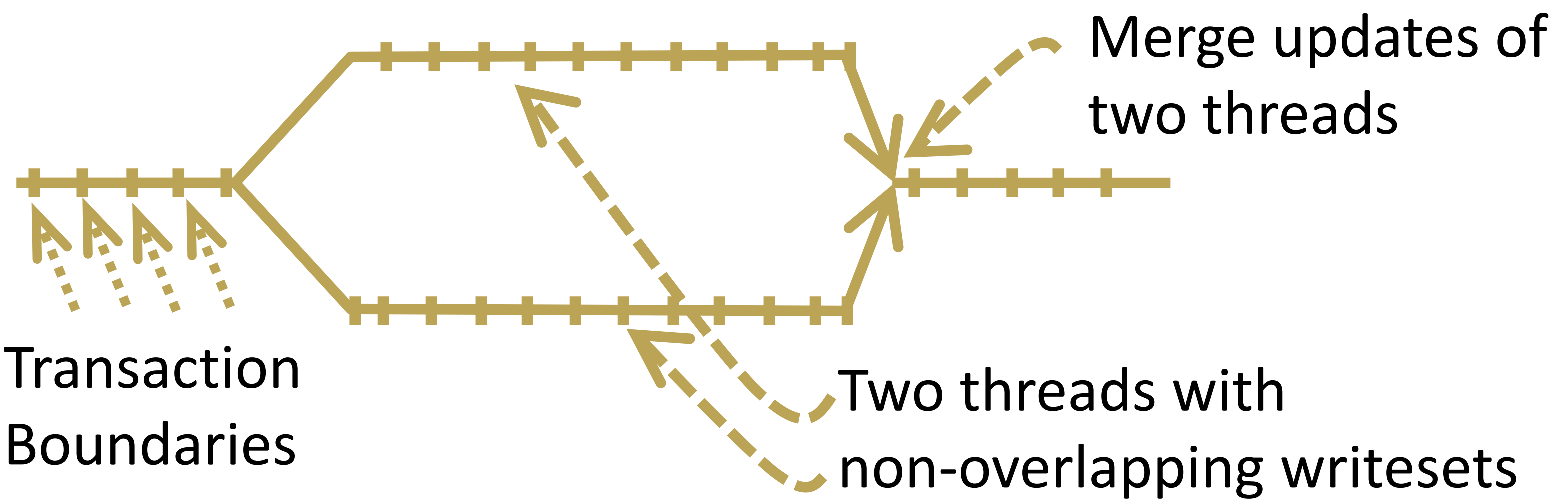
$r_1[readset_1]$ $w_1[writeset_1]$	$r_4[readset_4]$ $w_4[writeset_4]$		
$r_2[readset_2]$ $w_2[writeset_2]$	$r_5[readset_5]$ $w_5[writeset_5]$	• • •	
$r_3[readset_3]$ $w_3[writeset_3]$	$r_6[readset_6]$ $w_6[writeset_6]$		

$ws_1 \cap ws_2 \cap ws_3 = \varnothing$ $\qquad$ $ws_4 \cap ws_5 \cap ws_6 = \varnothing$

- Benefit of SI: Don’t need to test read-write conflicts

Parallel Snapshot Isolation (PSI)

- <u>Parallel SI</u> - Execution is equivalent to one that allows parallel threads with non-conflicting writesets running SI
- Allows a transaction to read stale copies



[Sovran, Power, Aguilera, & Li, SOSP 2011]

