

Big Data Systems

Djellel Difallah

Spring 2023

Lecture 4 – Transactions, Column Stores,
Parallel DBMSs, CAP

Outline

1. Transactions
2. Column Stores
3. Parallel DBMSs
4. CAP

Transactions

- Major concept and component in database systems
- Turing awards to database researchers:
 - Charles Bachman 1973 (DBMS)
 - Edgar Codd 1981 (Inventing the relational database model)
 - Jim Gray 1998 (Inventing transactions)
 - Mike Stonebraker 2014 (RDBMS)

Transaction Example

START TRANSACTION

UPDATE Budget SET money = money - 100

WHERE pid = 1

UPDATE Budget SET money = money + 60

WHERE pid = 2

UPDATE Budget SET money = money + 40

WHERE pid = 3

COMMIT

ROLLBACK

- If the app gets to a place where it can't complete the transaction successfully, it can execute **ROLLBACK**
- This causes the system to “abort” the transaction
 - Database returns to a state without any of the changes made by the transaction

Reasons for Rollback

- User changes their mind (“ctl-C”/cancel)
- Explicit in program, when app program finds a problem
 - e.g. when qty on hand < qty being sold
- System-initiated abort
 - e.g. System crash

ACID Properties

- **Atomicity**: Either all changes performed by transaction occur or none occurs
- **Consistency**: A transaction as a whole does not violate integrity constraints (only valid tuples are written)
- **Isolation**: Transactions appear to execute one after the other in sequence
- **Durability**: If a transaction commits, its changes will survive failures

What Could Go Wrong?

- Why is it hard to provide ACID properties?
- **Concurrent** operations
 - Isolation problems
- **Failures** can occur at any time
 - Atomicity and durability problems
- Transaction may need to **abort**

Different Types of Problems

Client 1: **INSERT INTO** SmallProduct(name, price)
SELECT pname, price
FROM Product
WHERE price <= 0.99

DELETE Product
WHERE price <=0.99

Client 2: **SELECT** count(*)
FROM Product

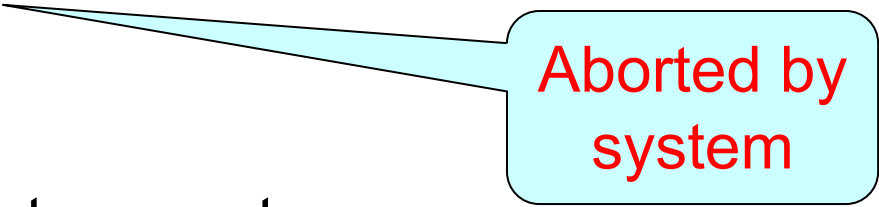
SELECT count(*)
FROM SmallProduct

What could go wrong ?

Inconsistent reads

Different Types of Problems

Client 1: **UPDATE SET** Account.amount = 1000000000
 WHERE Account.number = 'my-account'



Aborted by
system

Client 2: **SELECT** Account.amount
 FROM Account
 WHERE Account.number = 'my-account'

What could go wrong ?

Dirty reads

Different Types of Problems

Client 1:

```
UPDATE Product  
SET Price = Price - 1.99  
WHERE pname = 'Gizmo'
```

Client 2:

```
UPDATE Product  
SET Price = Price*0.5  
WHERE pname='Gizmo'
```

What could go wrong ?

Lost update

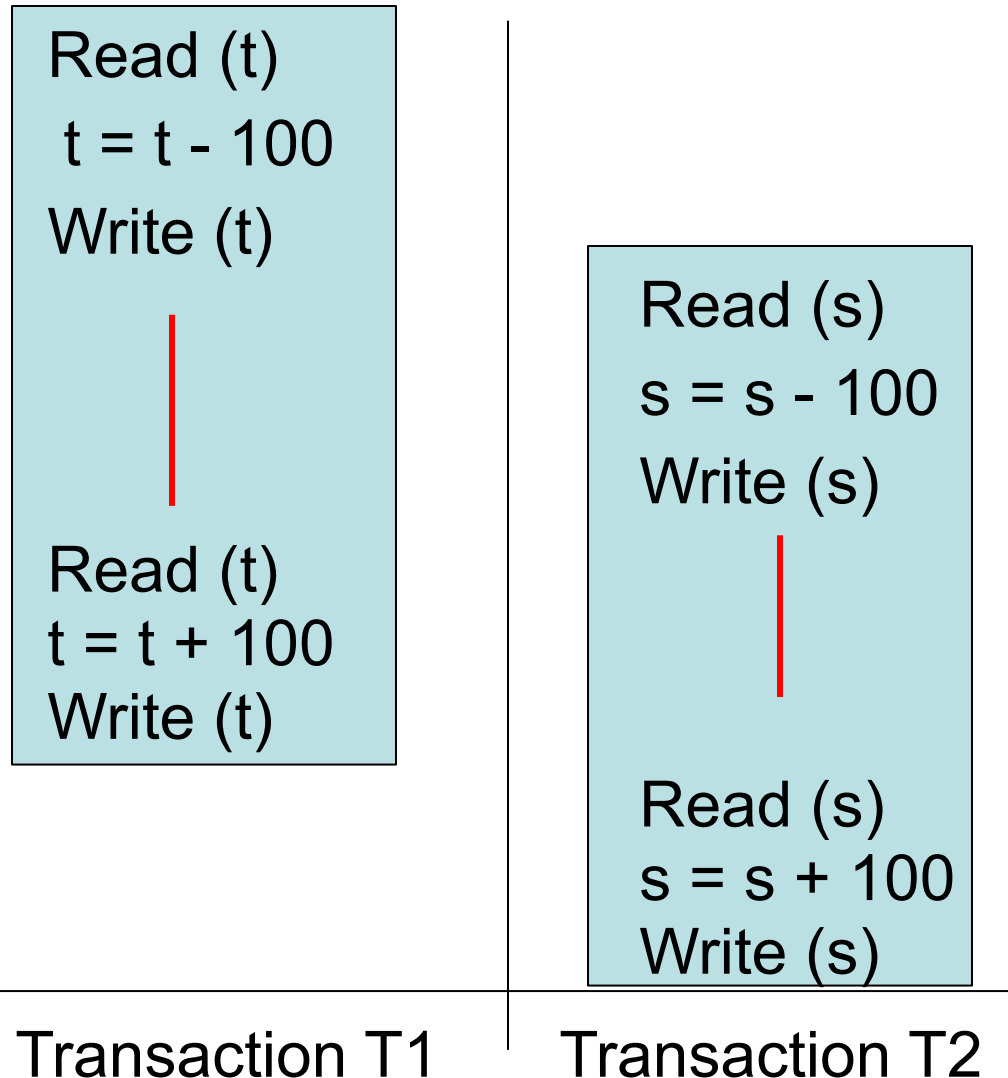
Types of Problems

- Concurrent execution problems
 - Write-read conflict: inconsistent read, dirty read
 - A transaction reads a value written by another transaction that has not yet committed
 - Read-write conflict: unrepeatable read / Phantom read (see later)
 - A transaction reads the value of the same object twice. Another transaction modifies that value in between the two reads
 - Write-write conflict: lost update
 - Two transactions update the value of the same object. The second one to write the value overwrite the first change
- Failure problems
 - DBMS can crash in the middle of a series of updates
 - Can leave the database in an inconsistent state

Serializable Execution

- **Serializability**: ensures that a schedule for executing concurrent transactions is equivalent to one that executes the transactions serially in some order

Is This Serializable?



Equivalent Serial Schedule

Read (t)
 $t = t - 100$
Write (t)
Read (t)
 $t = t + 100$
Write (t)

Read (s)
 $s = s - 100$
Write (s)
Read (s)
 $s = s + 100$
Write (s)

Transaction T1

Transaction T2

Locking

- A lock is a synchronization mechanism to limit access to a resource
- Can serve to enforce serializability
- A process can acquire a lock on a resource (ie. data)
- Lock contention
 - occurs whenever one process or thread attempts to acquire a lock held by another process or thread
- Two types of locks: **Shared (read-only) and Exclusive (read and write)**
- Implementation and granularity of locks depends on the DBMS implementation

Deadlocks

- Two or more transactions are waiting for each other to complete
- **Deadlock avoidance and detection**
 - Acquire locks in pre-defined order
 - Acquire all locks at once before starting
 - Lock timeouts
 - *Wait-for graph* Algorithm
 - This is what commercial systems use

Two-Phase Locking (2PL)

- Two simple phases:
 - 1) acquire *all* locks (do processing) [**grow**]: all locks are acquired and no locks are released
 - if any lock is not acquired on the first attempt, gives up all locks and start again
 - 2) release locks [**shrink**]: all locks are released and no locks are acquired
- Guarantees serializability
- If **strictly** applied → avoids deadlock
 - no process is ever in a state where it is holding some shared resources, and waiting for another process to release a shared resource which it requires

