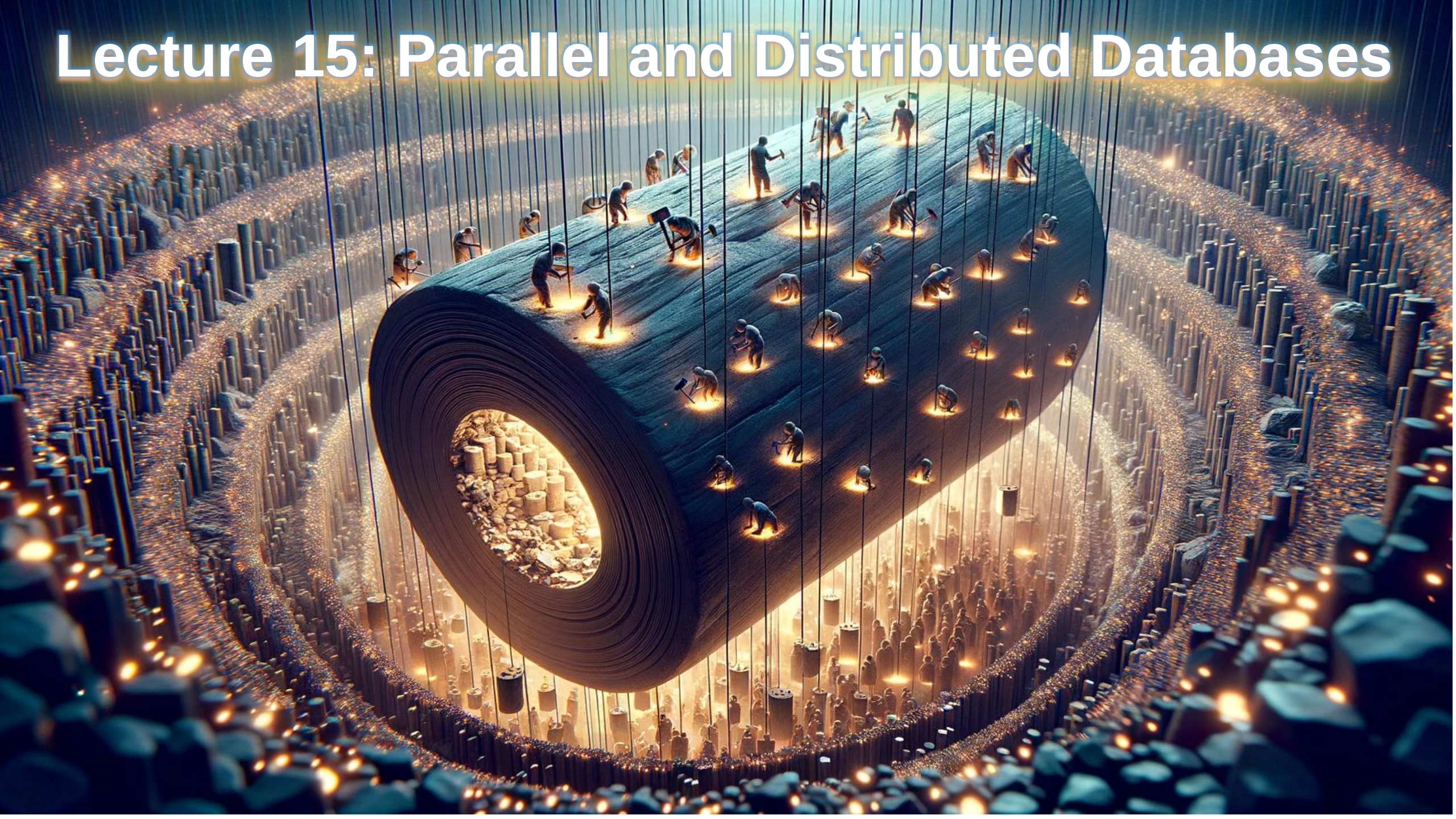
Lecture 15: Parallel and Distributed Databases



RECAP: OCC

- Divide transaction execution in 3 phases
  - **Read**: transaction executes on DB, stores local state
  - **Validate**: transaction checks if it can commit
  - **Write**: transaction writes state to DB

RECAP: Validation Rules

When Tj completes its read phase, require that for all Ti < Tj, one of the following conditions must be true for validation to succeed (Tj to commit):

1) Ti completes its write phase before Tj starts its read phase
2) W(Ti) does not intersect R(Tj), and Ti completes its write phase before Tj starts its write phase.
3) W(Ti) does not intersect R(Tj) or W(Tj), and Ti completes its read phase before Tj completes its read phase.
4) W(Ti) does not intersect R(Tj) or W(Tj), and W(Tj) does not intersect R(Ti) [no conflicts]

These rules will ensure serializability, with Tj being ordered after Ti with respect to conflicts

Recap: OCC Validation

Restating previous rules, aborts required if:

1) $W(Ti) \cap R(Tj) \neq \{\}$, and Ti does not finish writing before Tj starts, Tj must abort, because Tj may have only seen some of what Ti wrote

or

2) $W(Ti) \cap (W(Tj) \cup R(Tj)) \neq \{\}$, and Tj overlaps with Ti validation or write phase, Tj must abort because it needs its writes to all appear after Ti's writes

https://clicker.mit.edu/6.8530/

In OCC when do you assign the Transaction Identifier?

a) At the beginning of the trx,
b) At the start of the validation phase
c) At the start of the write phase
d) At the end of the write phase

https://clicker.mit.edu/6.5830/

- Goal: assign transaction ids T1, … Tn, such that this is the serial equivalent order
- When should we assign transaction identifiers?
- At start of read phase?
  - No! Would be “pessimistic” – don’t want to pre-assign the transaction order before transactions finish running
  - Long running transactions would have to commit before later short transactions
- Assign at end of read phase, just before validation starts

Recap: Snapshot Isolation

- When a TA starts it receives a timestamp, T.
- All reads are carried out *as of* the DB version of T.
  - Need to keep historic versions of all objects!!!
- All writes are carried out in a separate buffer.
  - Writes only become visible after a commit.
- When TA commits, DBMS checks for conflicts
  - Abort TA1 with timestamp T1 if exists TA2 such that
    - TA2 committed after T1 and before TA1
    - TA1 and TA2 updated the same object

https://clicker.mit.edu/6.8530/

Assume a system that manages holidays for hospital staff.
The system must ensure that always one chief doctor is on duty and uses the following transaction for it:

```
BEGIN TRANSACTION;
   IF EXISTS(SELECT * FROM staff
       WHERE title='chief'
          AND vacation = false
          and name <> @user) THEN
       UPDATE staff SET vacation = true
          WHERE name = @user
   END IF;
COMMIT TRANSACTION;
```

