Transactions

- Dr. Roger Young

Databases

CONCURRENCY CONTROL

"CONCURRENCY CONTROL deals with ensuring that the integrity of the database is preserved when it is being simultaneously accessed by more than one user"

If you've been working with MySQL on your own computer, you've been the only user of your database. In the real world though, a database may be used by thousands of users at the same time. Then what happens when two users e.g. try to update the same data at the same time ..?

Database Consistency

A database is in a CONSISTENT state if there are no contradictions between items stored in the database.

However, during the course of routine processing, it is sometimes necessary for the database to momentarily enter an inconsistent state.

Database Consistency

- The DBMS must take measures to ensure that this momentary inconsistency does not become permanent

- If another user were to access the data at this point, that user would see inconsistent data; and if user were performing an operation that updated the database, the inconsistent data might be incorporated into that update.

The Transaction Concept

- At the heart of strategies for preventing problems like these is to make the DBMS work by processing of a series of TRANSACTIONS.

- Each transaction begins with the database in a consistent state, and ends with the database in a consistent state - but may momentarily place the database into an inconsistent state due to the necessity of performing updates one after another.

Definition of a Transaction

- A transaction is a group of SQL statements that you combine into a **single logical unit of work**. By combining SQL statements like this, you can prevent certain kinds of database errors.

ACID Principles

- To preserve system consistency, we must guarantee that each transaction satisfies four requirements.

- These are called the ACID properties, after the first letters of their names; Atomicity, Consistency, Isolation, Durability

ATOMICITY

- ATOMICITY: We must guarantee that each transaction is processed ATOMICALLY - i.e. either none of it is done, or all of it is done.
- It must NOT be possible for only part of a transaction to be carried out.
- This means that if a transaction is aborted for any reason (due to a logical error in the data or a request by the user), then all effects of the transaction must be removed from the database and the database must be restored to the state it was in before the transaction was begun

CONSISTENCY

- If a transaction is executed in isolation (with no other transactions executing concurrently), and the database is in a consistent state when the transaction starts, then it will still be in a consistent state when it is finished.

ISOLATION

- Even if transactions are executing concurrently, the overall result is the same as if they executed serially - i.e. as if each transaction executed in isolation, with one transaction completing before the next begins.

DURABILITY

- Once a transaction is completed, its effects on the database must persist, even if there is a subsequent system crash.

- This may mean restoring some data that was destroyed by a crash upon restart.

Transaction States

1. **Active**: from the time it starts, until it either fails or reaches its last statement.
2. **Partially committed**: its last statement has executed, but its changes to the database have not yet been made permanent.

Transaction States

3. **Committed**: its changes to the database have been made permanent. As soon as a transaction has partially committed, the DBMS attempts to move it to the committed state - though there is no guarantee it will be able to successfully do so.