Phantom Problem

- A “phantom” is a tuple that is invisible during part of a transaction execution but not all of it.
- Example:
 - T0: reads list of books in catalog
 - T1: inserts a new book into the catalog
 - T0: reads list of books in catalog
 - New book will appear!
- Can this occur?
- Depends on locking details (eg, granularity of locks)
- To avoid phantoms needs **predicate locking**
 - Instead of locking records, lock predicates

Lock Granularity

- **Fine granularity locking** (e.g., tuples)
 - High concurrency
 - High overhead in managing locks
- **Coarse grain locking** (e.g., tables)
 - Less overhead in managing locks
 - Many false conflicts
- Alternative techniques exist
 - Hierarchical locking
 - Lock escalation

Degrees of Isolation

- Isolation level “serializable” (i.e. ACID)
 - Golden standard
 - Requires strict 2PL and predicate locking
 - But often too inefficient
 - Imagine there are only a few update operations and many long read operations
- Weaker isolation levels
 - Sacrifice correctness for efficiency
 - Often used in practice (often **default**)
 - Sometimes are hard to understand

Degrees of Isolation

- **Four levels of isolation**

- All levels use **long-duration exclusive locks**
- **READ UNCOMMITTED**: no read locks (dirty reads, i.e., tuples not yet committed, can occur)
- **READ COMMITTED**: short duration read locks (non-repeatable reads can occur)
- **REPEATABLE READ**:
 - Long duration read locks on individual items, not on ranges (phantom reads might occur)
- **SERIALIZABLE**:
 - All transactions occur in a completely isolated fashion (i.e., as if all transactions in the system had executed serially, one after the other)

- **Trade-off: consistency vs concurrency**

- Commercial systems give **choice** of level + **others**

Your MySQL instance?

- `SHOW VARIABLES;`
 - Inspect configuration options such as `innodb_buffer_pool_size`
- `SELECT @@transaction_ISOLATION;`
 - Default?
- MySQL's storage engines: `INNODB` vs `MyISAM`

Optimistic Concurrency Control

- Assumes that multiple transactions can complete their work without affecting each other
 - No locking
- Before committing, transactions checks that no other transaction has modified its data
 - If a conflict is detected, roll back
- Hence used in **low data contention** environments

Outline

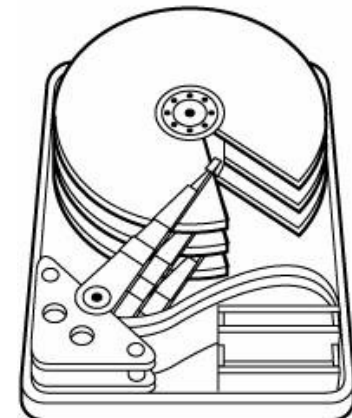
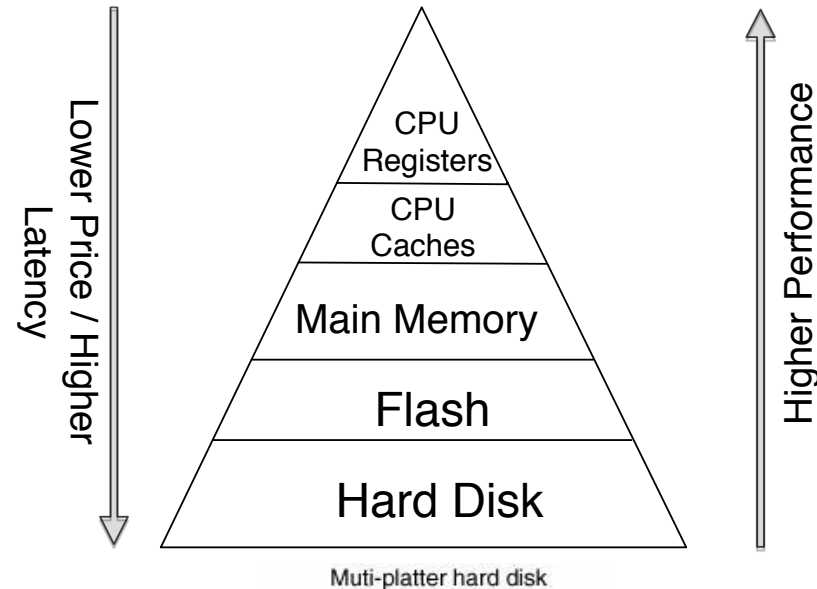
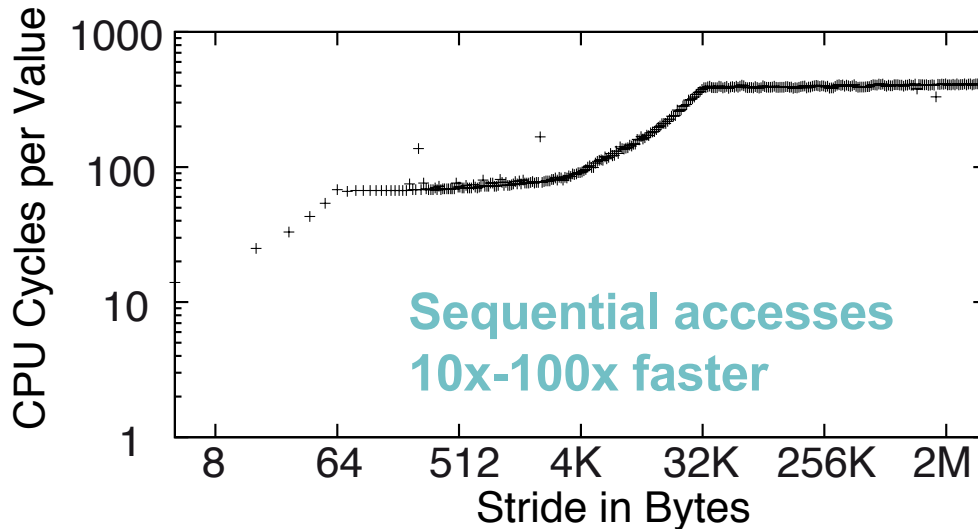
1. Transactions
2. Column Stores
3. Parallel DBMSs
4. CAP

Introduction

- What are the *major* cost factors in query processing?
 - Size of database
 - Index or not
 - Join, sort
- Current hardware configuration
 - Cheap storage – allow distributed redundant data store
 - Fast CPUs/GPUs – compression/decompression
 - Limited disk bandwidth – reduce I/O

Introduction

- Computer Architecture: deep memory hierarchies!
- Disk seeks *extremely* slow
- Avoid loading data that is not accessed (OLAP, analytics)



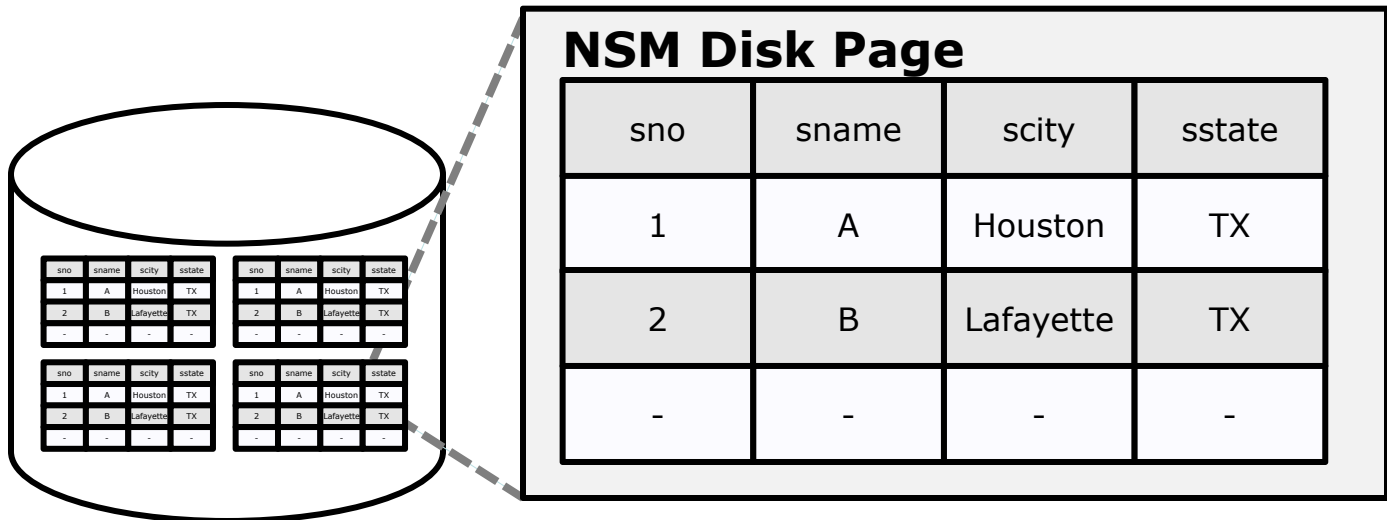
Data Storage Models

- There are different ways to store tuples.
- We have been assuming the ***n*-ary storage model** so far

n-ary Storage Model (NSM)

- The DBMS stores all attributes for a single tuple contiguously in a block.