However, the system removes the vacation status by simply executing:

```
UPDATE staff SET vacation = false
          WHERE name = @user
```

What consistency guarantees are needed to ensure that the invariant is always guaranteed (select all that apply):

(A) READ UNCOMMITTED
(B) READ COMMITTED
(C) REPEATABLE READ
(D) SNAPSHOT ISOLATION
(E) SERIALIZABILITY

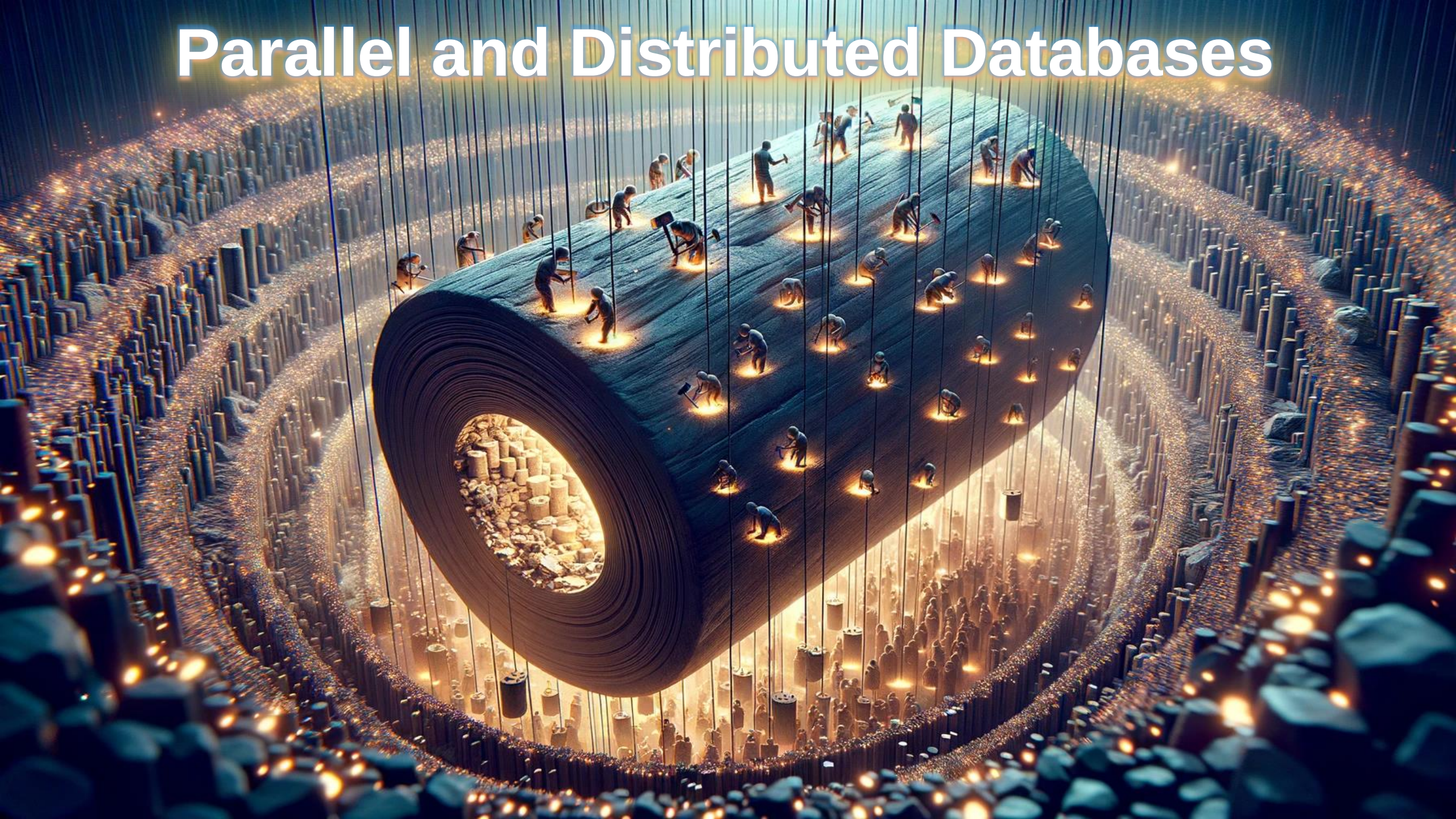
You can assume that there are no changes for staff members

- Observation: Snapshot isolation does not prevent Write-Read conflicts, since it doesn't check whether it read something another transaction wrote
- This leads to so-called write-skew, i.e.:

T1	T2
RX	
	RY
WY	
	WX

- Neither transaction saw the other's write; this would not be permitted under serializability

Parallel and Distributed Databases



Parallel & Distributed DBs Overview

- Parallel DBs: how to get multiple processors/machines to execute different parts of a SQL query
  - Especially relevant for big, slow running queries

Today!

- Distributed DBs: what happens when these machines are physically disjoint / fail independently
  - Especially relevant for transaction processing

Parallel DB Goal

- SQL, but faster by running on multiple processors

- What do we mean by faster?

$$speed\ up = \frac{old\ time}{new\ time} \text{ on same problem, with N times more hardware}$$

$$scale\ up = \frac{1x\ larger\ problem\ on\ 1x\ hardware}{Nx\ larger\ probelm\ on\ Nx\ hardware}$$

