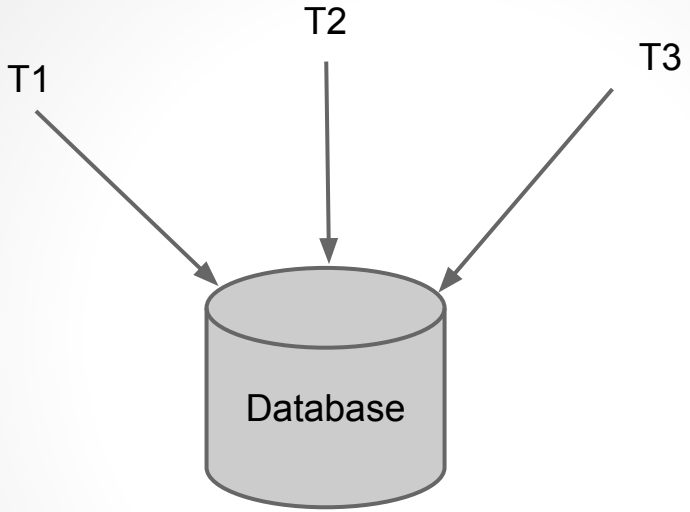
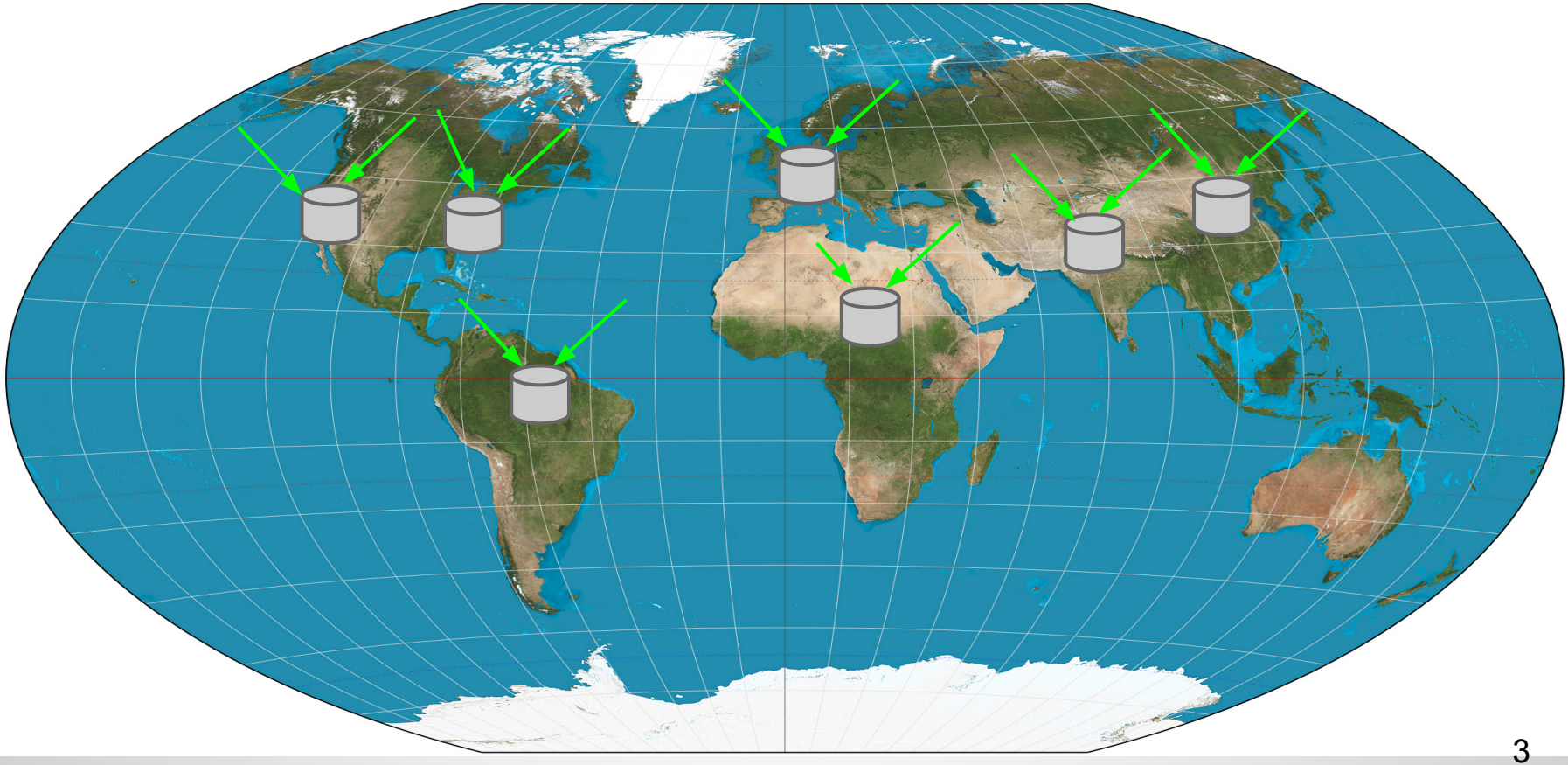
CAP Theorem and Distributed Database Consistency