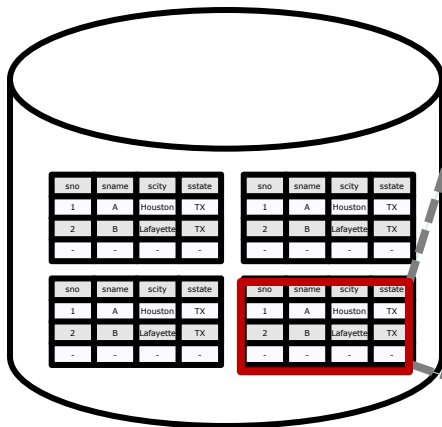
Supplier(sno, sname, scity, sstate)



n-ary Storage Model

SELECT * FROM SUPPLIER
WHERE sname= '**A**'

INSERT INTO SUPPLIER
VALUES (**?**,**?**,...**?**)

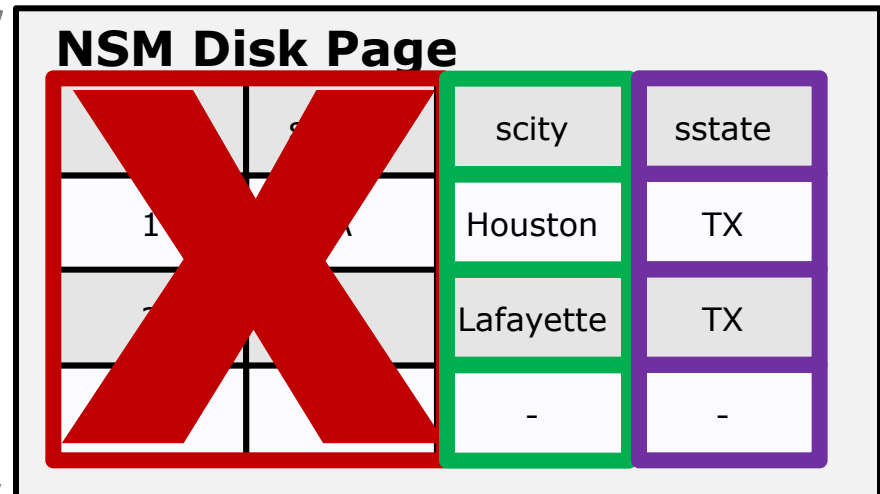
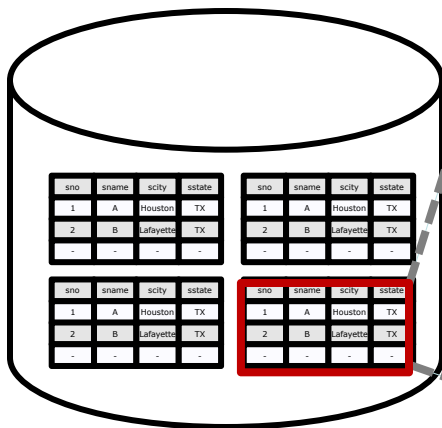


NSM Disk Page

sno	sname	scity	sstate
1	A	Houston	TX
2	B	Lafayette	LA
-	-	-	-

n-ary Storage Model

```
SELECT COUNT(SCITY),  
        SSTATE  
FROM SUPPLIER  
GROUP BY SSTATE
```



Column Stores: Main Idea

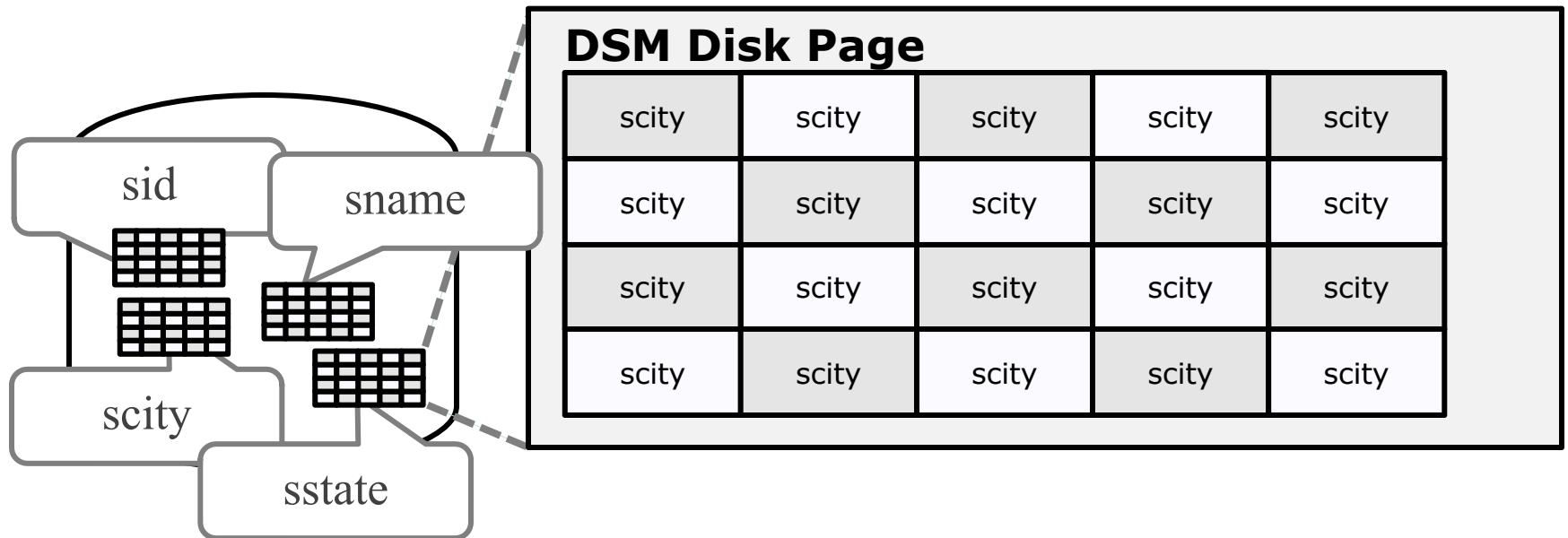
- N-ary storage (standard DBs) implies that all attributes are always stored and read together
- Data analytics (OLAP operations slice/dice) often require to analyze/aggregate attributes in isolation
 - It is not uncommon to have tables with hundreds of columns (e.g., fact tables)
 - Thus, would make sense to store each attribute separately!

Column Stores: Main Idea

- Simple idea but has profound consequences
 - Rewrite/rethink the whole database (incl. storage, indexes, optimizations, etc.)
 - Yet, keep the same API (SQL) and data model (relational data)
 - Success stories (e.g., Vertica and MonetDB; all main database vendors now have a column store)

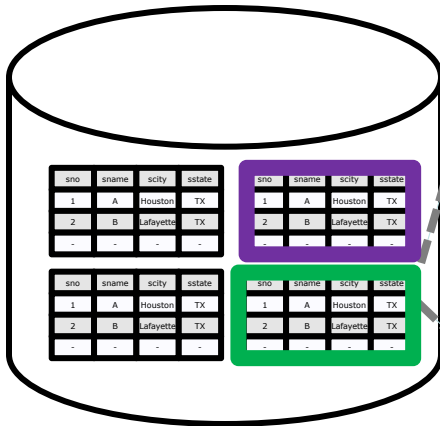
Decomposition Storage Model

- The DBMS stores a single attribute for all tuples contiguously in a block.



Decomposition Storage Model

```
SELECT COUNT(SCITY),  
       SSTATE  
FROM SUPPLIER  
GROUP BY SSTATE
```



DSM Disk Page

scity	scity	scity	scity	scity
scity	scity	scity	scity	scity
scity	scity	scity	scity	scity
scity	scity	scity	scity	scity

From Row-Stores to Column-Stores

ID	Name	Dpt	Salary

ID	Name	Dpt	Salary

- + Easy to add new tuples
- + Good for Queries that require entire tuples
- Might read unnecessary data

- + Read relevant data only
- Tuple write is expensive

Example (2 attributes relation)

Row-based
(4 pages)

Page {

A	1
A	2
A	2
A	2
B	2
B	4
C	4
C	4

Column-based
(4 pages)

A	1
A	2
A	2
A	2
B	2
B	4
C	4
C	4

} Page

Virtual IDs vs. Offsets

- How do we put the tuples back together?
 - Compute offsets (moves some complexity to the CPU)
 - Virtual ids in columns (joins)

	sid	sname	scity	sstate
0				
1				
2				
3				
4				
5				
6				
7				

Offsets

sid	sname	scity	sstate
0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4
5	5	5	5
6	6	6	6
7	7	7	7