- Not necessarily the same: smaller problem may be harder to parallelize

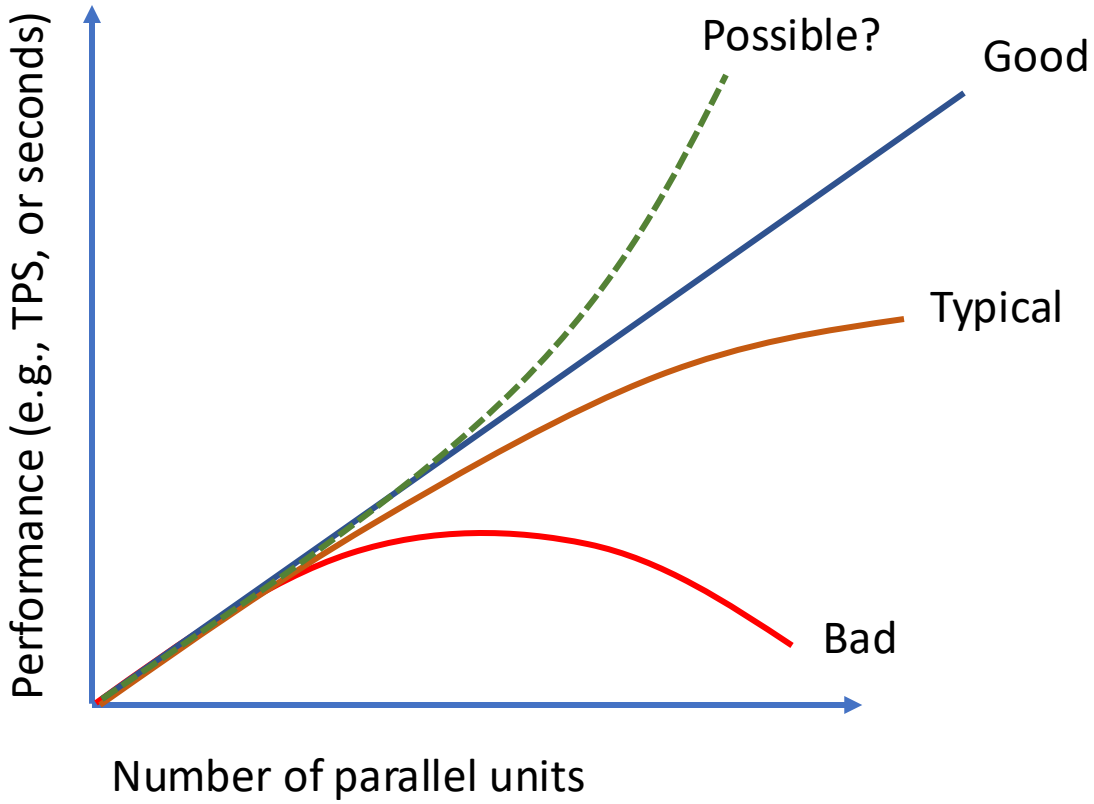
DB Specific Metrics

- **Transaction speedup**: fixed set of txns, with 1 vs N machines
- **Query speedup**: fixed sized DB, with 1 vs N machines

- **Transaction scaleup**: N times as many txns for N machines
- **Query scaleup** : N times as big a query for N machines

Speedup Goal

- Linear?



Barriers to Linear Scaling

What are some barriers in analytics and transactional workloads?

- Startup times
  - e.g., may take time to launch each parallel executor
- Interference
  - processors depend on some shared resource
  - E.g., input or output queue, or other data item
- Skew
  - workload not of equal size on each processor

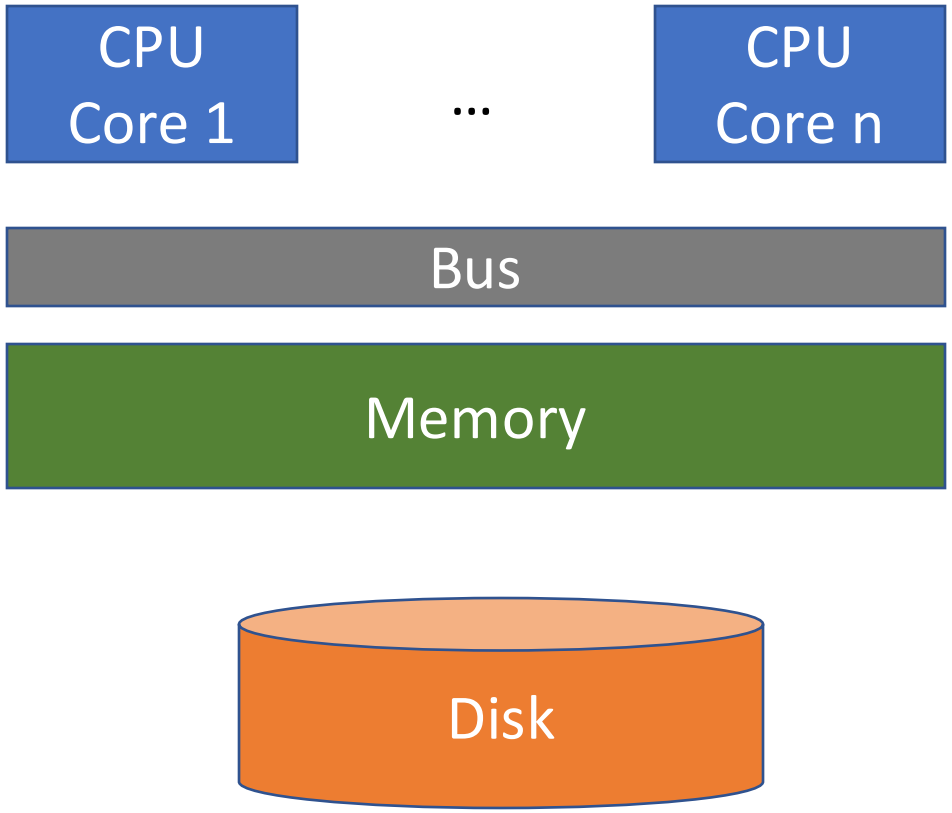
Properties of Parallelizable Workloads

- Provide linear speedup
- Usually can be decomposed into small units that can be executed independently
  - "embarrassingly parallel"
- As we will see, relational model generally provides this