Syed Akbar Mehdi
Lara Schmidt

Classical Database Model



Databases these days



Problems due to replicating data

- Having multiple copies of the data can create some problems

- A major problem is consistency i.e. how to keep the various copies of data in sync. Some problems include:
  - Concurrent writes (possibly conflicting)
  - Stale Data
  - Violation of Database Constraints

Consistency Models

- Consistency Model = Guarantee by the datastore that certain invariants will hold for reads and/or writes.

- Some standard consistency models (weakest to strongest):
  - Eventual Consistency
  - Session Consistency Guarantees
  - Causal Consistency
  - Serializability

Consistency Models

- Some standard consistency models (weakest to strongest):
  - **Eventual Consistency**
  - Session Consistency Guarantees
  - Causal Consistency
  - Serializability

- No guarantees on order of writes applied to different replicas
- No guarantees on what intermediate states a reader may observe.
- Just guarantees that “eventually” replicas will converge.

Consistency Models

- Some standard consistency models (weakest to strongest):
  - Eventual Consistency
  - **Session Consistency Guarantees**
  - Causal Consistency
  - Serializability

- Session means a context that persists across operations (e.g. between “log in” and “log out”)
- Example of a session guarantee = Read-My-Writes

Consistency Models

- Some standard consistency models (weakest to strongest):
  - Eventual Consistency
  - Session Consistency Guarantees
  - **Causal Consistency**
  - Serializability

- Operations to the datastore applied in causal order.
  - e.g. Alice comments on Bob's post and then Bob replies to her comment.
  - On all replicas, Alice's comment written before Bob's comment.
- No guarantees for concurrent writes.

Consistency Models

- Some standard consistency models (weakest to strongest):
  - Eventual Consistency
  - Session Consistency Guarantees
  - Causal Consistency
  - **Serializability**

- Global total order operations to the datastore.
- Read of an object returns the latest write.
  - Even if the write was on a different replica.

Introducing the CAP Theorem

Consistency - equivalent to having a single up-to-date copy of the data (i.e. serializability)

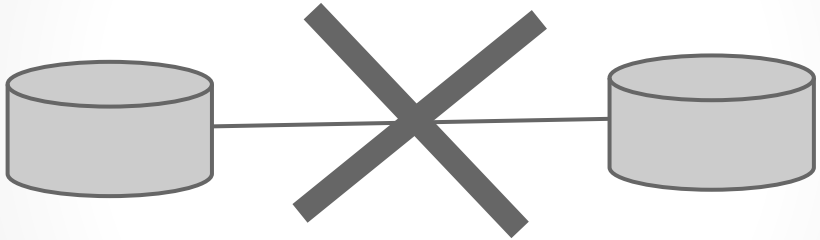
Availability - any reachable replica is available for reads and writes

Partition Tolerance - tolerance to arbitrary network partitions

Any networked shared-data system can have at most two of three of the above properties