Example: Parallel SI

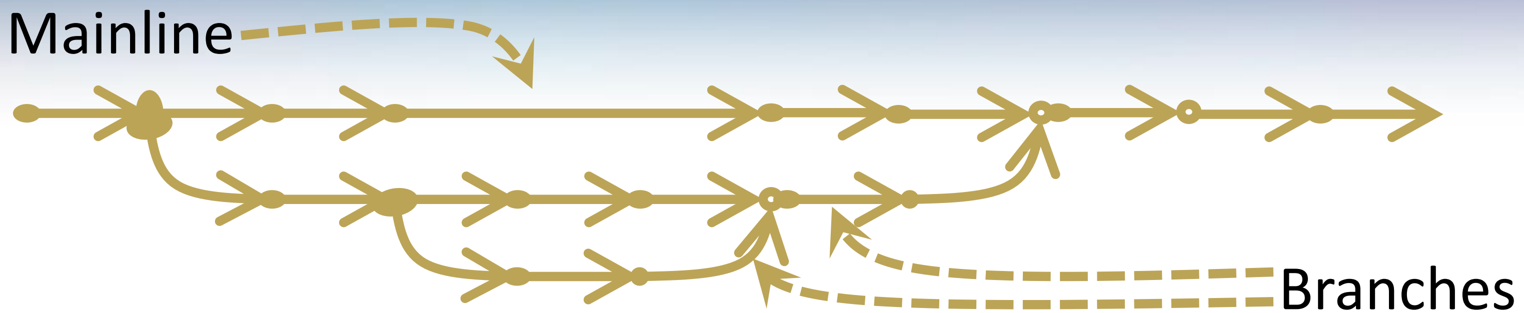
Site 1	Site 2
$r_1(x,y)$	$r_3(x,y)$
$w_1(x)$	$w_3(y)$
c_1	c_3
$r_2(x,y)$	$r_4(x,y)$
$w_2(x)$	$w_4(y)$
c_2	c_4
$w_3[y]$	$w_1[x]$
$w_4[y]$	$w_2[x]$

Site 1 has x's primary
Site 2 has y's primary

- A parallel SI execution may not be equivalent to a serial SI history
- Site 1 and Site 2 are each snapshot isolated.
- But the result is not equivalent to
  $T_1\ T_2\ T_3\ T_4$ or
  $T_3\ T_4\ T_1\ T_2$ or