Parallel Architectures

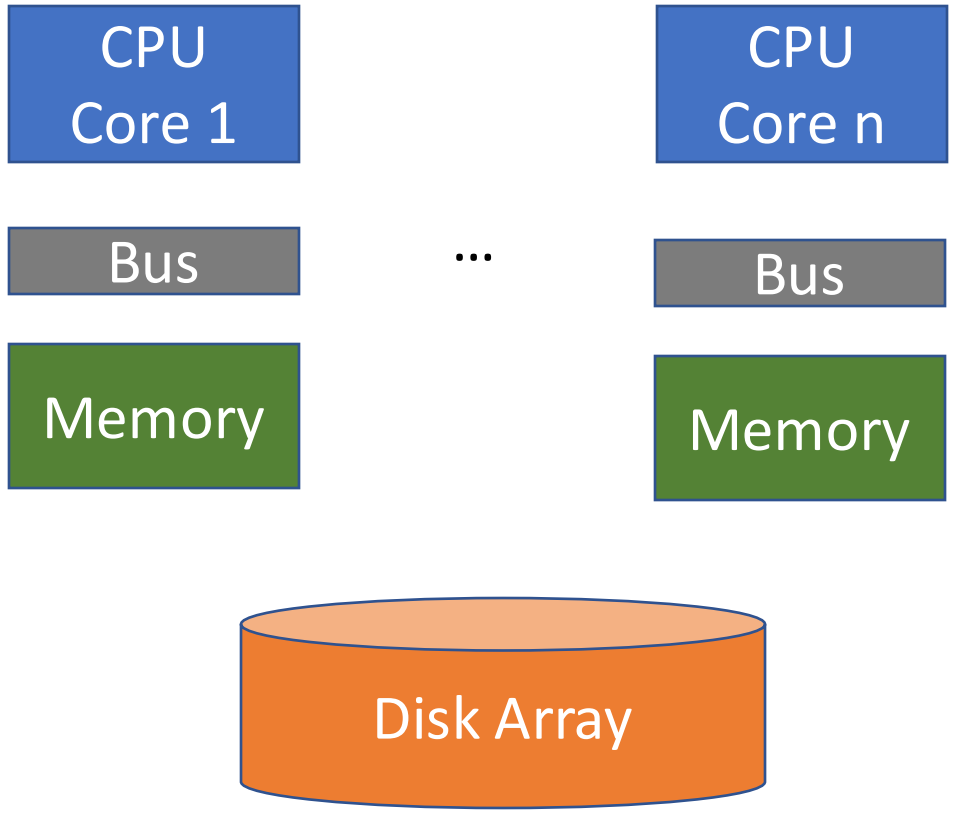
- Several different ways we might parallelize databases
- Multiple cores?
- Multiple machines?

Types of Parallelism – Shared Everything



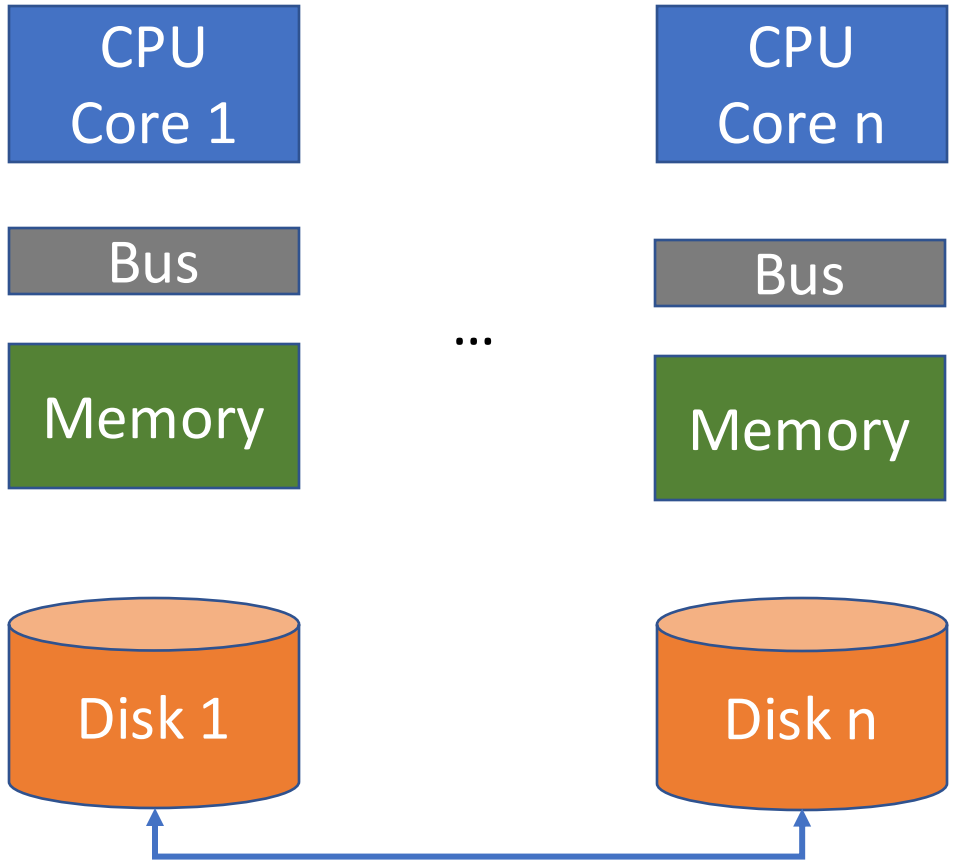
- Conventional multicore computer
- Multiple threads for execution
- Each core can access any record
- Difficult to scale beyond a few cores
- Not fault tolerant

Types of Parallelism – Shared Disk



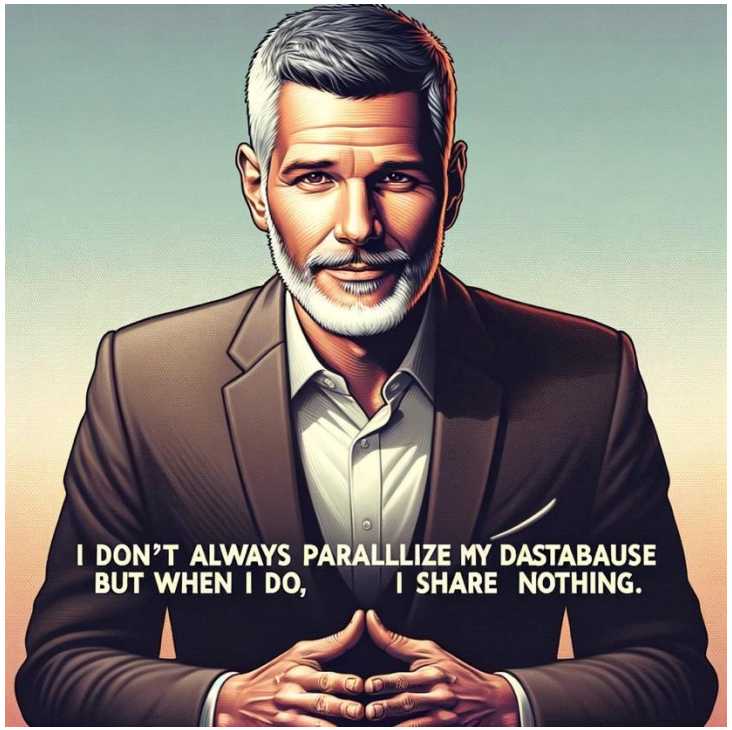
- Several machines
- Each can access any record on disk
- Requires coordination to ensure writes to disk are done safely
- Relies on reliable disk array for fault tolerance
- Popularized by Oracle; reborn in the cloud era (more in a bit)