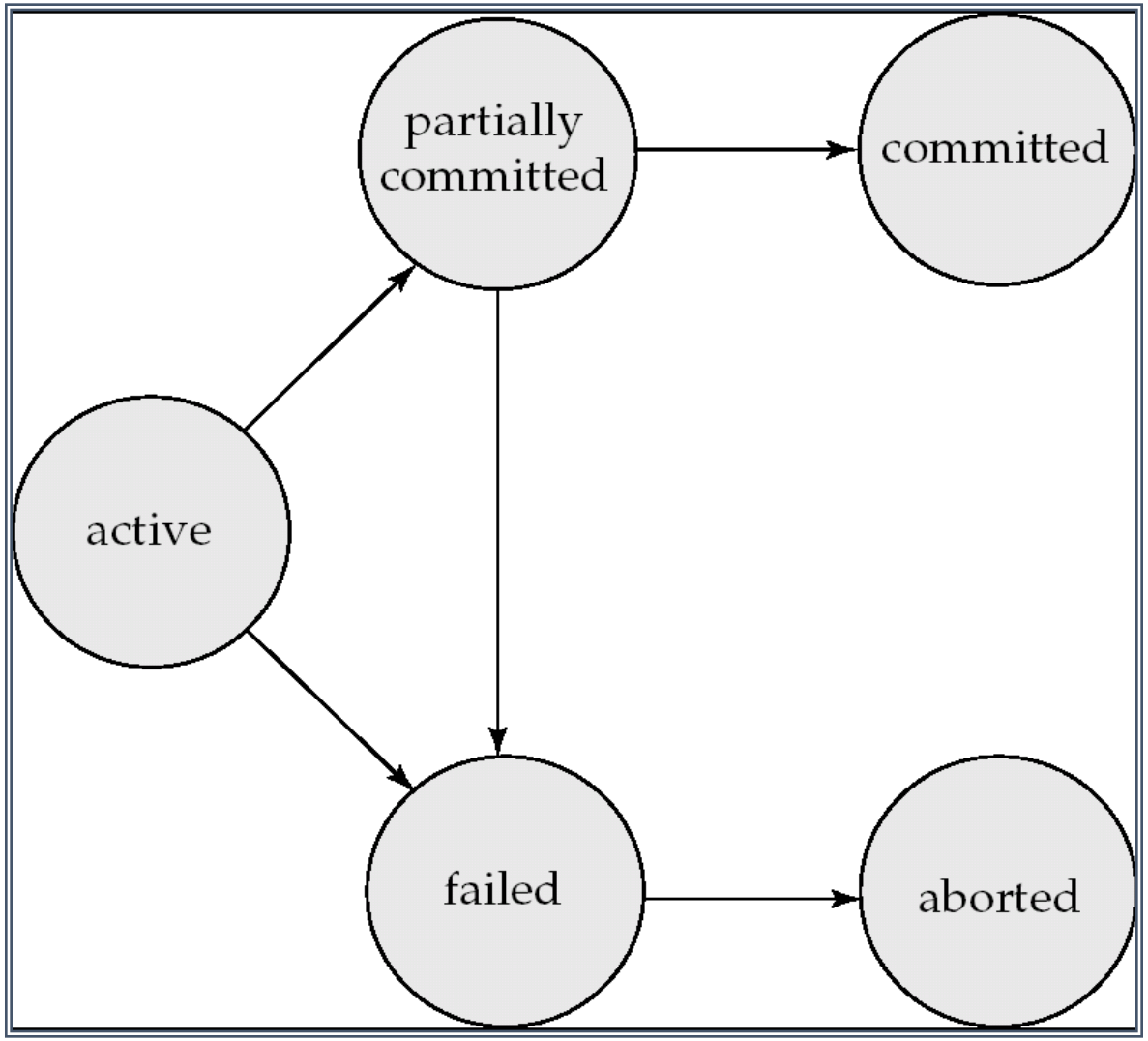
Transaction States

4. **Failed**: a logical error or user abort has precluded completion, so any changes it has made to the database must be undone.

Note:
The SQL ROLLBACK statement places the transaction into the failed state.
An SQL COMMIT that fails due to a constraint violation also places the transaction into a failed state. **Once a transaction has failed, the DBMS must move the transaction to the aborted state.**

5. **Aborted**: all effects of the transaction have been removed from the database.

Transaction State Diagram



AutoCommit Mode

- By default, a MySQL session uses the autocommit mode which automatically commits INSERT, UPDATE and DELETE statements immediately after you execute them,
- You can use transactions to change this.

Stored Procedures

- Transactions are often coded as Stored Procedures in MySQL.
- We will look at a SP named test that does not accept any parameters.
- The DELIMITER statement changes the delimiter from the default delimiter of the semicolan (;) to two slashes (//).
- This is necessary because the semicolan is used within the CREATE PROCEDURE statement and it allows you to use two front slashes to identify the end of the CREATE PROCEDURE statement.

Test SP - Error Handling

- The SP named NewOrder has 2 INSERT statements that are coded as a transaction.
- The SP declares a variable named ***sql_error*** and sets it to **FALSE** indicating that no SQL error has occurred.
- Then the second DECLARE statement creates a condition handler that sets the ***sql_error*** variable to **TRUE** if a SQL error occurs.