Virtual Ids

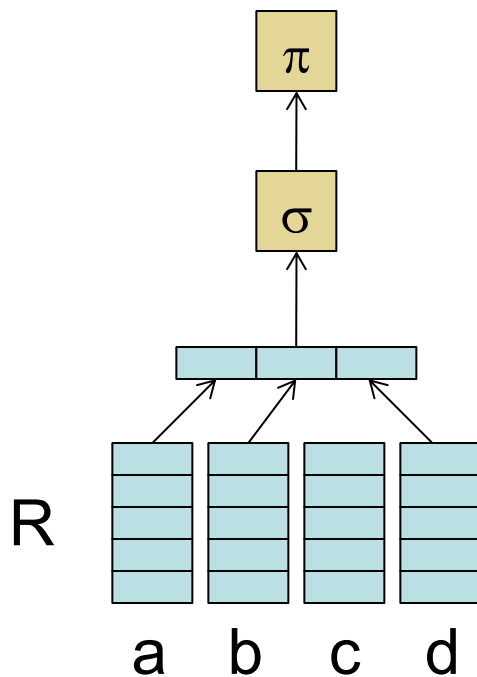
Column-Store Optimizations

- The system **sequentially** reads the **data needed [only]** (2x to 100x?)
- **Late tuple materialization**
 - Process individual columns as long as possible
 - Merge columns into complete tuples as late as possible
- **Block iteration**
 - Pass blocks of values between ops instead of individual tuples
- **Compression**: e.g., run-length encoding (RLE) of columns

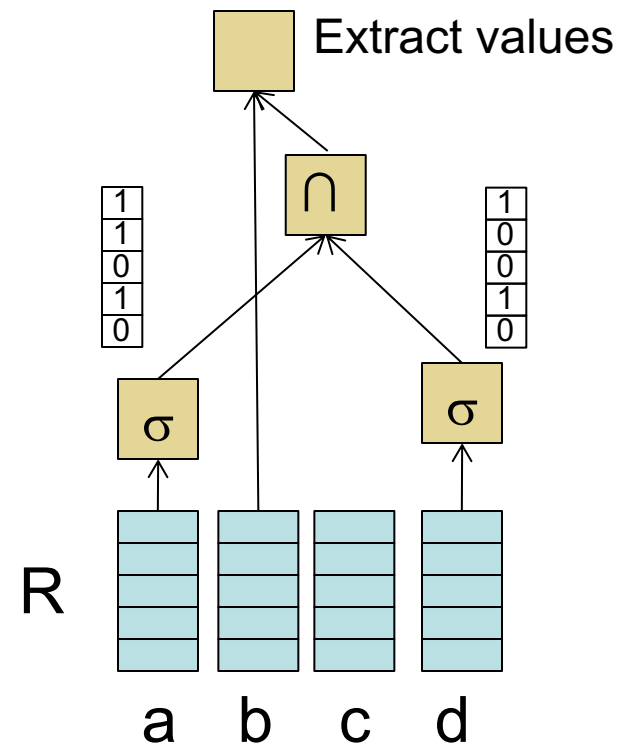
Late Tuple Materialization

Ex: `SELECT R.b from R where R.a=X and R.d=Y`

Early materialization



Late materialization



Compression Example

Row-based
(4 pages)

Page {

A	1
A	2

A	2
A	2

B	2
B	4

C	4
C	4

Column-based
(4 pages)

A
A
A
A

B
B
C
C

1
2
2
2

2
4
4
4

} Page

Compressed
(2 pages)

4XA
2XB
2XC

1X1
4X2
3X4

Column Stores

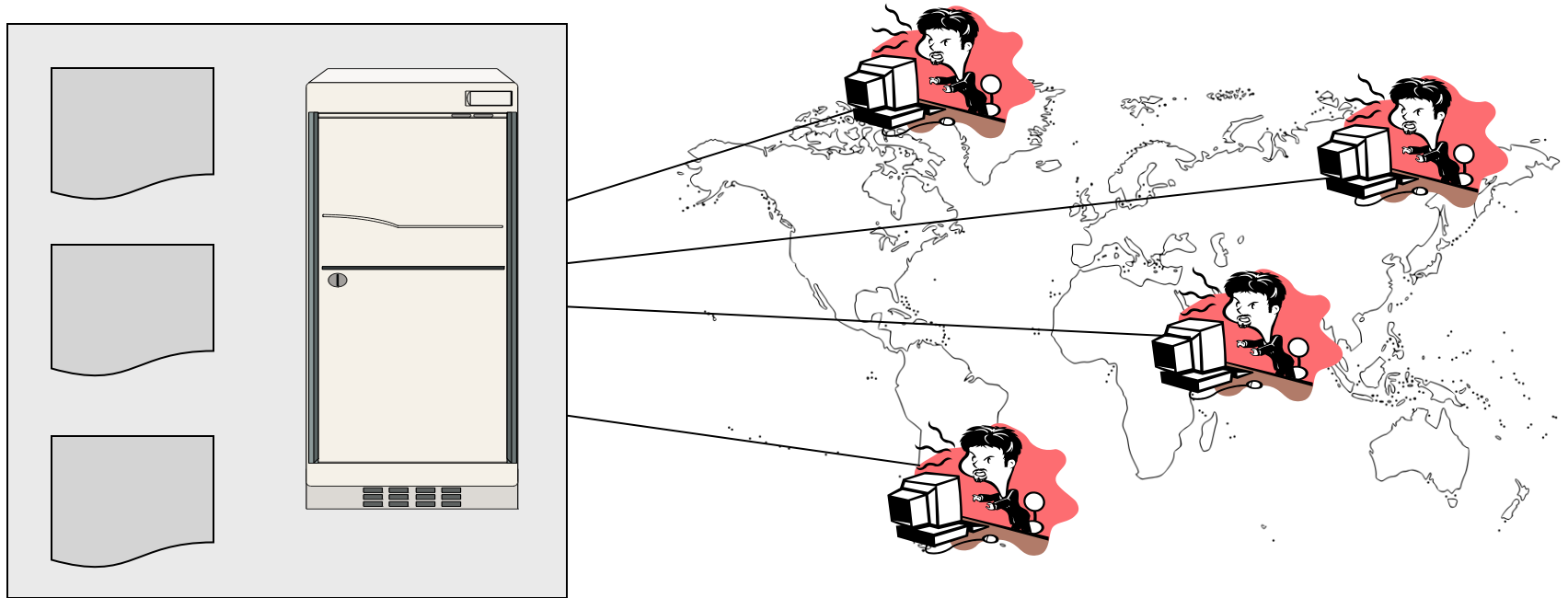
- Install MonetDB
 - <https://www.monetdb.org/Downloads>

Outline

1. Transactions
2. Column Stores
3. Parallel DBMSs
4. CAP

Review DBMS Architecture

- Client-server



Parallel DBMSs

- Goal
 - Improve **performance and scalability**, without sacrificing advances in DBMSs
- Key benefit
 - Cheaper to scale than relying on a single increasingly more powerful processor
 - Executing multiple operations in parallel
- Key challenges
 - **ACID** compliance
 - Ensure **overhead** and contention do not kill performance