Types of Parallelism – Shared Nothing

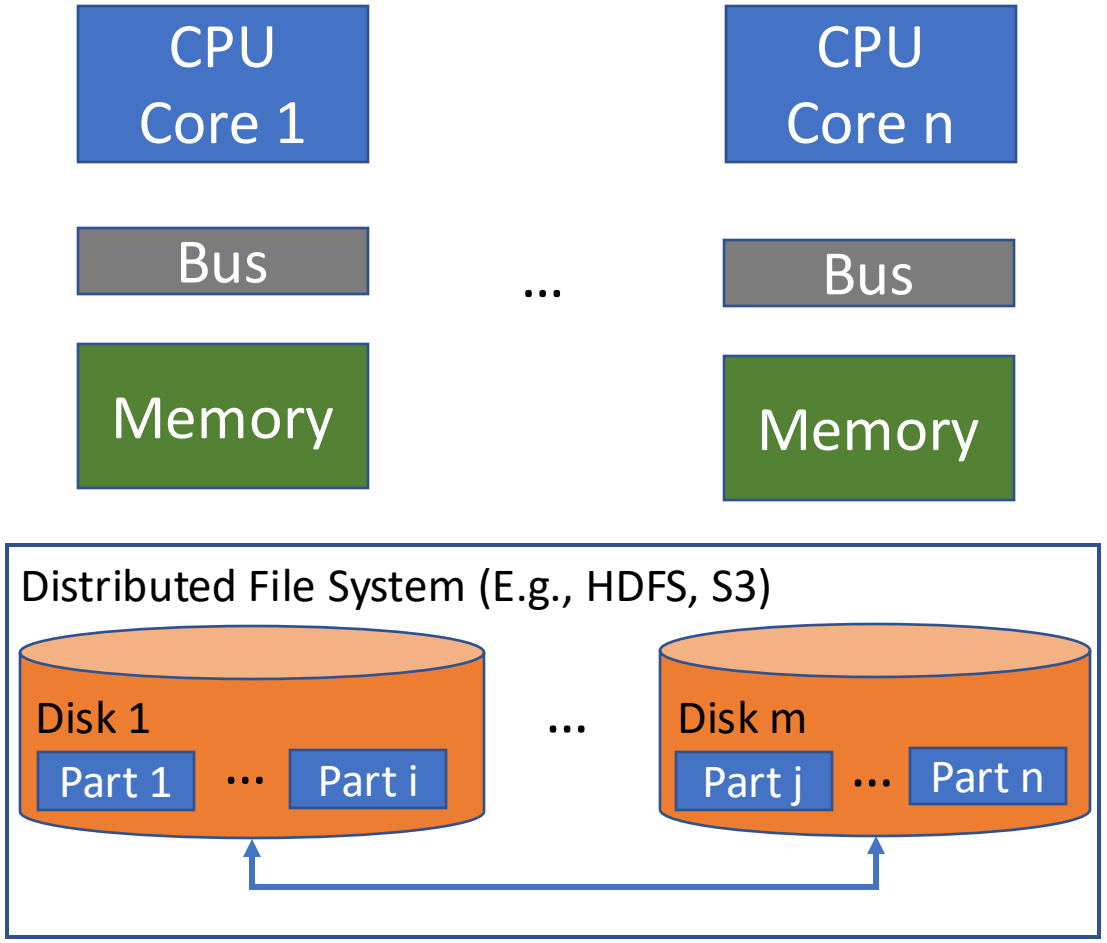


High speed interconnect (e.g., 10GB Ethernet, Infiniband, …)

- Several machines
- Data partitioned across machines
  - Each machine responsible for processing & modifying its data
- Scales very well
  - Easy to add new machines & partitions
- Fault tolerance via replication



Types of Parallelism – Shared Nothing on Distributed File System



- Decouples scaling of storage from scaling of processing
- Storage layer implements its own fault tolerance
- Logically data is still partitioned and operated on by different processors
- Has become common with rise of cloud computing
  - E.g., SnowFlake, MapReduce, …

Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, Tim Kraska: **Building a Database on S3**, SIGMOD'08

Tradeoffs Between Parallel Architectures

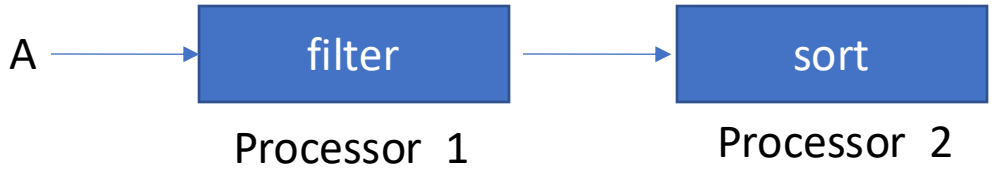
	Pros	Cons
Shared Memory	Name pros and cons	
Shared Disk		
Shared Nothing (partitioned data)		

Tradeoffs Between Parallel Architectures

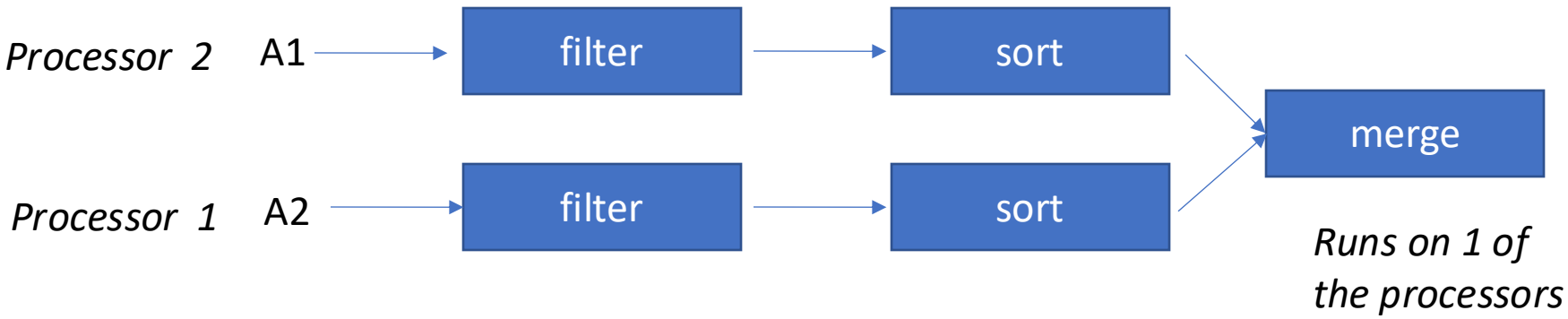
	Pros	Cons
Shared Memory	Easy to build	Performance / scalability
	No changes to concurrency control / recovery	Poor fault tolerance
Shared Disk	Better scalability	Complex cache coherency
	Better fault tolerance	Poor scalability
		Relies on expensive disk array
Shared Nothing (partitioned data)	Cost	New concurrency control/recovery
	Scalability	New executor
	Fault tolerance	