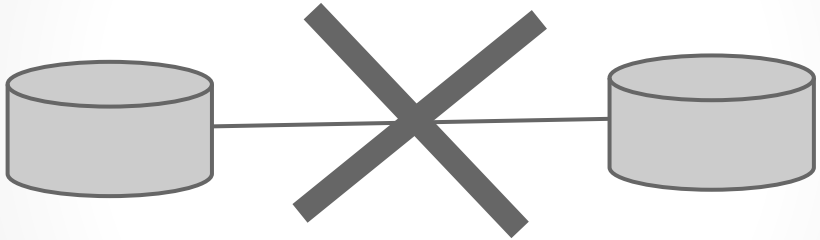
Understanding the CAP Theorem

- Imagine two replicas which are network partitioned.



Understanding the CAP Theorem

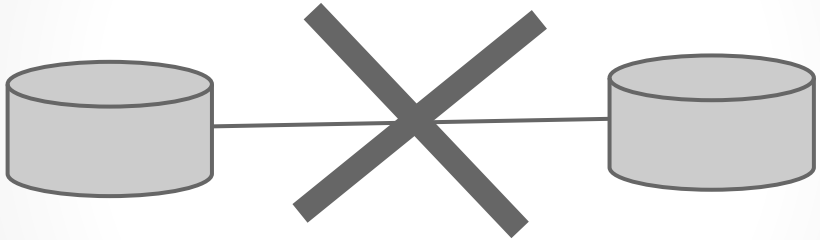
- Imagine two replicas which are network partitioned.



Allowing writes on either replica = Loss of Consistency

Understanding the CAP Theorem

- Imagine two replicas which are network partitioned.



Allowing one replica to be unavailable = Loss of availability

Understanding the CAP Theorem

- Imagine two replicas which are network partitioned.

Assuming that partitions never occur = Loss of partition tolerance

Revisiting CAP Theorem*

- Last 14 years, the CAP theorem has been used (and abused) to explore variety of novel distributed systems.

- General belief = For wide-area systems, cannot forfeit **P**

- NoSQL Movement: “Choose **A** over **C**”.
  - Ex. Cassandra - Eventually Consistent Datastore
- Distributed ACID Databases: “Choose **C** over **A**”
  - Ex. Google Spanner - provides linearizable

* from the paper “CAP 12 years later: How the rules have changed by Eric Brewer”

Revisiting CAP Theorem

- CAP only prohibits a tiny part of the design space
  - i.e. perfect availability and consistency with partition tolerance.
- “2 of 3” is misleading because:
  - Partitions are rare. Little reason to forfeit C or A when no partitions.
  - Choice between C and A can occur many times within the same system at various granularities.
  - All three properties are more continuous than binary.

Eric Brewer: **Modern CAP goal should be to "maximize combinations of consistency and availability" that "make sense for the specific application"**

Revisiting the CAP Theorem

- Recent research adopts the main idea
  - i.e. don’t make binary choices between consistency and availability
- Lets look at two examples

Consistency Rationing in the Cloud: Pay Only When It Matters

ETH Zurich, VLDB 2009

Tim Kraska
Martin Hentschel
Gustavo Alonso
Donald Kossmann

Pay Only When It Matters: Problem

Consider the information stored by a simple online market like Amazon:

- **Inventory**
  - Serializability: don’t oversell
- **Buyer Preferences**
  - Weaker Consistency: who cares if the user gets slightly more correct advertising 5 minutes later than they could.
- **Account Information**
  - Serializability: don’t want to send something to the wrong place. Won’t be updated often.

Pay Only When It Matters: Problem

Consider an online auction site like Ebay:

- **Last Minute**
  - Serializability: Database should be accurate. Want to show highest bids so people bid higher.
- **Days Before**
  - Weaker Consistency: Will be okay if data is a few minutes delayed. No high contention.