T1	T2	…
T3	T4	

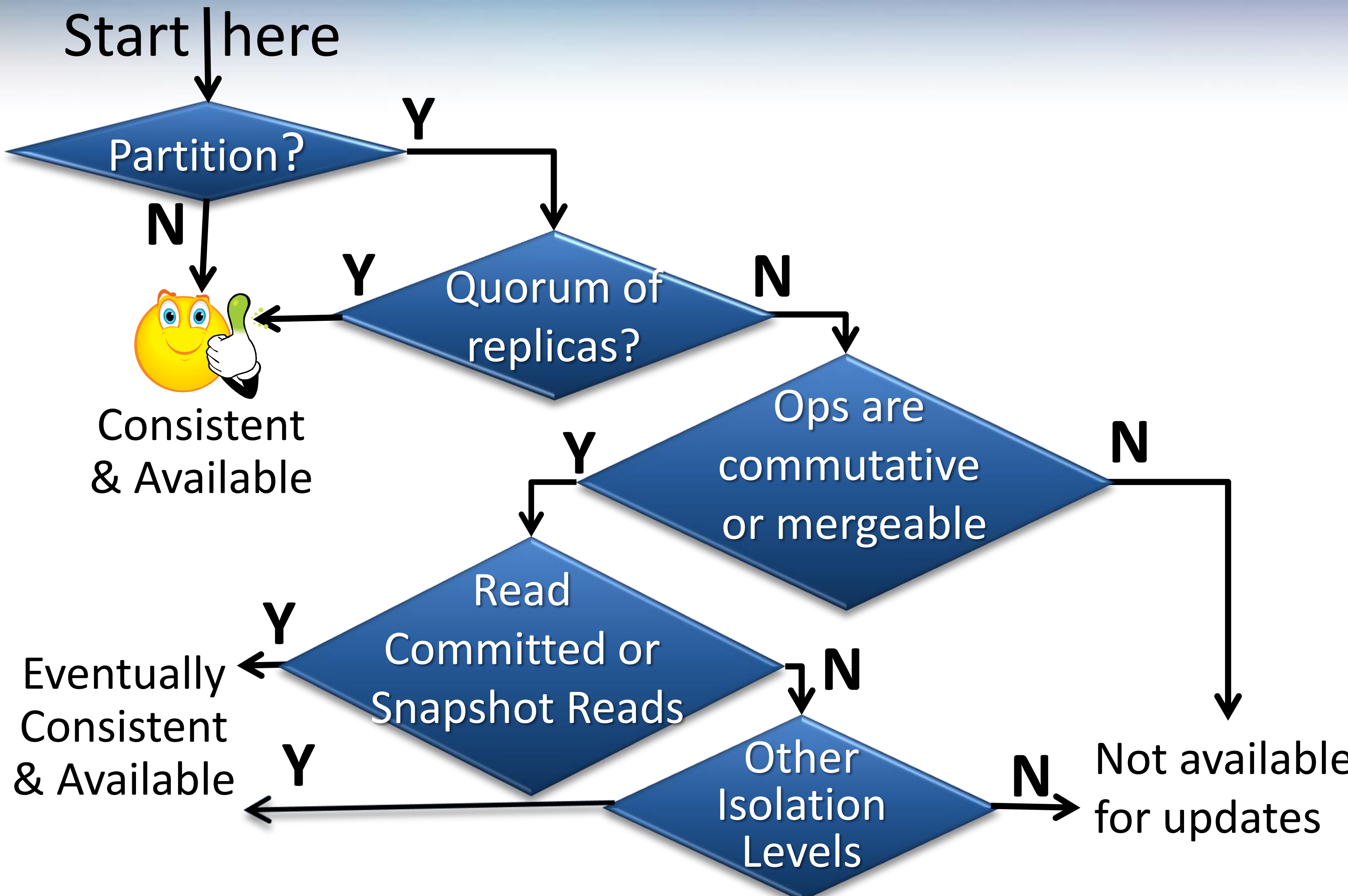
Concurrent Revisions



- Each arrow is an operation or transaction
- A fork defines a new private snapshot and a branch
- A join causes all updates on the branch to be applied
- Ops are pure reads or pure writes. Writes never fail.

[Burckhardt, et al., ESOP 2012]

In the Transaction World



RETURNING TO CAP …

Guidance for App Development

- If the system guarantees only eventual consistency, then be ready to read nearly arbitrary database states.
- Use commutative operations whenever possible.
  - System needn't totally order downstream writes, which reduces latency
- Else use convergent merges of non-commutative ops
  - Enables updates during partitioned operation and in multi-master systems

Guidance for Development (2)

- If availability and partition-tolerance are required, then consider strengthening eventual consistency with admissibility criteria
- If possible, use consistency-preserving operations, in which case causal consistency is enough
- Hard case for all admissibility criteria is rebinding a session to a different replica
  - Replica might be older or newer than the previous one it connected to.

Enforcing Admissibility in a Minority Partition

	Session maintains connection to server		Session migrates to another replica	
	Primary Copy or Quorum-based	Multi-master	Primary Copy or Quorum-based	Multi-master
Read-Your-Writes	✓	✓	✓ ?W	✓ ?W
Monotonic Writes	✓	✓	✓	✓ ?W
Bounded staleness	☹	☹	☹	☹
Consistent Prefix	✓	☹	✓	☹
Monotonic Reads	✓	✓	✓ ?R	✓ ?R
Causality	✓	✓	☹	☹

Writes disabled

?W: Only if the session caches its writes
?R: Only if the session caches its reads

Research Opportunity

- Encapsulate solutions that offer good isolation for common scenarios
  - Commutative Replicated Data Types
  - Convergent merges of non-commutative operations
  - Research: Scenario-specific design patterns
    - Overbooking with compensations
    - Queued transactions
    - • • •

Does this design space matter?

- Probably not to enterprise developers
- Spanner [OSDI 2012] “Many applications at Google … use Megastore because of its semi-relational data model and support for synchronous replication, despite its relatively poor write throughput.”
- Mike Stonebraker [blog@ACM, Sept 2010]:
  “No ACID Equals No Interest” for enterprise users
- Same comment from a friend at Amazon

So Why Bother?

- The design space does matter to Einstein-level developers of high-value applications that need huge scale out.
- People like you! ☺

Summary

- Eventual consistency
  - Commutative operations
    - Thomas' write rule
    - Convergent data types
  - Custom merge
    - Vector clocks
- Admissible executions
  - Causality constraints
  - Session constraints
    - Read your writes
    - Monotonic reads
    - Monotonic writes
    - Consistent prefix
    - Bounded staleness
  - Isolation constraints