Parallel Query Processing

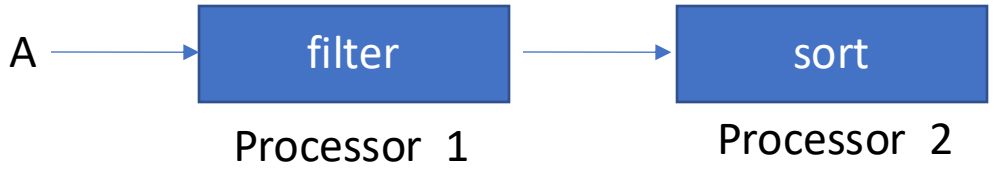
- Three main ways to parallelize
  1. Run multiple queries, each on a different thread
  2. Run operators in different threads ("pipeline")



  3. Partition data, process each partition in a different processor



Pipelined Parallelism



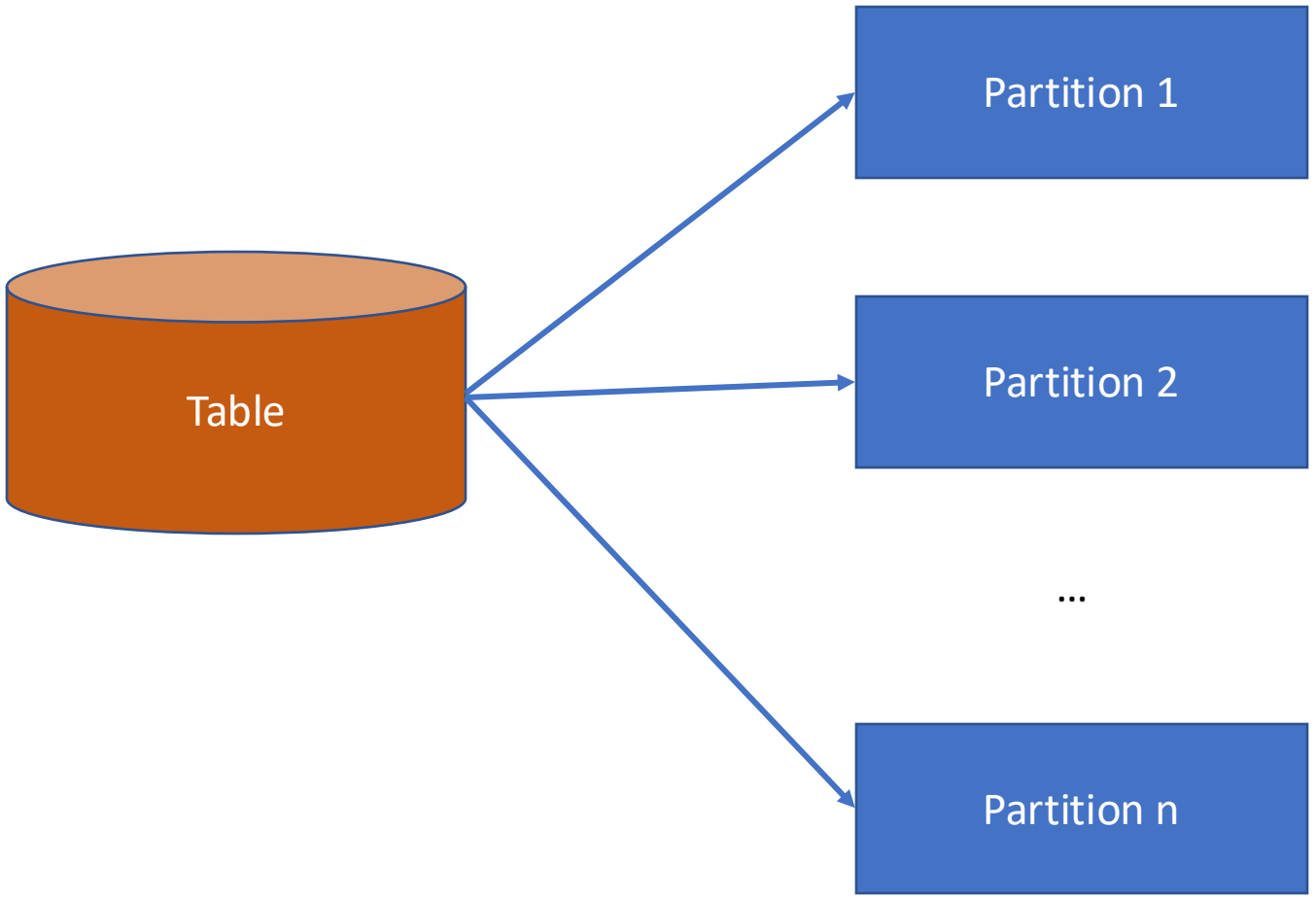
- Only works when each pipeline stage is about the same speed
- Limited parallelism as most pipelines are short
- Inputs to stage i+1 depend on stage i
- If stage i blocks (i.e., sorts), breaks pipeline

- As a result, partitioned parallelism is the primary way database systems scale

Partitioning Strategies

- **Random / Round Robin**
  - Evenly distributes data (no skew)
  - Requires us to repartition for joins
- **Range partitioning**
  - Allows us to perform joins without repartitioning, when tables are partitioned on join attributes
  - Subject to skew
- **Hash partitioning**
  - Allows us to perform joins without repartitioning, when tables are partitioned on join attributes
  - Only subject to skew when there are many duplicate values

