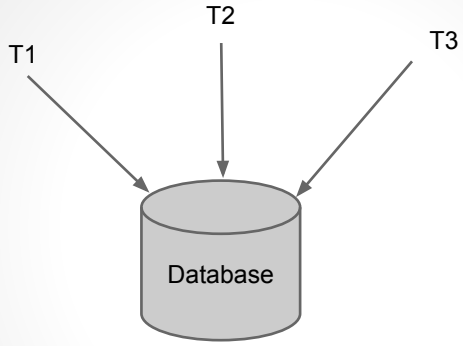
CAP Theorem and Distributed Database Consistency

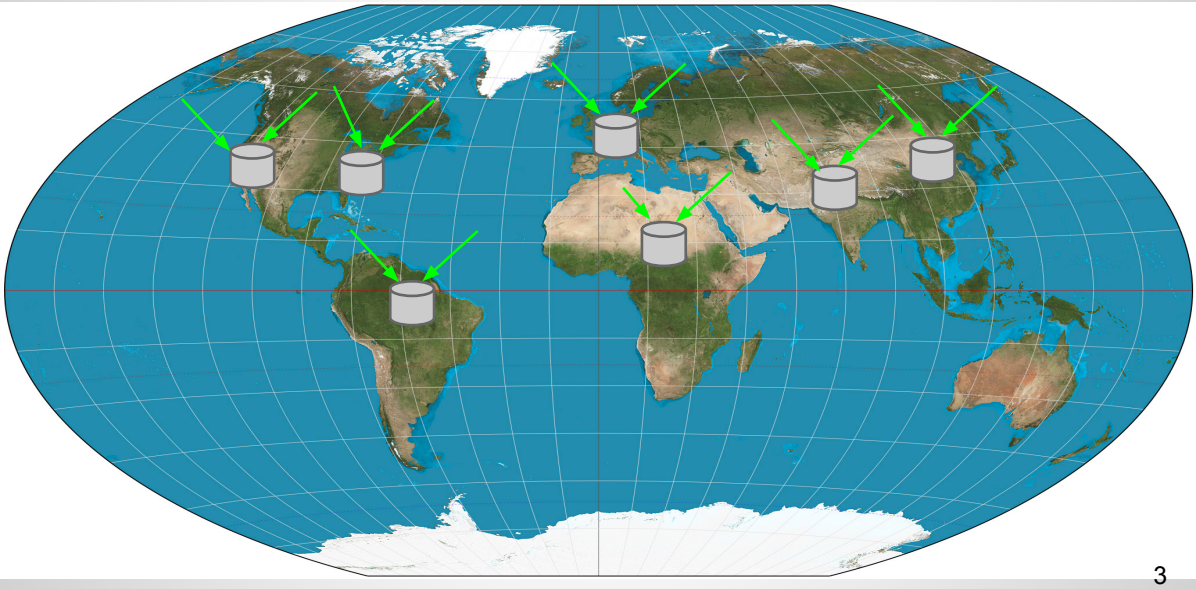
Syed Akbar Mehdi
Lara Schmidt

Classical Database Model



What we have seen so far in class is a model where multiple concurrent transactions are accessing a single copy of the database.

Databases these days



However these days databases are frequently replicated or to put it more accurately geo-replicated. There are several reasons for this:
1) Scalability is a major reason. Databases are so huge these days that a single copy cannot scale well because of the size as well as the number of people simultaneously accessing them.
2) Another reason is latency. If a user has to go halfway round the world every time to access their data then they get a poor user experience.
3) Disaster Tolerance is yet another reason.

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

So having different copies of data solves the scalability and fault tolerance problems but it creates some problems too.
What if multiple clients update the same data on different replicas at the same time. Which update do you choose ? How do you merge the updates.
Also what if you write to one replica of the data without synchronization and somebody reads the same data at another replica before the replicas have had a chance to reconcile.
You could also have a case where you violate database constraints e.g. assume you have $100 in the bank account and you make withdrawals of $75 from clients accessing different replicas. If the writes are not synchronous you could overdraw your account.