Test SP

- The START TRANSACTION statement identifies the start of the transaction which temporarily turns off autocommit mode.
- Then the 1st INSERT adds a new order to an orders table.
- Next, another INSERT add a line to an order_items table.

Test SP

- After the INSERT statements, an IF statement uses the SQL error variable to check whether an error occurred when executing any of the INSERT statements.
- If an SQL error did not occur, this code uses the COMMIT statement to commit the changes to the database, which makes the changes permanent.
- Otherwise the ROLLBACK statement rolls back the changes, which cancels them

Maintaining Data Integrity

- To understand why this is necessary, suppose that each of these INSERT statements is committed to the database immediately after it's executed. Then what will happen if the 2nd INSERT statement fails?
- In that case the orders and order_items tables won't match.
- In other words, the **integrity of the data won't be maintained**

```
delimiter //
create procedure NewOrder()

Begin
    declare sql_error tinyint default FALSE;
    declare continue handler for sqlexception
    set sql_error = TRUE;

    start transaction;
    INSERT INTO orders VALUES (1215, 11, '2009-11-23', '2009-11-28');

    INSERT INTO order_details VALUES(1215,1,1);
```

```
if sql_error  = FALSE then
commit;
else
    rollback;
end if;
end //
```

```
delimiter //

create procedure NewOrder()

Begin
declare sql_error tinyint default FALSE;
declare continue handler for sqlexception
        set sql_error = TRUE;
start transaction;
INSERT INTO orders VALUES (1215, 11, '2009-11-23', '2009-11-28');
INSERT INTO order_details VALUES(1215,1,1);
if sql_error  = FALSE then
        commit;
else
        rollback;
end if;
end //
```

Four types of Concurrency problems

1. **Lost Updates**
2. **Dirty Reads**
3. **Nonrepeatable reads**
4. **Phantom Reads**

Lost Updates

This occurs when two transactions select the same row and then update the row based on the values originally selected. Since each transaction is unaware of the other, the later update overwrites the earlier update

Dirty Reads

This occurs when a transaction selects data that hasn’t been committed by another transaction.

For example, transaction A changes a row. Transaction B then selects the changed row before transaction A commits the change.

If transaction A then rolls back the change, transaction B has selected data that doesn’t exist in the database.

Nonrepeatable reads

Nonrepeatable Reads Occur when two SELECT statements that try to get the same data get different values because another transaction has updated the data in the time between the two statements.

For example, transaction A selects a row. Transaction B then updates the row. When transaction A selects the same row again, the data is different.

Phantom Reads

Phantom Reads occur when you perform an update or delete on a set of rows at the same time that another transaction is performing an insert or delete that affects one or more rows in that same set of rows.

For example, transaction A updates the payment total for each invoice that has a balance due, but transaction B inserts a new, unpaid, invoice while transaction A is still running. After transaction A finishes, there is still an invoice with a balance due.

The transaction isolation level

- The simplest way to prevent concurrency problems is change the default locking behaviour. To do that, you use the SET TRANSACTION ISOLATION LEVEL statement.
- The transaction isolation level controls the degree to which transactions are isolated from one another.

Concurrency Problems prevented by each transaction isolation level

Isolation Level	Dirty Reads	Lost Updates	Nonrepeatable reads	Phantom reads
READ UNCOMMITTED	Allows	Allows	Allows	Allows
READ COMMITTED	Prevents	Allows	Allows	Allows
REPEATABLE READ	Prevents	Prevents	Prevents	Allows
SERIALIZABLE	Prevents	Prevents	Prevents	Prevents

This figure lists the 4 transaction isolation levels that MySQL provides and shows which concurrency problems they prevent or allow

Serializable

- We see from the previous table that if you use the SERIALIZABLE option, all 4 concurrency problems will be prevented.
- Each transaction is completely isolated from every other transaction and concurrency is severely restricted. The server does this by locking each resource, preventing other transactions accessing it. Since each transaction must wait for the previous transaction to commit, the transactions are executed serially, one after another.