Round Robin Partitioning



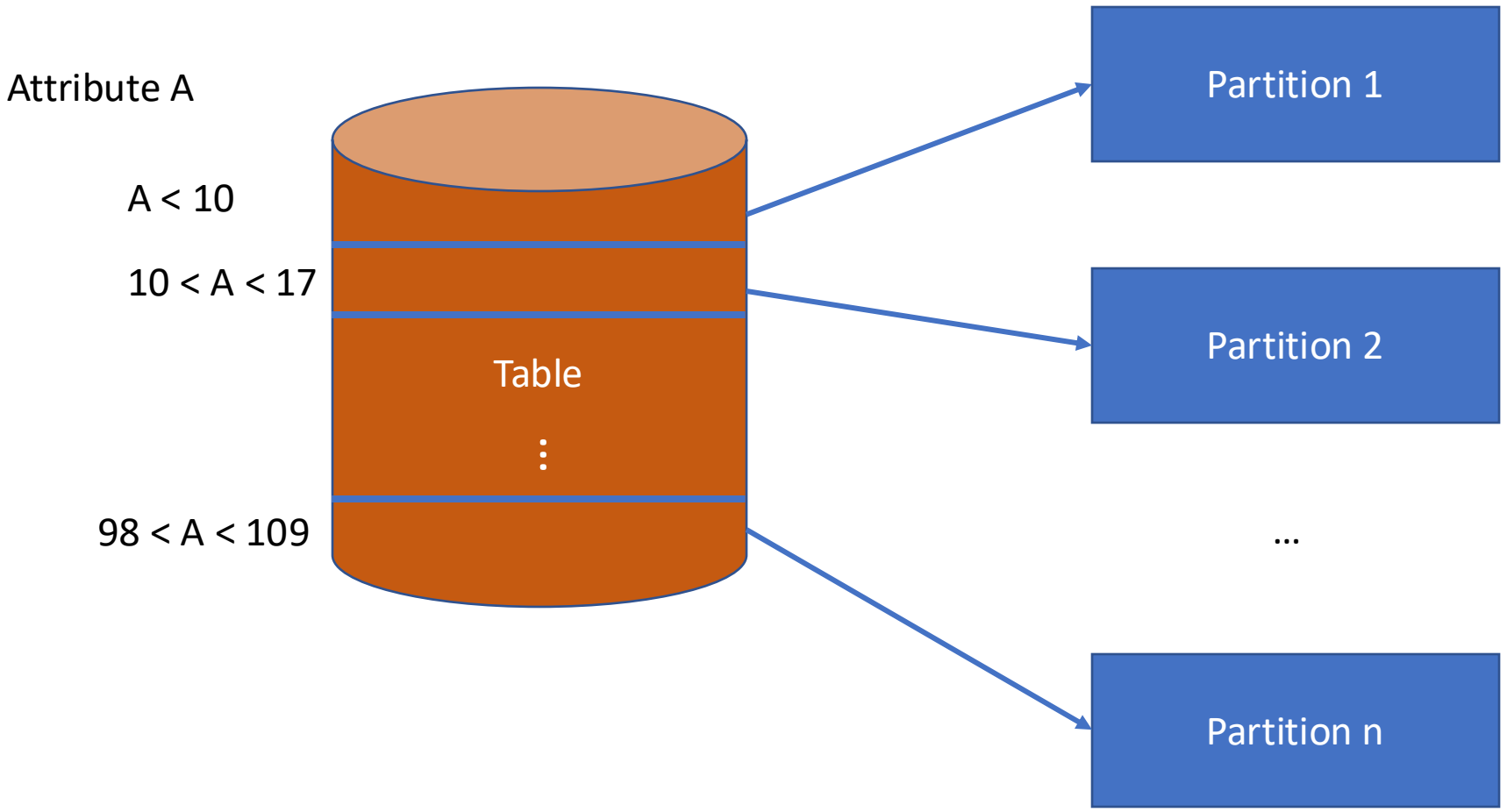
Advantages:

Each partition has the same number of records

Disadvantage:

No ability to push down predicates to filter out some partitions

Range Partitioning



Advantages:

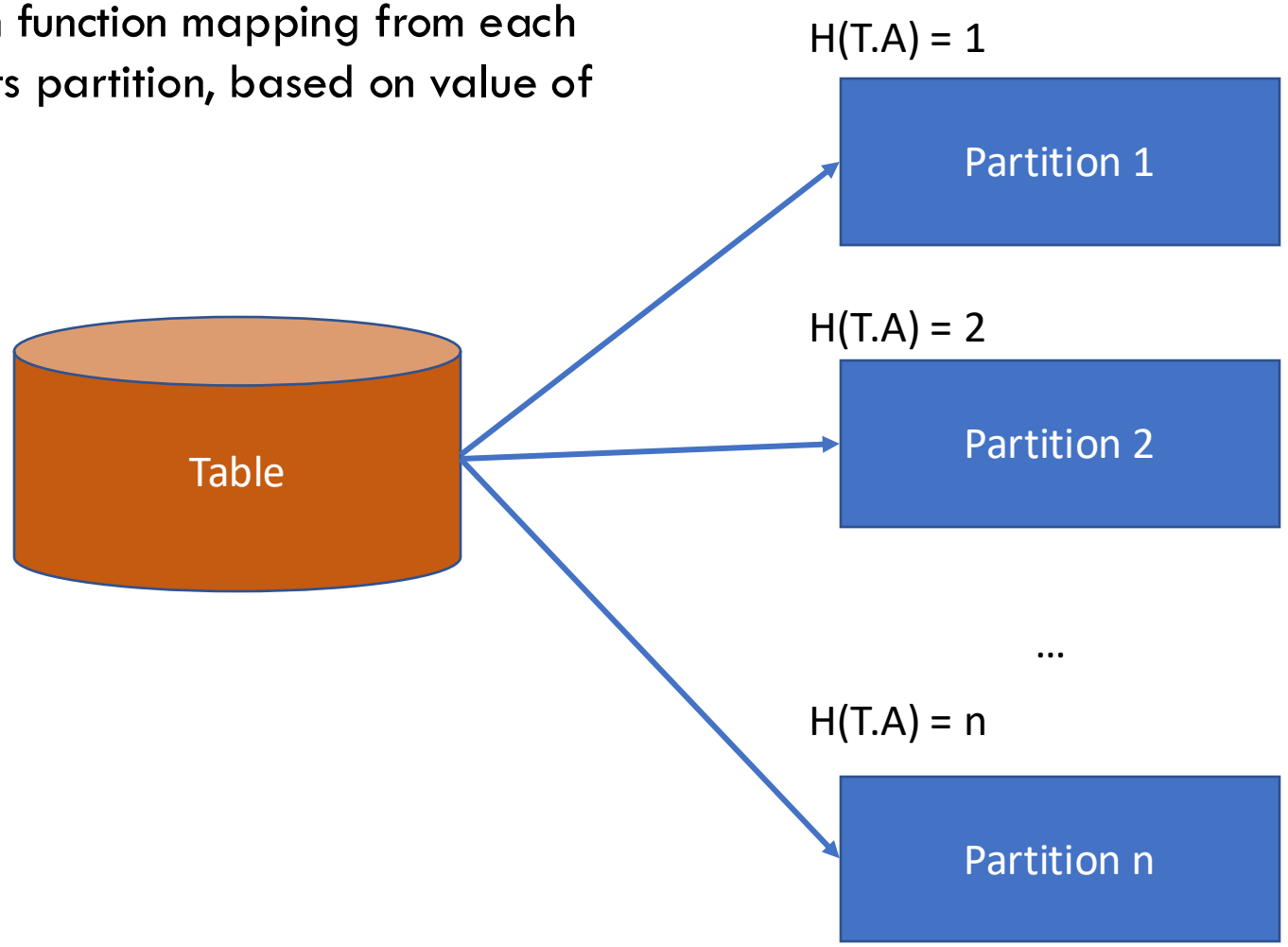
Easy to push down predicates (on partitioning attribute)