Pay Only When It Matters: Problem

Another example is a collaborative document editing application:

- **Parts of the paper which are usually done by 1 person**
  - Weaker Consistency: Since less editors there will be less conflicts and serializability isn’t as important.
  - Really just need to read your writes.
- **Parts of the paper which are highly edited**
  - Serializability: Would want parts of the document like the references to be updated often as it may be updated by many people.

Pay Only When It Matters : Problem

- How can we balance cost, consistency, and availability?
- Assume partitions.
- Don’t want your consistency to be stronger than you need
  - causes unnecessary costs if not needed.
- Don’t want your consistency to be weaker than you need
  - causes operation costs. For example, showing you are out of stock when you are not means lost sales.

Pay Only When It Matters: Solution

Avoid costs by using both!

- Serializability costs more
- Avoid costs by only using it when you really need it.
- Provide policies to users to change consistency.
- Provide 3 kinds of consistency: A, B, C

Pay Only When It Matters: Solution

- **A Consistency**
- C Consistency
- B Consistency

- Serializable
- All transactions are isolated
- Most costly
- Uses 2PL
- Used for high importance and high conflict operations.
- Ex. address information and stock count for webstore.

Pay Only When It Matters: Solution

- A Consistency
- **C Consistency**
- B Consistency

- Session Consistency
- Can see own updates
- Read-my-writes
- Used for low conflict operations that can tolerate a few inconsistencies
- For example: User preferences on web store

Pay Only When It Matters: Solution

- A Consistency
- C Consistency
- **B Consistency**

- Switches between A and C consistency
- Adaptive, dynamically switches at run-time
- Users can pick how it should change with provided policies
- For example, the auction example uses B Consistency.

Pay Only When It Matters: B Consistency

- How to switch between A and C in a way that makes sense?
  - Provide policies for switching
  - Try to minimize costs but keep needed consistency
  - 3 basic policies
    - General Policy
    - Time Policy
    - Numerical Policy

Pay Only When It Matters: B Consistency

- **General Policy**
  - Try to statistically figure out frequency of access
  - Use this to determine probability of conflict
  - Then determine the best consistency
- **Time Policy**
  - Pick a timestamp afterwhich the consistency changes.
- **Numerical Policy**
  - For increment and decrement
  - Knows how to deal with conflicts
  - Three kinds

Pay Only When It Matters: Numerical Policy

Numerical Policies : For increment and decrement

- **Fixed Threshold Policy**
  - If data goes below some point switch consistency.
  - Ex: Only 10 items left in stock, change to serializability.
- **Demarcation Policy**
  - Assign part of data to each server.
  - For example if 10 in stock, 5 servers, a server can sell 2.
  - Use serializability if want to use more than their share.
- **Dynamic Policy**
  - Similar to fixed threshold but the threshold changes
  - Threshold depends on the probability that it will drop to zero.

Pay Only When It Matters: CAP

- How can we provide serializability while allowing partitions to follow the CAP theorem? We can’t.
- If your application needs A Consistency, it won’t be available if there is a partition.
- But it will be available for the cases where your application needs C Consistency.
- Note that B Consistency can fall into either case depending on which consistency at the time.

Pay Only When It Matters: Implementation

- This specific solution uses s3 which is Amazon’s key value store which provides eventual consistency. In simplest terms can think of it as a replica per server.
- Build off of their own previous work which provides a database on top of S3 which
- However don’t really talk about how they switch consistencies and talk more about how they allow the user to tell them to switch consistencies.

Pay Only When It Matters: Summary

- Pay only what you need too.
- Allow application to switch between consistencies at runtime.
- Allow application to have different consistencies in the same database.

Highly Available Transactions: Virtues and Limitations

UC Berkeley, VLDB 2014

Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, Ion Stoica

Recap: ACID

- **ACID** = **A**tomicity **C**onsistency **I**solation **D**urability
- Set of guarantees that database transactions are processed reliably.

- The acronym is more mnemonic than precise.
- The guarantees are not independent of each other.
  - Choice of Isolation level affects the Consistency guarantees.
  - Providing Atomicity implicitly provides some Isolation guarantees.