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

A consistency model is a guarantee by the datastore that certain invariants will hold as reads and writes are performed to the datastore (and this is done taking into account the fact that the datastore is replicated).

These invariants could be of different types:

a) state based invariants e.g. bank account balance never goes below zero.

b) operation based invariants e.g. operations are always applied at all replicas in causal order (causal consistency) or all operations have a total global order (i.e. 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.

Eventual consistency is the weakest model possible. Simply stated it is a liveness guarantee which says that eventually all replicas will converge to the same copy. It does not guarantee that invariants will hold either regarding the order in which writes are applied or regarding the states that can exist for the distributed data store. E.g. it is entirely possible to read a new value of an object and then an old value of the same object in a later read. The only guarantee is that eventually, possibly when writes have ceased to the system then the 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

b) Session guarantees provide real-time or client centric ordering within a session, where a session describes a context that persists across operations (or possibly transactions) e.g. on a social networking site all of a users operations submitted between “log in” and “log out”. A popular session guarantee is Read-your-writes, which means that within a session you can read any data that you have written. e.g. if you write a comment on Facebook and then you should be able to read your comment within your same login session.

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.

Causal consistency guarantees that operations will be applied to the datastore in causal order. E.g. if Alice comments on Bob's post and then Bob replies to her comment then on all replicas Alice's comment is written before Bob's comment. This also means that it should be impossible to read Bob's comment before reading Alice's comment.
There is no guarantee however for concurrent writes e.g. if two people comment on two unrelated facebook posts then it is okay to apply the two writes in different orders on different replicas. Notice however that concurrent writes can be conflicting in which case some policy needs to be specified about how to converge the two writes. e.g. if two people concurrently modify the same Facebook post. One policy that is used is last writer wins.

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.

Serializability means that operations appear as if there was a global total order or equivalently that there was a single copy of the data. This has the implication that a read of an object returns the latest write to that object, even if the write was on a different replica. Ensuring this property often means that writes have to be a synchronous operation involving multiple replicas. This also means that fundamentally concurrent writes on multiple replicas are not allowed, because serializability can potentially be violated. We will see examples of this later.

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

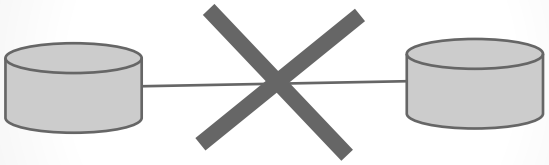
The CAP Theorem relates the Consistency model provided by a replicated data store with its availability and tolerance to network partitions. It was conjectured in 2000 by UCB Prof. Eric Brewer and was proved in 2002 by Nancy Lynch and Seth Gilbert from MIT.

Lets define each of the three terms in the CAP acronym.
Consistency means serializability i.e. equivalent to having a single up-to-date copy of the data.
Availability means that if a client can reach a replica then the replica is available for reads and writes.
Partition Tolerance means that the system can tolerate network partitions.

The theorem basically states that any networked shared-data system can provide at most two of the three properties above.

Understanding the CAP Theorem

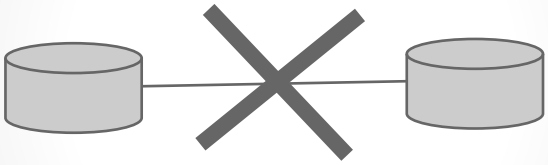
- Imagine two replicas which are network partitioned.



To understand the theorem imagine two replicas which are partitioned from each other because of a network disconnection.

Understanding the CAP Theorem

- Imagine two replicas which are network partitioned.

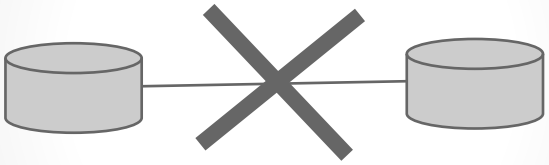


Allowing writes on either replica = Loss of Consistency

So obviously we cannot have synchronous writes in this case because of the partition. If we still allow writes on one replica without synchronizing with the other we lose one-copy serializability since the states of the replicas diverge.