Serializable – ctd

- Since the SERIALIZABLE level eliminates all concurrency problems, you may think that this is always the best option.
- However, this option requires more overhead to manage all of the locks, so the access time for each transaction is increased.
- For some systems, this may cause significant performance problems- typically you want to use SERIALIZABLE isolation level only for situations in which phantom reads aren’t acceptable.

READ UNCOMMITTED

- The lowest isolation level is READ UNCOMMITTED, which allows all four of the concurrency problems to occur.
- It does this by performing SELECT queries without setting any locks and without honouring any existing locks. Since this means that your SELECT statement will always execute immediately, this setting provides the best performance.
- Since other transactions can retrieve and modify the same data, however, this setting can’t prevent concurrency problems.

READ COMMITTED

- The READ COMMITTED isolation level prevents transactions from seeing data that has been changed by other transactions but not committed.
- This prevents dirty reads, but allows for other types of concurrency problems.

REPEATABLE READ

- The default isolation level for MySQL is REPEATABLE READ.
- It is a relatively high isolation level that only allows Phantom Reads

Deadlock

- A deadlock occurs when neither of two transactions can be committed because each has a lock on a resource needed by another transaction.
  - See explanation on next slide

Transaction A

```
START TRANSACTION;
UPDATE savings SET balance = balance - transfer_amount;
UPDATE checking SET balance = balance + transfer_amount;
COMMIT;
```

Transaction B (possible deadlock)

```
START TRANSACTION;
UPDATE checking SET balance = balance - transfer_amount;
UPDATE savings SET balance = balance + transfer_amount;
COMMIT;
```

Deadlock Victim

- Here, transaction A updates the savings account first, then the checking account, while transaction B updates the checking account first and then the savings account.
- Now, suppose that the first statement in transaction A locks the savings account, and the first statement in transaction B locks the checking account.
- At that point, a deadlock occurs because transaction A needs the savings account and transaction B has to be rolled back so the other can proceed, and the loser is known as a *deadlock victim*.

Four ways to prevent Deadlock

1. Don’t allow transactions to remain open very long
2. Don’t use a transaction isolation level higher than necessary
3. Make large changes when you are assured of nearly exclusive access
4. Take locking behaviour into consideration when coding your transactions

Don’t allow transactions to remain open very long

- **Keep transactions short** – the longer you leave a transaction open and uncommitted, the more likely it is that another transaction will need to work with that same resource

- When coding your transaction make sure to include the appropriate **COMMIT** and **ROLLBACK** statements. Don’t code statements that take a long time to execute between the START TRANSACTION statement that starts the transaction and the COMMIT or ROLLBACK statement that finishes the transaction.

- Keep SELECT statements outside of the transaction except when absolutely necessary

- Never code requests for user input during a transaction

Don’t use a transaction isolation level higher than necessary

The transaction isolation level controls the degree to which transactions are isolated from one another.

The higher you set the isolation level, the more likely it is that two transactions will be unable to work with the same resource at the same time.

The default level of REPEATABLE READ is usually acceptable, but you should consider changing to READ COMMITTED if deadlock becomes a problem.

Make large changes when you can be assured of nearly exclusive access

- You should schedule transactions that modify a large number of rows to run when no other transactions, or only a small number of other transactions, will be running. That way, it's less likely that the transactions will try to change the same rows at the same time.
- If you need to change millions of rows in an active table, don't do so during hours of peak usage. If possible, give yourself exclusive access to the database before making large changes.