Performance Metrics for Parallel DBMSs

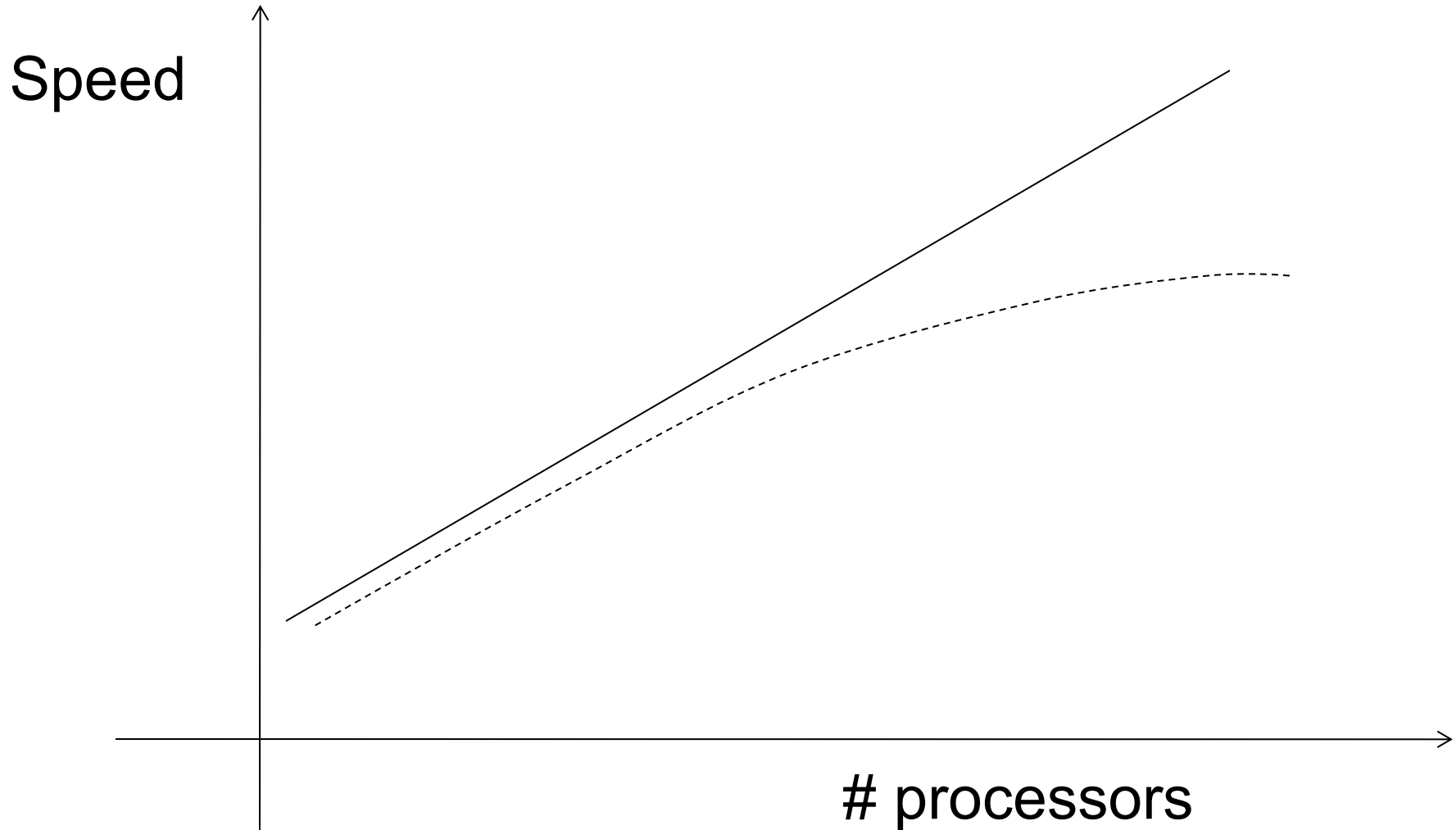
- Speedup

- More processors → higher speed
- Individual queries should run faster
- Should do more transactions per second (TPS). Throughput!

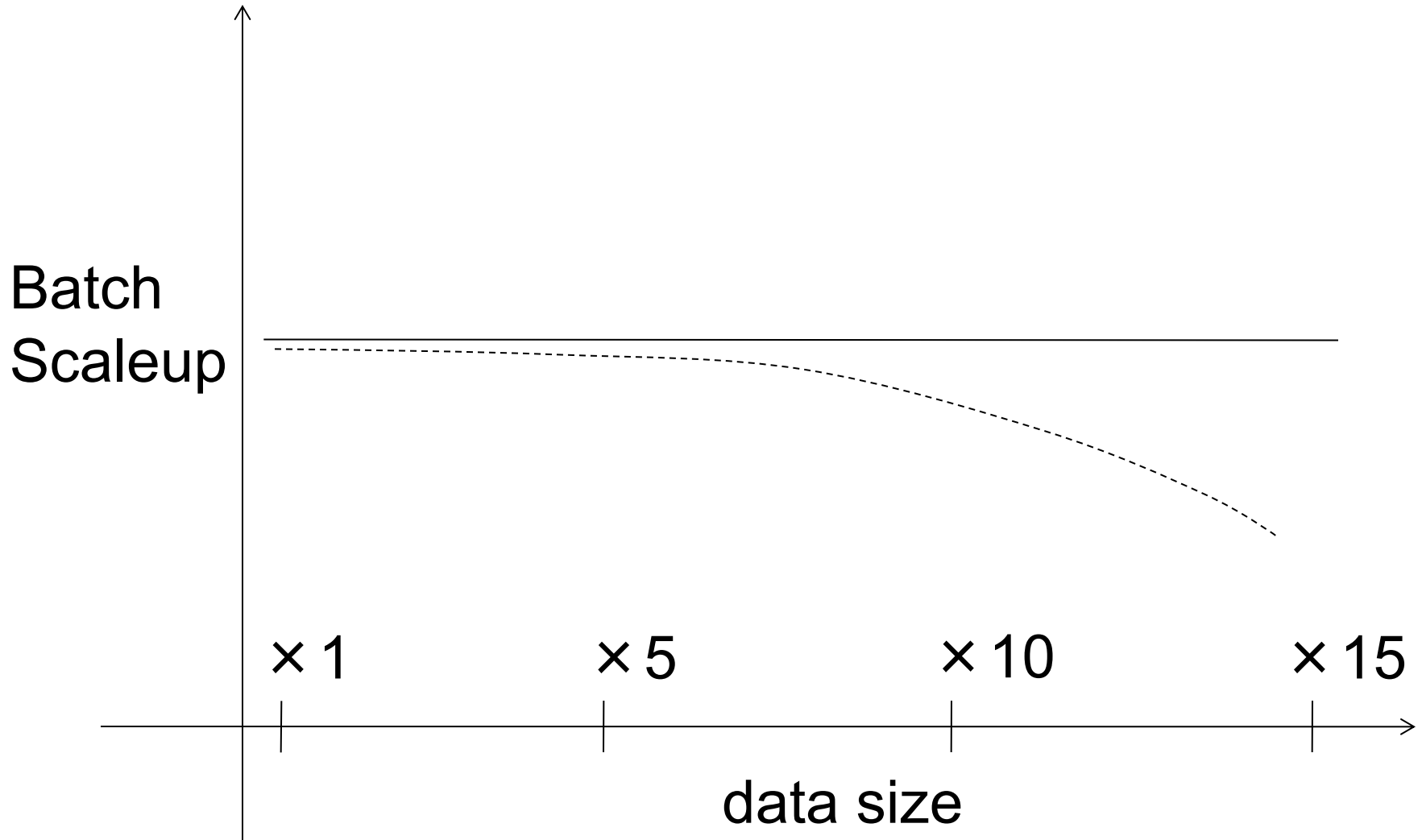
- Scaleup

- More processors (or nodes or servers) → can process more data
- Batch scaleup
 - Approximately same execution time of queries on larger input data
- Transaction scaleup
 - N-times as many TPS on N-times larger database
 - But each transaction typically remains small

Linear vs Non-linear Speedup



Linear vs Non-linear Scaleup



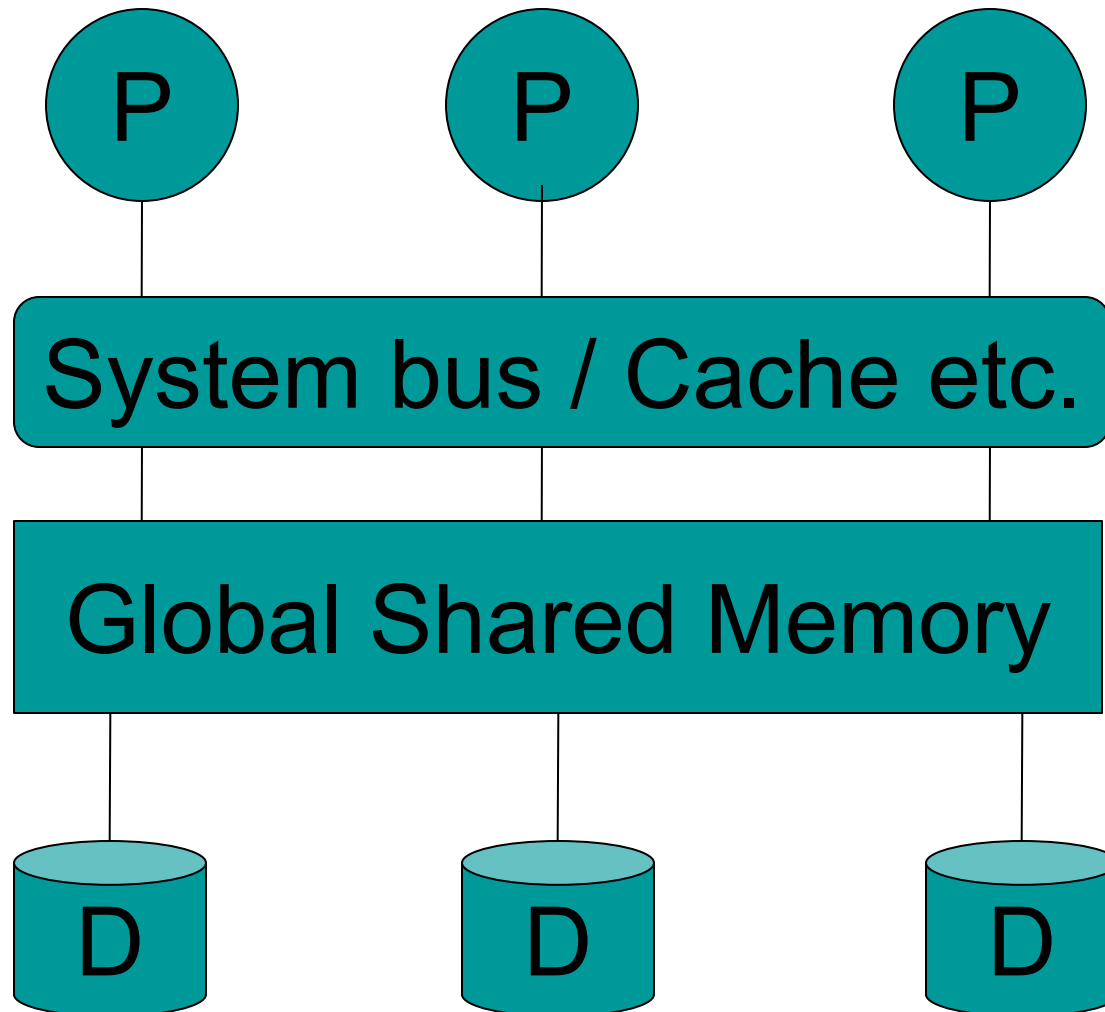
Challenges to Linear Speedup and Scaleup