Understanding the CAP Theorem

- Imagine two replicas which are network partitioned.



Allowing one replica to be unavailable = Loss of availability

On the other hand assume that replicas wait to synchronize, then we have lost availability because as long as the network partition persists we cannot write to any of the replicas.

Understanding the CAP Theorem

- Imagine two replicas which are network partitioned.

Assuming that partitions never occur = Loss of partition tolerance

Finally if we assume that partitions never occur then we have lost the property 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”

Over the last 14 years, the CAP theorem has been used to explore new distributed systems. The general belief is that for wide-area systems you can’t forfeit P or partitions. For wide-area systems you can’t choose to not have partitions.

Therefore it comes down to balancing consistency and availability. The NoSQL movement usually centers on choosing availability over consistency. For example cassandra is an eventually consistent datastore that is always available. On the other hand there is the distributed ACID database which chooses consistency over availability. For example Google Spanner provides linearizable transactions.

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"**

All in all, CAP only prevents a tiny part of the design space. The only thing it prohibits is perfect availability and consistency with partition tolerance.
The 2 of 3 in CAP is misleading for several reasons. For one, partitions are rare. You have to be able to continue when they are there, but you can assume this won’t be often. There is little reason to forfeit availability or consistency when there are no partitions. Can just forfeit C and A when partitions do exist.
Secondly the choice between C and A can occur many times within the same system at various granularities. All three properties are more continuous than a binary ‘on or off’.
A quote from Eric Brewer, the person who initially conjectured the CAp theorm says that the 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

Recent research adopts this main idea in that they don't make binary choices between consistency and availability.

Intro to next slide: So we are going to now show you two examples of papers that show how to push the boundaries of system design to achieve combinations of consistency and available given the CAP theorem.

Consistency Rationing in the Cloud: Pay Only When It Matters

ETH Zurich, VLDB 2009

Tim Kraska
Martin Hentschel
Gustavo Alonso
Donald Kossmann

Consistency Rationing in the Cloud: Pay Only when It Matters is a paper that was in VLDB 2009 by ETH Zurich. The goal of this paper is to lower the overall cost of consistency by paying only when you need to. It's goal is to minimize costs.

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.

Consider a simple online market similar to Amazon. You want the inventory count to be serializable. If two people buy the same object from different servers you don't want each server to say it is okay if you only have 1 left. Similarly, account information should be kept serializable. You don't want to send n object to the wrong house!

However consider that at the same time, the web store calculates and updates buyer preferences whenever something is bought. We don't need strong consistency for this. For one, chances are it won't be updated often from different servers. Secondly, if it is delayed or out of order with sales, it doesn't affect too much.

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.

Another example of this is an online auction site like Ebay. Consider days before the auction time runs out. Weak consistency is okay because for one there won't be a lot of contention. Maybe once or twice a day a user will check the auction and bid higher. It will be okay if it takes some time to update the value. However if the auction is in the last few minutes, the auction price is really important. You don't want one user to see it at 3$ and another to see it at 5$. Then the former user will try to bid 4$ and will have no chance of getting the object. In this case we would need something kind of ability to change consistency of the data based on some kind of policy or rule.

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.

A third example is of a collaborative document editing service. For example usually with a paper you split parts of a paper between different members of the group. Each member will work on part so there won't be a lot of conflicts. After all, it's hard to work on the exact same piece together. For this you really just need a weaker consistency since there won't be a lot of conflicts. As long as you can see your own rights you should be okay.

On the other hand there might be a few parts of the paper that are highly edited. Like for example, the bibliography. As people write their parts they want to add papers. And you don't want people adding papers twice because they didn't see the other updates in time. So you would need serializability for this to avoid conflicts.

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.