Recap: Isolation Levels

- Isolation levels defined in terms of possibility or impossibility of following anomalies
  - **Dirty Read**: Transaction T1 modifies a data item which T2 reads before T1 commits or aborts. If T1 aborts then anomaly.
  - **Non-Repeatable Read**: T1 reads a data item. T2 modifies that data item and then commits. If T1 re-reads data item then anomaly.
  - **Phantoms**: T1 reads a set of data items satisfying some predicate. T2 creates data item(s) that satisfy T1's predicate and commits. If T1 re-reads then anomaly.

Recap: Isolation Levels

Isolation Level	**Dirty Read**	**Non-Repeatable Read**	**Phantoms**
Read Uncommitted+	Possible*	Possible	Possible
Read Committed	Not Possible	Possible	Possible
Repeatable Read	Not Possible	Not Possible	Possible
Serializable	Not Possible	Not Possible	Not Possible

+ Implicit that Dirty Writes are not allowed
* Standard does not say anything about recovery

CAP and ACID

- **C** in **CAP** = single-copy consistency (i.e. replication consistency)
- **C** in **ACID** = preserving database rules e.g. unique keys
- *C in CAP is a strict subset of C in ACID.*

- Common Misunderstanding: “CAP Theorem → inability to provide ACID database properties with high availability”.
- CAP only prohibits serializable transactions with availability in the presence of partitions.
  - No need to abandon Atomicity or Durability.
  - Can provide weaker Isolation guarantees.

ACID in the Wild

- Most research on Wide-Area Distributed Databases chooses serializability.
  - i.e. Choose **C** over **A** (in terms of CAP)
- Question: What guarantees are provided by commercial, single-site databases?
  - Survey of 18 popular databases promising “ACID”
  - Only 3 out of 18 provided serializability as default option.
  - 8 out of 18 did not provide serializability as an option at all
  - Often the default option was Read Committed.
- Conclusion: If weak isolation is acceptable for single-site DBs then it should be ok for highly available environments.

Goal of the paper

- Answers the question: “**Which transactional semantics can be provided with high availability** ? ”

- Proposes HATs (Highly Available Transactions)
  - Transactional Guarantees that do not suffer unavailability during system partitions or incur high network latency.