- **Startup cost**
 - Cost of starting an operation on many processors
- **Interference**
 - Contention for resources between processors
- **Skew**
 - Slowest step becomes the bottleneck

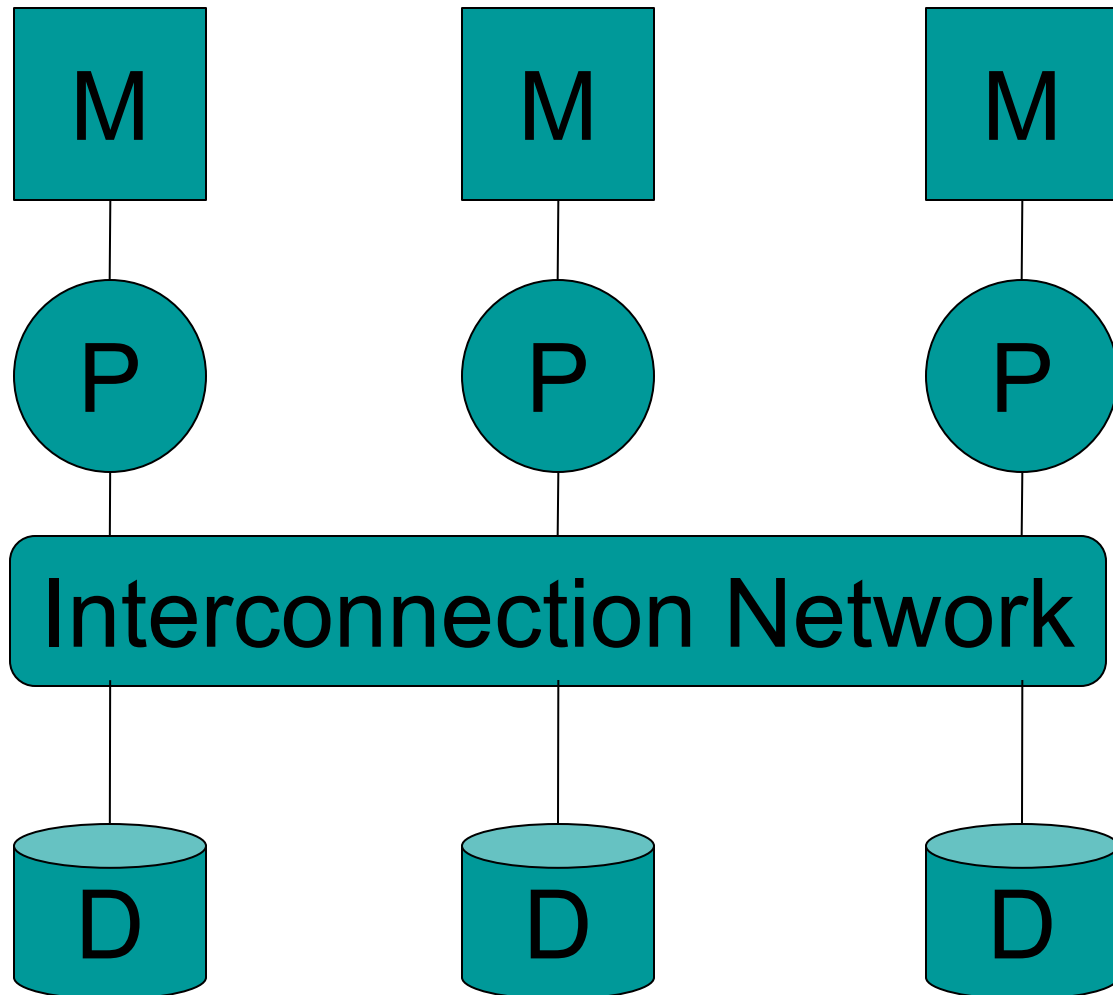
Physical Architectures for Parallel Databases

- Shared memory
- Shared disk
- Shared nothing

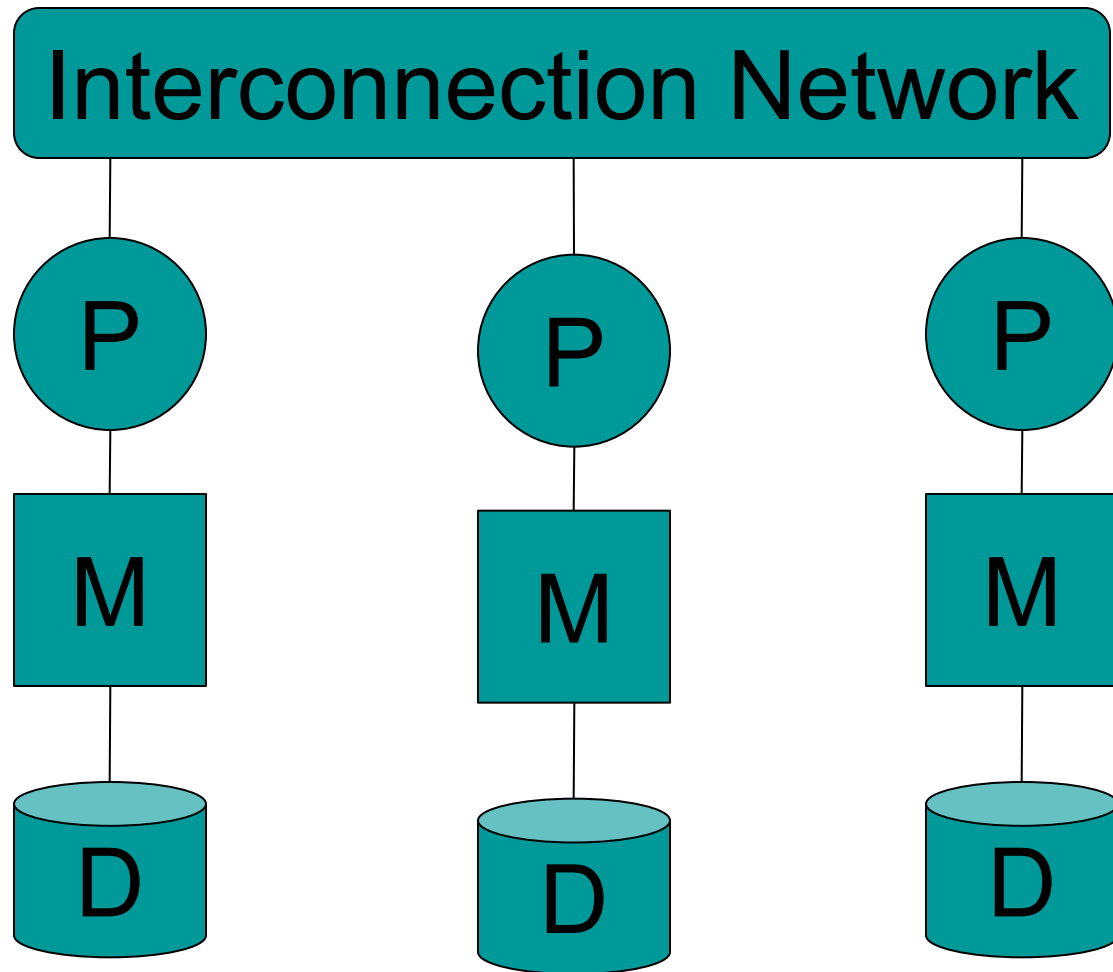
Shared Memory



Shared Disk



Shared Nothing



Shared Nothing

- Most scalable architecture
 - Minimizes interference by minimizing resource sharing
 - Can use commodity hardware
- Difficult to program and manage
- Processor = server = node