Take locking behaviour into consideration when coding your transaction

- If you need to code two or more transactions that update the same resources, code the updates in the same order in each transaction. – see transactions on next slide

Update Statements that illustrate Deadlocking

UPDATE statements that illustrate deadlocking

Transaction A

START TRANSACTION;

UPDATE savings SET balance = balance - transfer_amount;

UPDATE checking SET balance = balance + transfer_amount;

COMMIT;

Transaction B (possible deadlock)

START TRANSACTION;

UPDATE checking SET balance = balance - transfer_amount;

UPDATE savings SET balance = balance + transfer_amount;

COMMIT;

Transaction B (prevents deadlocks - note order)

START TRANSACTION;

UPDATE savings SET balance = balance + transfer_amount;

UPDATE checking SET balance = balance - transfer_amount;

•Database systems traditionally support ACID requirements:

- Atomicity, Consistency, Isolation, Durability

•In distributed web applications the focus shifts to:

- Consistency, Availability, Partition tolerance

•CAP theorem - At most two of the above can be enforced at any given time.

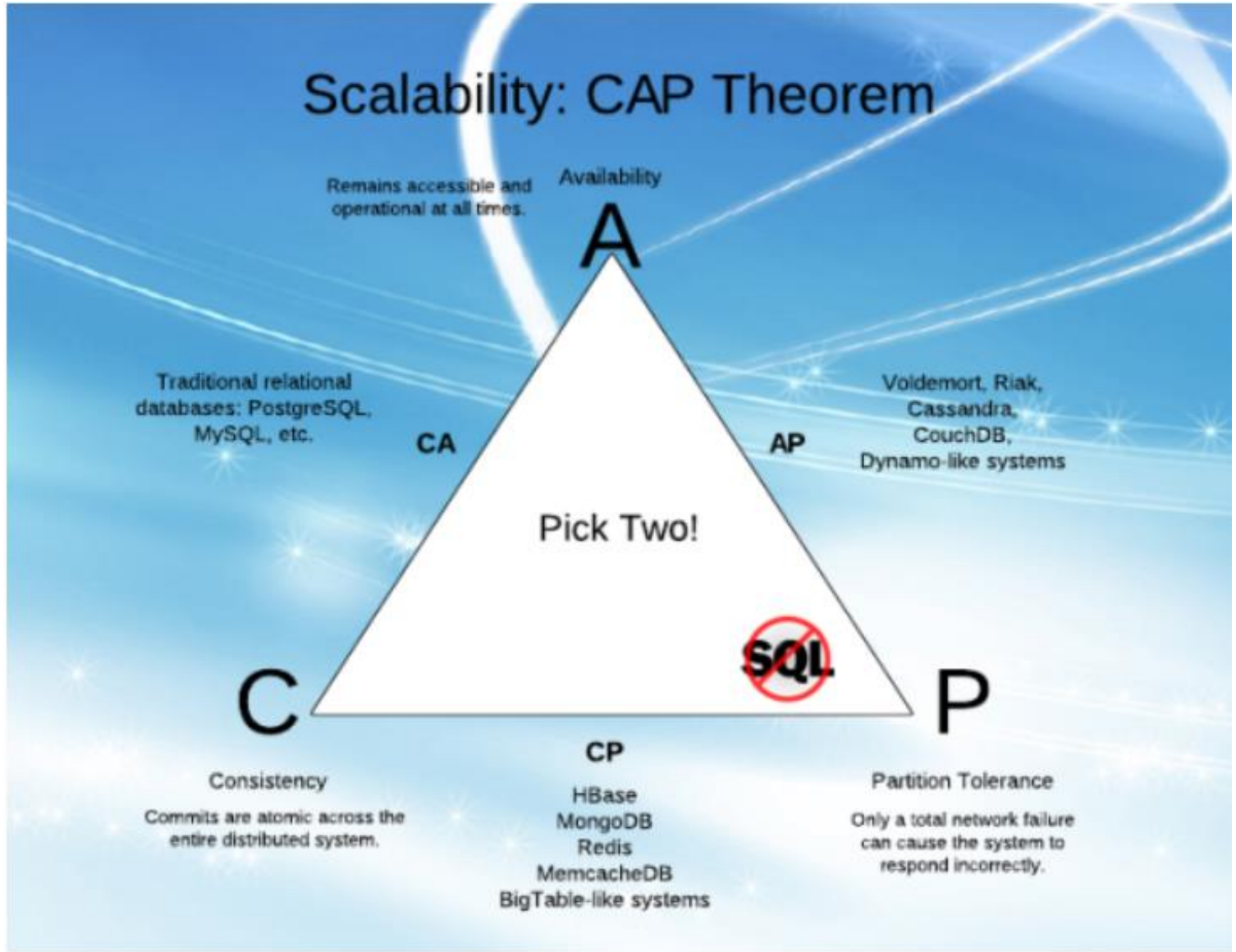
- Conjecture – Eric Brewer, ACM Symposium on the Principles of Distributed Computing, 2000.
- Proved – Seth Gilbert & Nancy Lynch, ACM SIGACT News, 2002.