The goal of this paper is to balance costs with consistency and availability. Also, this paper assumes partitions. This is because for large scale systems partitions is a possibility and must be taken into account. It is fairly standard that you can't just assume partitions won't happen even if they don't happen often.
For every application there are usually several pieces. Different pieces require different consistencies (I'll show some examples next). Usually systems provide a single consistency throughout the whole database. For example, the whole database is eventually consistent. However if any part of your application requires serializability then you have to set up the whole database as serializable. The problem with this is that you end up paying more than you need for the parts of your application that don't require strong consistency.
However consider the opposite case. If you need serializability but you don't have it, then you will have operation costs. For example showing that you are out of stock on an item when in fact someone had returned it 30 seconds ago.

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

This paper solves this problem by allowing more than one consistency through the database. It also allows consistency of a set of data to change throughout it's use. This allows for things like the online auction site to take advantage of the database. The idea here is to avoid costs of consistency unless you really need them while also providing the guarentees you need to make your application work. They do this by providing the option of three kinds of consistency which they name A, B, C consistency. I'll also refer to them as such.

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.

A consistency is pretty basic. It is the stronger consistency and guarantees serializability. All transactions are isolated. However this is very costly. They use a 2PL protocol to implement serializability.
For example the address information and stock count for the webstore would use serializability.

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

The next consistency the offer is C consistency. This is the weaker consistency or session consistency. It is basically eventual consistency. However there is an added guarantee that if a user connects to a server for a session, they will see their own updates as long as they are connected to that session and server. This is also known as ‘read-my-writes’ guarantee as we talked about earlier.

For example user preferences on a web store would use session consistency.

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.

The final consistency they offer is B Consistency. This consistency will switch between A and C consistency depending on a user-picked policy that we will go over later. It is adaptive and can switch dynamically at run time. This is used in the auction example, to enable it to switch consistency towards the end of the auction.

One might also note that if you wanted to do a join on A consistency and C consistency data, the data would only have the weaker (or C) 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

How do they allow the user to switch between A and C consistency in a cheap way? It wouldn't be good to switch constantly between the two. They provide policies for switching. The goal is to minimize costs but keep A consistency when you need it. They provide 3 basic policies, General Policy, Time Policy, and 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

The first policy, General Policy, attempts to statistically figure out frequency of access of certain pieces of data. It uses this to determine probability of conflict. Then it uses the probability of conflict to determine the best consistency. If there are less conflicts you might not need such a strong consistency. Note that this does not work for things that might get hit hard for short periods of time because it will not be able to see that in the statistics.

Time policy is the simplest policy. You just pick a timestamp afterwhich the consistency changes.

Numerical policy is used for data that is incremented or decremented. It knows how to deal with conflicts (since inc/dec are cummulative). This is advantageous in dealing with eventual consistency or C consistency. They provide 3 basic types of numerical policies.

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.

The first is fixed threshold policy. In this policy if data goes below a certain point it will switch consistency. For example consider the previous web market. You can set it so that when you only have 10 of an item left it will switch to A consistency so that the count will be accurate and will not oversell.

The next policy is demarcation policy. In this policy you assign a part of the data to each server. For example in the case of web market, you would let each server sell a certain number of some item. Say you had 10 items left and 5 servers. Each server could sell up to 2. Use serializability if you want to use more than your share and to talk to other servers about the amounts.

The last policy is dynamic policy. It is similar to fixed threshold but the threshold changes. The threshold is determined statistically based on the probability that something will drop to zero. It uses the statistics on updates to determine this probability.

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.

One more thing to think about is how this ties in to the CAP theorem. How can we provide serializability while we allow partitions? We can't. However the goal here is to use A consistency as little as possible. And when you have to use A we assume the system will be unavailable while there is a partition. But it will be available for C consistency and then for certain times for B consistency (depending on whether B consistency is currently A consistency or C consistency).

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.

The specific solution they provide uses Amazon's key value store s3 which by itself provides eventual consistency. In simplest terms you can think of it as a replica per server. It is built off their own previous work which implements different consistencies on s3. However they don't really talk about the protocols they use to switch between consistencies and how this guarantee is provided and more focus on how they can allow users to denote the kind of needed consistency.

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.