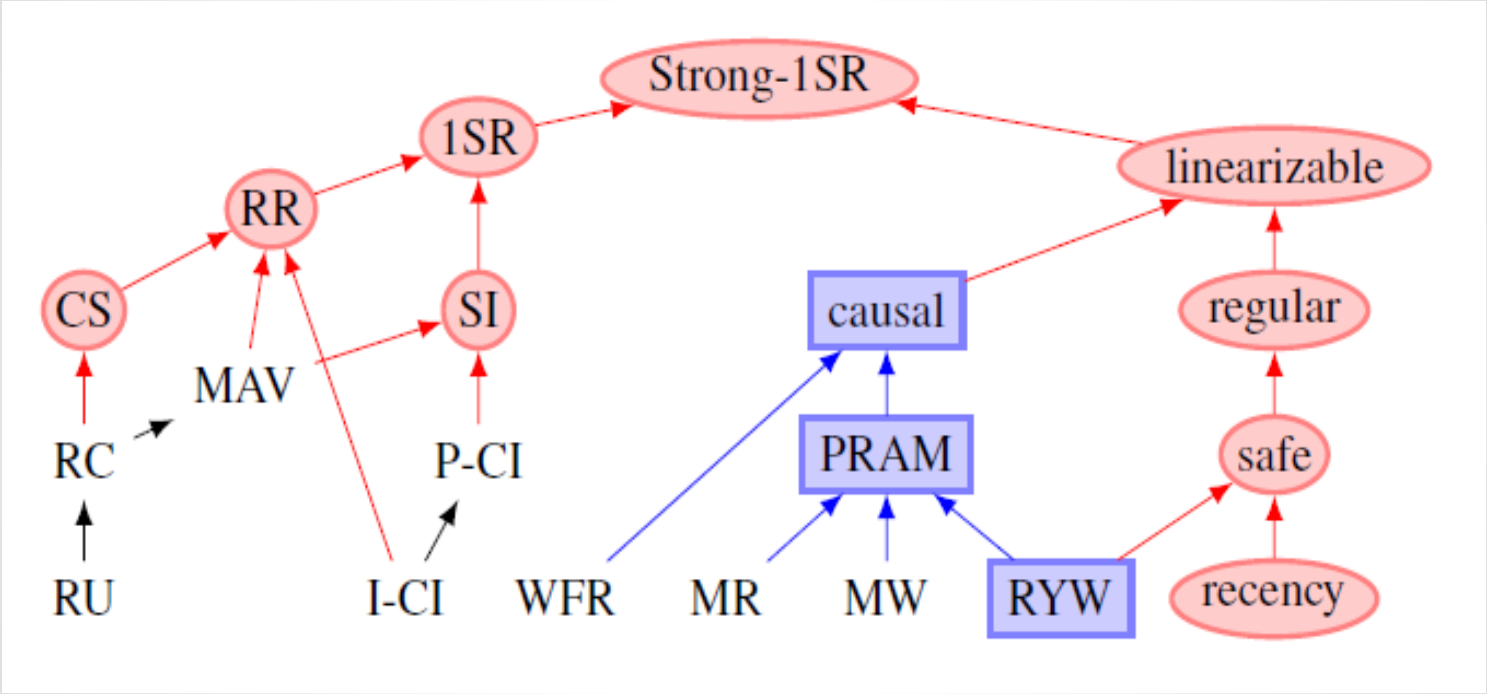
Definitions of Availability

- **High Availability**: If client can contact any correct replica, then it receives a response to a read or write operation, even if replicas are arbitrarily network partitioned.

- Authors provide a couple of more definitions:
  - **Sticky Availability:** If a client's transactions are executed against a replica that reflects all of its prior operations then …

  - **Transactional Availability:** If a transaction can contact at least one replica for every item it accesses, the transaction eventually commits or internally aborts

Overview of HAT guarantees



Example (HAT possible): Read Uncommitted

- Read Uncommitted = “No Dirty Writes”.
- Writes to different objects should be ordered consistently.
- For example consider the following transactions:

$$T1: w_1[x=1]\ w_1[y=1]$$

$$T2: w_2[x=2]\ w_2[y=2]$$

  - We should not have $w_1[x=1]\ w_2[x=2]\ w_2[y=2]\ w_1[y=1]$ interleaving on any replica.
- HAT Implementation:
  - Mark each write of a transaction with the same globally unique timestamp (e.g. ClientID + Sequence Number).
  - Apply last writer wins at every replica based on this timestamp.

Example (HAT possible): Read Committed

- Read Committed = “No Dirty Writes” and “No Dirty Reads”.
- Example: T3 should never see a = 1, and, if T2 aborts, T3 should not read a = 3:

T1: $w_1[x=1]$ $w_1[x=2]$

T2: $w_2[x=3]$

T3: $r_3[x=a]$

- HAT Implementation:
  - Clients can buffer their writes until commit **OR**
  - Send them to servers, who will not deliver their value to other readers until notified that writes have committed.
- In contrast to lock-based implementations, this does not provide recency guarantees.

Example (HAT possible): Atomicity

- Once some effects of a transaction T_i are observed by another transaction T_x , afterwards, all effects of T_i are observed by T_x

- Useful for contexts such as:
  - Maintaining foreign key constraints
  - Maintenance of derived data

- Example: T2 must observe b=c=1. However it can observe a=1 or a = _|_ (where _|_ is the initial value).

$$T1: w_1[x=1]\ w_1[y=1]\ w_1[z=1]$$

$$T2: r_2[x=a]\ r_2[y=1]\ r_2[x=b]\ r_2[z=c]$$

Example (HAT possible): Atomicity

- HAT system (Strawman implementation) :
  - Replicas store all versions ever written to every data item and gossip information about versions they have observed.
  - Construct a lower bound on versions found on every replica.
  - At start of a transaction, clients can choose read timestamp lower than or equal to this global lower bound.
  - Replicas return the latest version of each item that is not greater than the client's chosen timestamp.
  - If the lower bound is advanced along transactional boundaries, clients will observe atomicity.