•Reducing consistency, at least temporarily, maintains the other two.

The CAP Theorem

- The limitations of distributed databases can be described in the so called the CAP theorem
  - **C**onsistency: every node always sees the same data at any given instance (i.e., strict consistency)
  - **A**vailability: the system continues to operate, even if nodes in a cluster crash, or some hardware or software parts are down due to upgrades
  - **P**artition Tolerance: the system continues to operate in the presence of network partitions

CAP theorem: any distributed database with shared data, can have *at most two* of the three desirable properties, C, A or P


Scalability: CAP Theorem
Availability
A
Remains accessible and operational at all times.
Traditional relational databases: PostgreSQL, MySQL, etc.
CA
AP
Voldemort, Riak, Cassandra, CouchDB, Dynamo-like systems
Pick Two!
SQL
C
P
CP
Consistency
Commits are atomic across the entire distributed system.
HBase
MongoDB
Redis
MemcacheDB
BigTable-like systems
Partition Tolerance
Only a total network failure can cause the system to respond incorrectly.

- **CA** - data is consistent between all nodes - as long as all nodes are online - and you can read/write from any node and be sure that the data is the same, but if you ever develop a partition between nodes, the data will be out of sync (and won't re-sync once the partition is resolved).
- **CP** - data is consistent between all nodes, and maintains partition tolerance
- **AP** - nodes remain online even if they can't communicate with each other and will resync data once the partition is resolved, but you aren't guaranteed that all nodes will have the same data (either during or after the partition)

Large-Scale Databases

- When companies such as Google and Amazon were designing large-scale databases, 24/7 Availability was a key
  - A few minutes of downtime means lost revenue
- When *horizontally* scaling databases to 1000s of machines, the likelihood of a node or a network failure increases tremendously
- Therefore, in order to have strong guarantees on Availability and Partition Tolerance, they had to sacrifice “strict” Consistency (*implied by the CAP theorem*)

Trading-Off Consistency

- Maintaining consistency should balance between the strictness of consistency versus availability/scalability
  - Good-enough consistency *depends on your application*

Trading-Off Consistency

- Maintaining consistency should balance between the strictness of consistency versus availability/scalability
  - Good-enough consistency *depends on your application*

Easier to implement,
and is efficient

Generally hard to implement,
and is inefficient

The BASE Properties

- The CAP theorem proves that it is impossible to guarantee strict Consistency and Availability while being able to tolerate network partitions

- This resulted in databases with relaxed ACID guarantees

- In particular, such databases apply the BASE properties:
  - **B**asically **A**vailable: the system guarantees Availability
  - **S**oft-State: the state of the system may change over time
  - **E**ventual Consistency: the system will *eventually* become consistent

Eventual Consistency

- A database is termed as *Eventually Consistent* if:
  - All replicas will *gradually* become consistent in the absence of updates
  - When no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent

ACID vs. BASE

- **Distributed NoSQL systems are typically said to support some form of BASE:**
  - **Basic Availability**
  - **Soft state**
  - **Eventual consistency***

- ***"We'd really like everything to be structured, consistent and harmonious,…, but what we are faced with is a little bit of punk-style anarchy. And actually, whilst it might scare our grandmothers, it's OK..."***
- ***-Julian Browne***

- Everyone who builds big applications builds them on CAP and BASE: Google, Yahoo, Facebook, Amazon, eBay, etc