Big Data systems often
focus on shared nothing

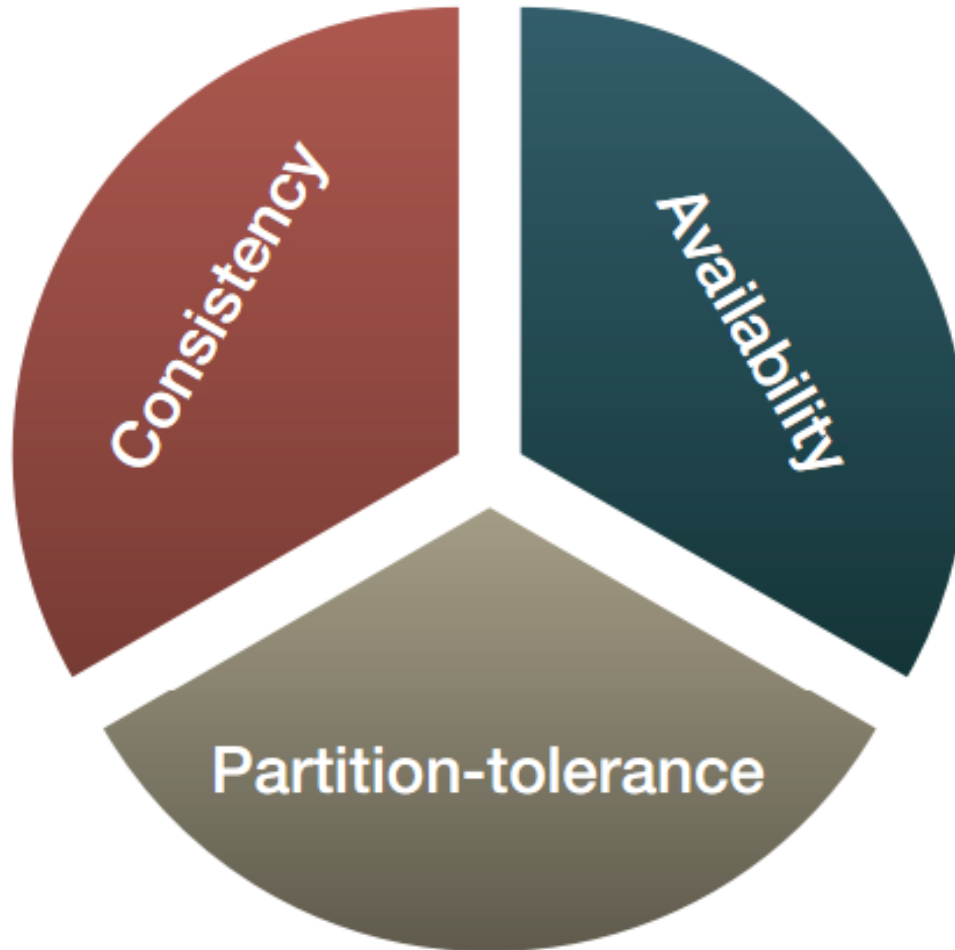
Parallel DBMS Architectures

- **Peer-to-Peer Architecture:** Each node in the network can act as a client or server. All nodes are equal and there is not *single-point-of-failure* (SPOF)
 - Ex. file-sharing networks like BitTorrent
 - Pros. Provide high scalability, and fault tolerance.
 - Cons. hard to implement
- **Master-Slave Architecture:** One (or few) node (master) is responsible for processing read/write requests and managing the database, while the other nodes (the slaves) replicate the data and handle specific operations.
 - Ex. Most Bigdata systems
 - Pros. Easy to implement when you have a single coordinator
 - Cons. Cannot scale indefinitely, SPOF
- **Federated Architecture:** Multiple independent databases are used to provide access to the data
 - Ex. *Linked Open Data*
 - Pros. Each database or system remains autonomous
 - Cons. Big Data (Variety): hard to integrate

Taxonomy for Parallel Query Evaluation

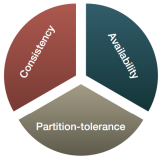
- Inter-query parallelism
 - Each query runs on one processor
- Inter-operator parallelism
 - A query runs on multiple processors
 - An operator runs on one processor
- Intra-operator parallelism
 - An operator runs on multiple processors
 - **Our focus: Most scalable**

From ACID to CAP Properties



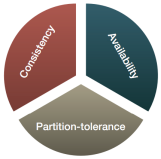
Definition

- **Consistency (Atomic/Linearizable)**
 - All nodes in a distributed system see the same and most up to date data
 - A write is immediately seen by all nodes
 - A read operation receives the most recent value or an error
- **Availability (Machine)**
 - Every request (read/write) receives a valid response
 - The whole system is still operational even if some nodes are down
- **Partitioning (Network)**
 - The system continues to function even if the network is experiencing connectivity issues, or partitioning



Common Techniques Replication

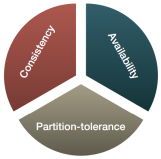
- **Replication** improves *availability* and *partition* tolerance
 - store copies the data on several machines.
- To maintain data **consistency**, replication can be implemented with consistency control
- **Quorums** ensure that updates are propagated to a certain number of nodes before they are considered successful



Common Techniques

Caching

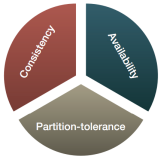
- Store frequently accessed data in memory to reduce the number of disk/network reads
- Caching can be implemented **locally** on each node or server, or it can be implemented in a **distributed** manner across the system.



Common Techniques

Load Balancing

- Distributing **client requests** across different nodes or servers in the system to improve performance and ensure that no single node is **overloaded**.
- Load balancing can be implemented using different algorithms and techniques, such as round-robin, weighted round-robin, or least connections.



Common Techniques

Partitioning (Sharding)

- Divide a database into smaller, more manageable pieces called partitions or shards.
 - Ex. Shard a relation on specific attributes.
- Each *shard* is then distributed across different nodes or servers in the system, which can improve scalability and reduce the impact of performance bottlenecks.

Horizontal Data Partitioning (*Sharding*)

- Typical shared-nothing parallelization technique
- Relation R split into P chunks R_0, \dots, R_{P-1} , stored at the P nodes
- How?
 - Round robin: tuple t_i to chunk $(i \bmod P)$
 - Hash based partitioning on attribute A :
 - Tuple t to chunk: $h(t.A) \bmod P$
 - Range based partitioning on attribute A :
 - Tuple t to chunk i if $v_{i-1} < t.A < v_i$

Parallel Selection

Compute $\sigma_{A=v}(R)$, or $\sigma_{v1 < A < v2}(R)$

- On a conventional database: cost = $B(R)$
- Q: What is the cost on a parallel database with P servers?
 - Round robin
 - Hash partitioned (on attribute A)
 - Range partitioned (on attribute A)

Parallel Selection

- Answer:
 - Round robin: $B(R)$; all servers do the work in parallel
 - Hash: $B(R)/P$ for $\sigma_{A=v}(R)$; one server works only
 $B(R)$ for $\sigma_{v1 < A < v2}(R)$; all servers work in parallel
 - » This is worst case, it can be smaller if A is discrete and we can enumerate it between $v1$ and $v2$
 - Range: (assuming relatively small range)
 $B(R)/P$; one server works only

Data Partitioning

What are the pros and cons ?

- Round robin
 - Good load balance but always needs to read all the data
- Hash based partitioning
 - Good load balance but works only for equality predicates and full scans
- Range based partitioning
 - Works well for range predicates but can suffer from data skew

Parallel Group By

- Compute $\gamma_{A, \text{sum}(B)}(R)$
- Step 1: servers i partitions chunk R_i using a hash function $h(t.A) \bmod P$: $R_{i0}, R_{i1}, \dots, R_{i,P-1}$
- Step 2: server i sends partition R_{ij} to server j
- Step 3: server j computes $\gamma_{A, \text{sum}(B)}$ on $R_{0j}, R_{1j}, \dots, R_{P-1,j}$

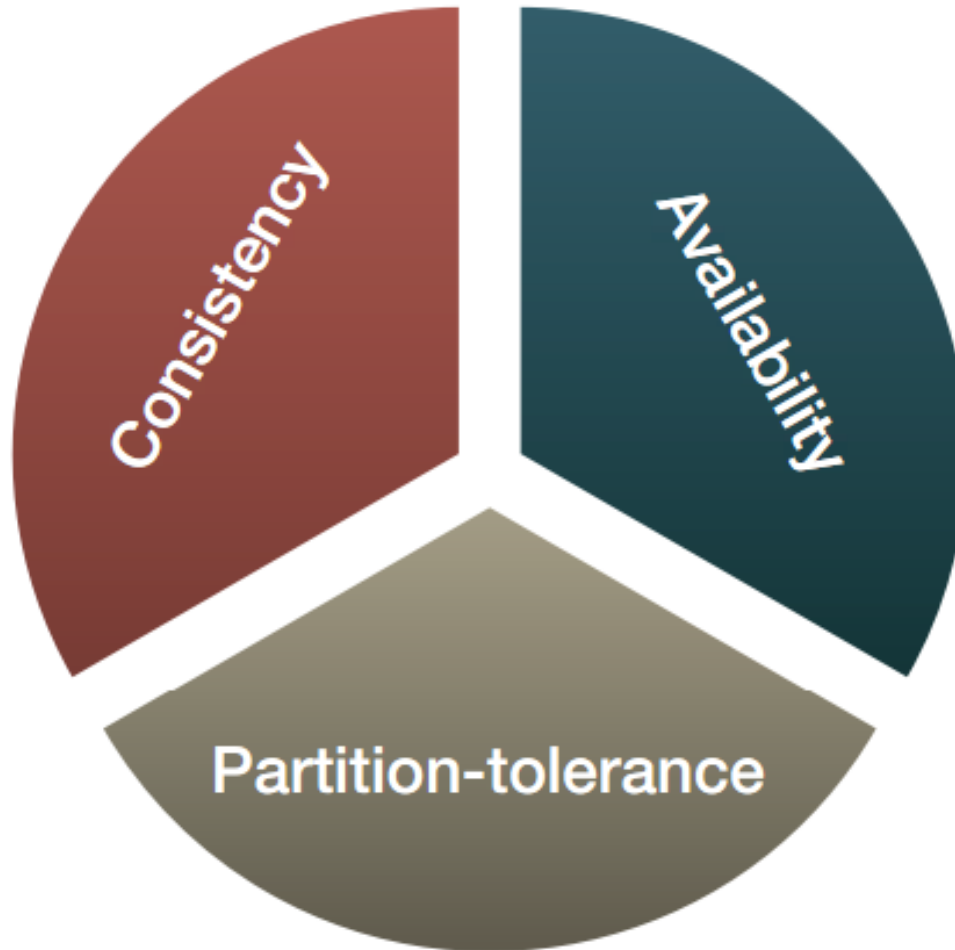
Parallel Join

- Simplest implementation:
 - Join R and S on Attribute A
- Step 1
 - For all servers in $[0, k]$, server i partitions chunk R_i using a hash function $h(t.A) \bmod P$: $R_{i0}, R_{i1}, \dots, R_{i,P-1}$
 - For all servers in $[k+1, P-1]$, server j partitions chunk S_j using a hash function $h(t.A) \bmod P$: $S_{j0}, S_{j1}, \dots, S_{j,P-1}$
- Step 2:
 - Server i sends partition R_{iu} to server u
 - Server j sends partition S_{ju} to server u
- Steps 3: Server u computes the join of R_{iu} with S_{ju}