So in summary the goal is to pay only what you need to. To do this they give a way for an application to switch between consistencies at runtime. And they allow a way for the 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.

The paper discusses the relationship between ACID and CAP. So lets quickly recap what ACID means.
ACID is an acronym that refers to a set of guarantees that database transactions are processed reliably. ACID stands for Atomicity Consistency Isolation and Durability.
Atomicity defines “all or nothing” behavior of a transaction i.e. to the outside world, a committed transaction appears (by its effects on the database) to be indivisible ("atomic"), and an aborted transaction does not happen.
Consistency ensures that any transaction will bring the database from one valid state to another. e.g. preserving a property such as unique keys.
Isolation mainly deals with concurrency control and visibility of effects of concurrently executing transactions to each other.
Durability means that once a transaction has committed it will remain that way in the event of failures, crashes etc

So one of the interesting things about this acronym is that it is more mnemonic than precise. The guarantees are not independent of each other. E.g the choice of consistency level is affected by the isolation level you choose or 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.

Lets also quickly review some standard Isolation levels since they will come up later during our presentation. So ANSI SQL standard defines isolation levels are defined in terms of the possibility or impossibility of three anomalies.

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

So the ANSI SQL specification defines 4 Isolation levels at which you can operate a concurrent database, based on the possibility or impossibility of these anomalies.
Read Uncommitted means that all three anomalies are possible. What is implicitly assumed however is that “Dirty Writes” are impossible. We will discuss this later.
Read Committed says that Dirty Read is not possible but the other two anomalies are possible
Repeatable Read only allows that Phantoms are possible
While Serializability says that none of the anomalies are possible.

Also notice that the fact that for read uncommitted, dirty reads are allowed which means that transactions are not recoverable. While the standard does not say anything about recovery, I guess the implicit assumption is that if you are running your database at this Isolation level then you do that taking into account the fact that your transactions may have read values that were written by an aborted transaction.

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.

So getting back to the paper, the main objective of the paper is to explore the relationship between CAP and ACID. So as a starting note, the C in CAP is different from the C in ACID. The C in CAP refers to replication consistency (or having a single copy of the database) which is a strict subset of ACID consistency. e.g. it is possible to have a single copy of the database at all replicas but which violates some database constraint e.g. unique keys or referential integrity of foreign keys

A common misunderstanding is that the CAP Theorem means the inability to provide ACID database properties with high availability.
However CAP only prohibits serializable transactions along with high availability in the presence of partitions.

There is no need to ab…...

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.

However despite the fact that CAP does not prevent any ACID guarantees, most research on Wide-Area Distributed Databases chooses to abandon availability entirely and focus on providing strong isolation i.e. serializability. So the authors of the paper decided to look at the the guarantees offered by conventional single-site databases. They found out that out of 18 popular databases promising ACID, only 3 provided serializability as a default option. 8 out of 18 databases did not provide serializability as an option at all. Often the default option was Read Committed. This was the case with Oracle 11g which provided Read Committed as the default option and Snapshot Isolation as the maximum option. As an aside Snapshot Isolation is an intermediate isolation guarantee provided by multiversion databases and is weaker than serializability.

So they conclude that if application writers and database vendors have already decided that the efficiency benefits of weak isolation outweigh potential application inconsistencies that can arise from running your single-site database at that isolation level, then, we can choose this for Wide-Area Distributed Databases as well.

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.

So the aim of the paper is to answer the question. If we choose Availability over Strong Consistency, what are the transactional isolation guarantees that we can provide? i.e. we know that we cannot provide serializability, but are there any other useful guarantees that we can provide.
So they propose HATs i.e. transactional guarantees that do not suffer unavailability during system partitions or incur high 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 …