- More efficient implementation in the paper.

Example (HAT sticky possible): Read-my-writes

- Read-my-writes is a session guarantee.

- Not provided by a **highly available** system.
  - Consider a client that executes the following transactions, as part of a session against different replicas partitioned from each other.

$$T1: \ w_1[x=1]$$

$$T2: \ r_2[x=a]$$

- However if a client remains sticky with one replica then this guarantee can be provided.

Examples (HAT Impossible)

- Fundamental problem with HATs is that the cannot prevent concurrent updates.
- Thus they cannot prevent anomalies like Lost Updates and Write Skew.
- Consider the following examples where clients submit T1 and T2 on opposite sides of a network partition.
  - Lost Update:

    T1: $r_1[x=100]$ $w_1[x=100 + 20 = 120]$

    T2: $r_2[x=100]$ $w_2[x=100 + 30 = 130]$
  - Write Skew:

    T1: $r_1[y=0]$ $w_1[x=1]$

    T2: $r_2[x=0]$ $w_2[y=1]$

Examples (HAT Impossible)

- Following Isolation guarantees require **no Lost Updates**:
  - Cursor Stability
  - Snapshot Isolation
  - Consistent Read
- Following Isolation guarantees require **no Lost Updates** and **no Write Skew:**
  - Repeatable Reads
  - Serializability

- As a result all of these are unachievable with high-availability.

Conclusions

- The paper provides a broad review of how ACID guarantees relate to the CAP theorem.

- Shows that a number of ACID guarantees which are provided by default in most conventional databases can be provided in a highly available environment.

- Draws a line between what ACID guarantees are achievable and not-achievable with HATs.

Summary

- The CAP Theorem is not a barrier which prevents the development of replicated datastores with useful consistency and availability guarantees
- Only prevents a tiny part of the design space
- We can still provide useful guarantees (even transactional guarantees)
- Leverage application information to maximize both availability and consistency relevant for a particular application scenario

Extra Slides

Overview of HAT guarantees

- Serializability, Snapshot Isolation and Repeatable Read Isolation are **not** HAT-compliant
  - Intuition: They require detecting conflicts between concurrent updates.
- Read Committed, Transactional Atomicity and many other weaker isolation guarantees are possible.
  - via algorithms that rely on multi-versioning and client-side caching.
- Causal Consistency possible with sticky availability.

Example (HAT possible): Cut Isolation

- Transactions read from a non-changing cut or snapshot over the data items.
- If a transaction reads the same data more than once, it sees the same value each time.
- Not quite Repeatable Read since this allows Lost Updates or Write Skew anomalies due to concurrent writes.

- HAT Implementation:
  - Clients store any read data such that the values they read for each item never changes unless they overwrite themselves.
  - Alternatively can be accomplished on sticky replicas using multi-versioning.

ACID and NewSQL Db Isolation Levels

Database	Default	Maximum
Actian Ingres 10.0/10S	S	S
Aerospike	RC	RC
Akiban Persistit	SI	SI
Clustrix CLX 4100	RR	RR
Greenplum 4.1	RC	S
IBM DB2 10 for z/OS	CS	S
IBM Informix 11.50	Depends	S
MySQL 5.6	RR	S
MemSQL 1b	RC	RC
MS SQL Server 2012	RC	S
NuoDB	CR	CR
Oracle 11g	RC	SI
Oracle Berkeley DB	S	S
Oracle Berkeley DB JE	RR	S
Postgres 9.2.2	RC	S
SAP HANA	RC	SI
ScaleDB 1.02	RC	RC
VoltDB	S	S
RC: read committed, RR: repeatable read, SI: snapshot isolation, S: serializability, CS: cursor stability, CR: consistent read		

Table 2: Default and maximum isolation levels for ACID and NewSQL databases as of January 2013 (from [8]).