Outline

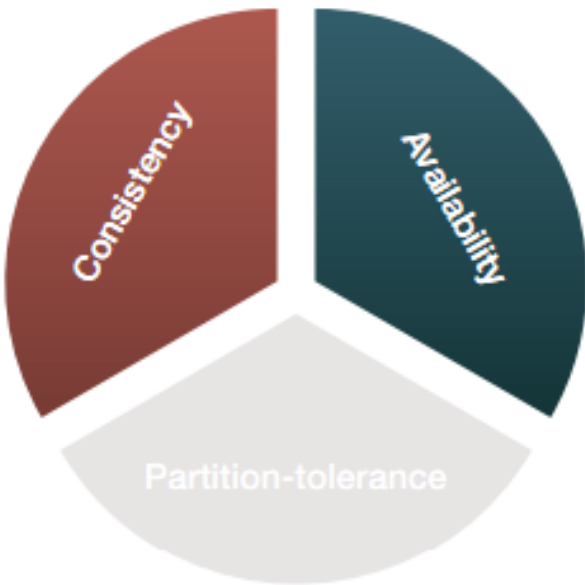
1. Transactions
2. Column Stores
3. Parallel DBMSs
4. CAP

CAP Properties

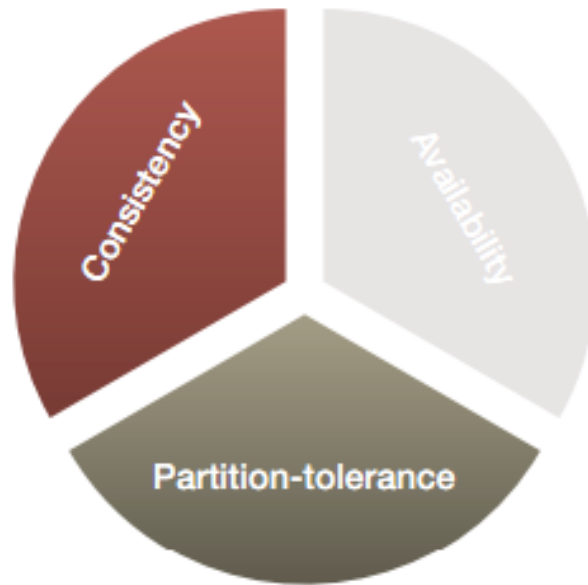


CAP Conjecture

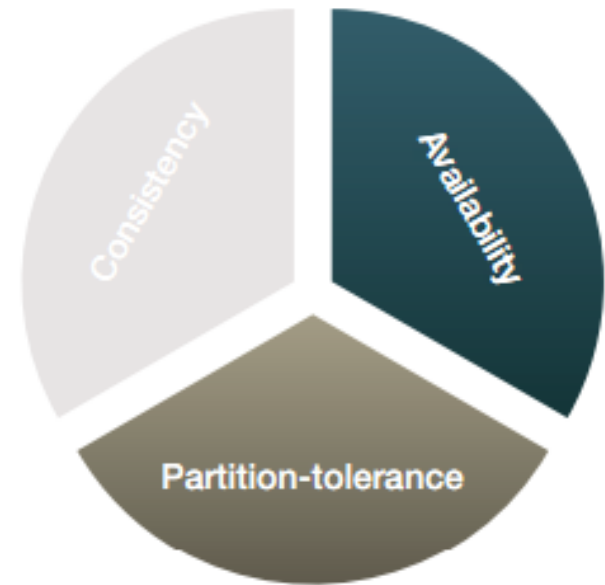
- It is **impossible** for a distributed computer system to simultaneously provide all three guarantees
 - Pick **at most two** properties



CA



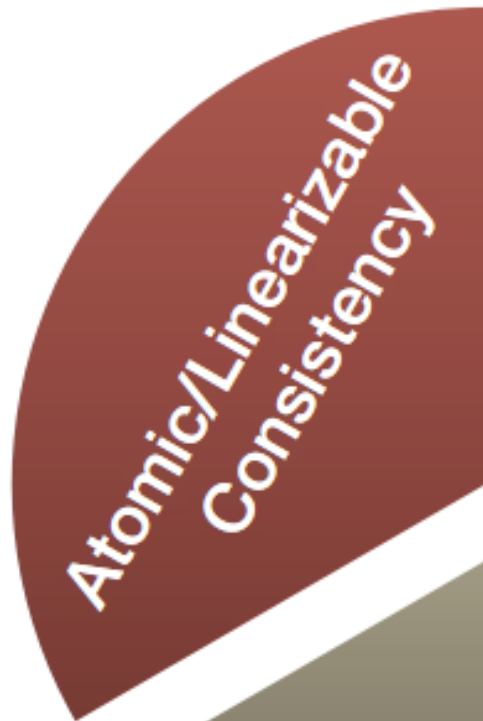
CP



AP

Formal Definitions

\exists total order
 \forall operations
so that they look
as if they were
completed at a
single instant



every request
received by a
non-failing
node must result
in a response



Partition-tolerance

no set of failures
less than total network
failure is allowed to
cause the system to
respond incorrectly

Theorem 1

- It is **impossible** in the asynchronous network model to implement a read/write data object that guarantees the following properties:
 - Availability
 - Atomic consistency

in all fair executions (including those in which messages are lost).

(Asynchronous, i.e., there is no clock, nodes make decisions based only on the messages received and local computation).

Theorem 2

- Similar to Theorem 1, but for partially synchronous networks.

(Partially synchronous, i.e., every node has a clock, and all clocks increase at the same rate. However, they are not synchronized.)

Sketch of Proof (1)

- For partially synchronous networks
- By contradiction:
 - Assume an algorithm A exists that meets the three criteria
 - Assume that the network consists of at least two disjoint, non-empty sets: $\{G1, G2\}$
 - Assume that both sets share a copy of variable $v0$
 - Assume that all messages between $G1$ and $G2$ are lost

Sketch of Proof (2)

- Construct an execution α by superimposing two executions α_1 and α_2 :
- Execution α_1 :
 - Write v_0 request to G_1
 - Acknowledgment by G_1
- Execution α_2 :
 - Wait (for at least the duration of α_1)
 - Read v_0 request to G_2

=> The read request returns the initial value, rather than the new value written by the write request, **violating** atomic consistency!

CAP Tradeoffs

- While it is impossible to provide all three properties, any two of those three properties can be achieved!
- **Trivial** examples for the asynchronous model:
 - CA: centralized RDBMS
 - CP: ignore all incoming requests
 - AP: always return initial value

Weak Consistency Models

- Following the CAP theorem, systems started to appear providing **weaker consistency** models
 - failures are unavoidable in large scale systems
 - availability is a must for many services
 - compromise on consistency!
 - Example: t-Connected Consistency (Lynch)
 - in the presence of no partitions, the system is consistent
 - in the presence of partitions, stale data can be returned
 - once a partition heals, there is a time limit on how long it takes for consistency to return
- ⇒ “Eventual” consistency in many NOSQL systems (e.g., Cassandra)

Visual Guide to Recent Systems

CA:
RDBMSs
(MySQL,
Postgres,
Oracle etc.)
Greenplum,
Vertica

A

AP:
Amazon Dynamo,
Cassandra,
SimpleDB,
CouchDB

C

P

CP: Google BigTable, Hbase,
Berkeley DB, MemcachDB, MongoDB

Recap on CAP

- Fundamentally changed CS for the past decade
 - Distributed RDBMSs are, strictly speaking, **impossible** to deploy on today's infrastructures!
- Still in the flux today
 - Weaker consistency models (*eventual* consistency)
 - We can influence whether faults impact **yield (availability)**, **harvest (consistency)**, or both

$$yield = \frac{queries_completed}{queries_offered}$$

$$harvest = \frac{data_returned}{complete_data}$$

- **PACELC** [Abadi12] .. Extends CAP and builds on eventual consistency:
 - If Partition, trade Availability for Consistency;
Else, trade Latency for Consistency