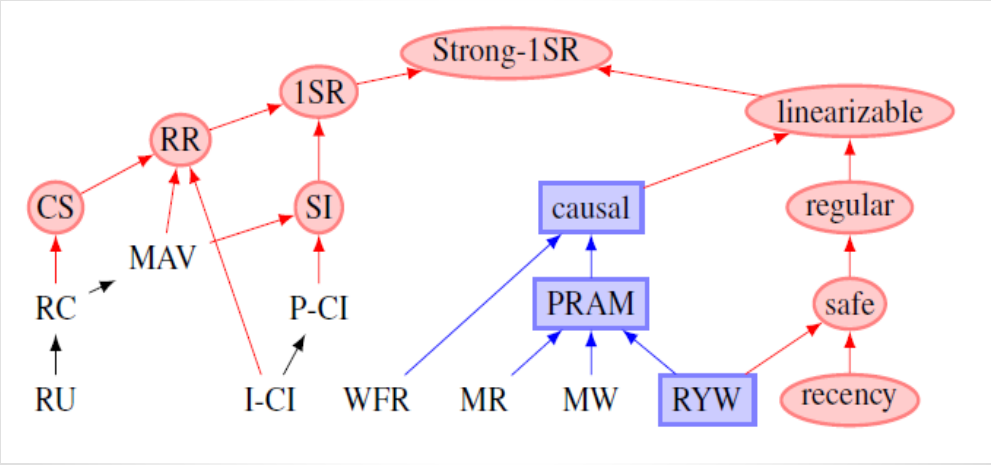
  - **Transactional Availability:** If a transaction can contact at least one replica for every item it accesses, the transaction eventually commits or internally aborts

So in order to do this they provide a couple of more nuanced definitions of availability. The traditional availability definition is that ……...

---

1) Distributed algorithms often assume a model in which clients always contact the same logical replica(s) across subsequent operations, whereby each of the client's prior operations (but not necessarily other clients' operations) are reflected in the database state that they observe. Any guarantee achievable in a highly available system is achievable in a sticky available system but NOT vice-versa.
2) Transactional Availability may result in "lower availability" than a non-transactional availability requirement (e.g., single-item availability).

Overview of HAT guarantees



41

So here is a high level overview of what they found. Now I understand these are a lot of acronyms to take in, so I will give the bigger picture rather than go in details of each. However notice that a number of them are familiar from class e.g. CS stands for Cursor Stability, RR stands for Repeatable Reads, 1SR stands for 1-copy serializability. Some I have described earlier e.g. RC is Read Committed, RYW is read-your-writes.

The arrows define a partial order on the guarantees e.g. CS is stronger than RC.
The guarantees in red color and circles are not achievable with high availability
The guarantees in blue color and squares are achievable only with sticky availability.
All others are possible with high availability.

---

1) Semantics **Not** provided by HATs are Circled and Red (Cursor Stability (CS), Snapshot Isolation (SI), Repeatable Read (RR), One-Copy Serializability(1SR), Recency, Safe, Regular, Linearizability, Strong 1SR)

2) Semantics provided with Sticky Availability are in Squares and Blue (Sticky Read Your Writes (RYW), PRAM, Causal)

3) HATs ((Read Uncommitted (RU), Read Committed (RC), Monotonic Atomic View (MAV), Item Cut Isolation (I-CI), Predicate Cut Isolation (PCI), Writes Follow Reads (WFR), Monotonic Reads (MR),

Monotonic Writes (MW))

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.

So lets look at some of these in detail.
Read Uncommitted, if you remember from the earlier slides, was the weakest isolation guarantee in the ANSI levels which allowed all three anomalies. However implicitly the one anomaly that it does not allow is “Dirty Writes”. This effectively implies that writes to different objects should be ordered consistently. So in the example if two transactions write to the same objects then we should not have an interleaving where we apply writes to x in one order and writes to y in another order.

Now assume that these two transactions are being applied to two replicas on opposite sides of a network partition. How do we make sure that when the replicas reconcile these transactions after recovery, they do not violate dirty writes.
They way to do it is to mark each write of a transaction with the same globally unique timestamp e.g. a unique ClientID plus a sequence number per client and then apply writes in order of this timestamp. This basically means that if you get a write with a lower timestamp than the last applied write to a data item then don’t apply it.

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.

Lets now look at Read Committed which if you remember from earlier is an isolation guarantee which is widely provided by default by conventional databases. The requirement for Read Committed is "No Dirty Writes" and "No Dirty Reads". So in the example shown, T3 should never see a = 1 and if T2 aborts, T3 should not see a = 3. Read Committed is enforced simply by making clients buffer their writes until commit OR even if they send them to the servers, they do not deliver their value to other readers until notified that writes have committed.
Notice however that if we use a lock-based implementation then Read Committed then the locking ensures that the values you read are the most recent. On the other hand, by buffering the writes we no longer have these guarantees. However it still satisfies Read-Committed in the implementation agnostic sense of the definition.

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).

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

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

So as mentioned earlier transactional atomicity is something that we can still enforce in a high availability environment. What that means is that once some effects of a transaction Ti are observed by another transaction Tx, afterwards all effects of Ti are observed by Tx.
This is useful for contexts such as maintaining foreign key constraints or maintaining consistency between derived data and the original relations.
So in the given example assume that all data items have the “null” original value. What we are saying is that once T2 observes a new value for a data item written by T1 then it cannot observe any old values. So in this case it should observe b = c = 1

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.

So here is a straw man implementation of this in a highly available environment. Replicas gossip information about versions of data items they have seen e.g. for every write the replica receives acks from other replicas once they have seen the write.
The lower bound is constructed when the client reads from a replica.
Lower bound advancing along transactional boundaries means that assign a write timestamp to each write similar to Read Uncommitted.

Key point is we want to maintain atomicity even if a client goes to different replicas for different objects written within a single transaction (in other words we have transactional availability). Either the client observes the versions of all items before the transaction or it observes the versions of all items after the transaction.

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.

So here is an example of a guarantee that can be provided with sticky availability but not with high availability.
As explained earlier, Read-my-writes is a session guarantee that allows you to read any writes written in your session. E.g. this could be a comment that you just made on a Facebook post. You need to be able to read your comment when you render you page again within a session.
Consider the example where the two transactions which are part of a session, execute on opposite sides of a partition. Clearly the write done by T1 cannot be read in T2.

However if we stick to the same 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: $r_2[x=0]\ w_2[y=1]$

Finally lets look at guarantees that cannot be enforced with high availability.
The fundamental problem is that in the the face of a network partition we cannot prevent concurrent updates if we want our datastore to be available as well. Thus we cannot prevent anomalies like Lost Updates or Write Skew.
Just to explain these assume we execute two transactions T1 and T2 on opposite sides of a network partition.

In the case of the Lost Update, whether a replica decides that the value of x is 120 or 130, this could not have resulted from a serial execution of these transactions.
Similarly for the Write skew, assume that we had a condition that either x or y can be one but not both. In this case our invariant would be violated. Whereas this would not happen, if we executed these serially.

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.

So a number of guarantees which require either preventing Lost Updates or both Lost Updates and Write Skew cannot be provided with high availability. These include …..

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.

1) highly available systems are fundamentally unable to prevent concurrent updates to shared data items and cannot provide recency guarantees for reads.

2) Snapshot Isolation is an isolation guarantee provided by multi-version (MV) concurrency control

a) At any time, each data item might have multiple versions, created by active and committed transactions.

b) Reads by a transaction must choose the appropriate version (and are non-blocking).

c) Each transaction reads data from a snapshot of the (committed) data as of the time the transaction started, called its Start-Timestamp

d) The transaction's writes (updates, inserts, and deletes) will also be reflected in this snapshot, to be read again if the transaction accesses (i.e., reads or updates) the data a second time.

e) Updates by other transactions active after the transaction Start-Timestamp are invisible to the transaction **(so no dirty reads)**

f) At commit time a transaction gets a Commit Timestamp.

g) A transaction can commit if no other transaction did conflicting writes during its execution i.e. [StartTimestamp, CommitTimeStamp]

e.g the following is snapshot isolation schedule: r1[x0=50] w1[x1=10] r2[x0=50] r2 [y0=50] c2 r1[y0=50] w1[y1=90] c1

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]).