Disadvantage:

Difficult to ensure equal sized partitions, particularly in the face of inserts and skewed data

Hash Partitioning

H(T.A) is a hash function mapping from each record in T to its partition, based on value of attribute A.



Advantages:

Each partition has about the same number of records, unless one value is very frequent

Possible to push down equality predicates on partitioning attribute

Disadvantages:

Can't push down range predicates

Parallel Operations in a Partitioned DB

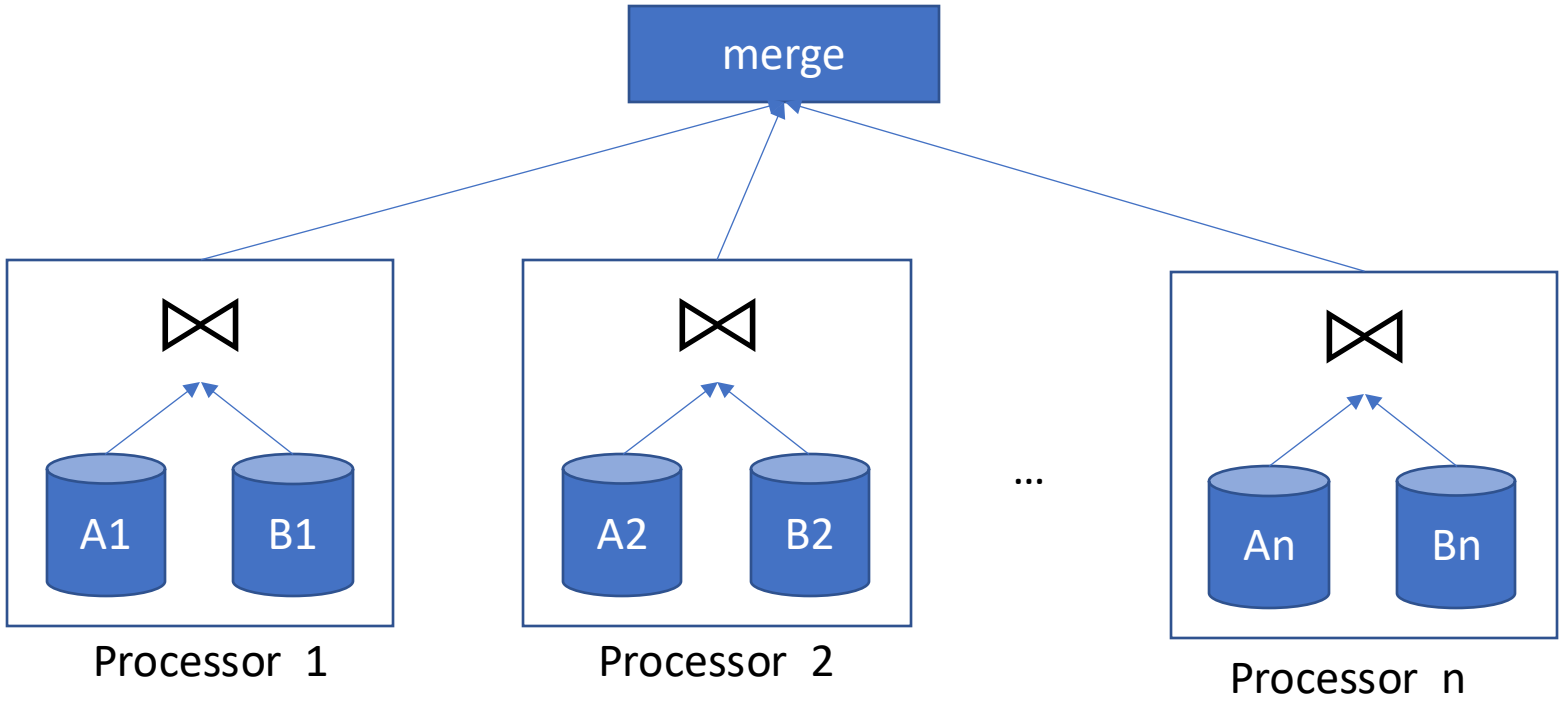
- **SELECT**
  - Trivial to “push down” to each worker
  - Depending on partitioning attribute, may be able to skip some partitions
- **PROJECT**
  - Assuming all columns are on each node, nothing to be done
- **JOIN**
  - Depending on data partitioning, may be able to process partitions individually and then merge, or may need to repartition
- **AGGREGATE**
  - Partially aggregate data at each node, merge final result

Join Strategies

- If tables are partitioned on same attribute, just run local joins
  - Also, if one table is replicated, no need to join
- Otherwise, several options:
  1. Collect all tables at one node
     - Inferior except in extreme cases, i.e., very small tables
  2. Re-partition one or both tables – "shuffle join"
     - Depending on initial partitioning
  3. Replicate (smaller) table on all nodes

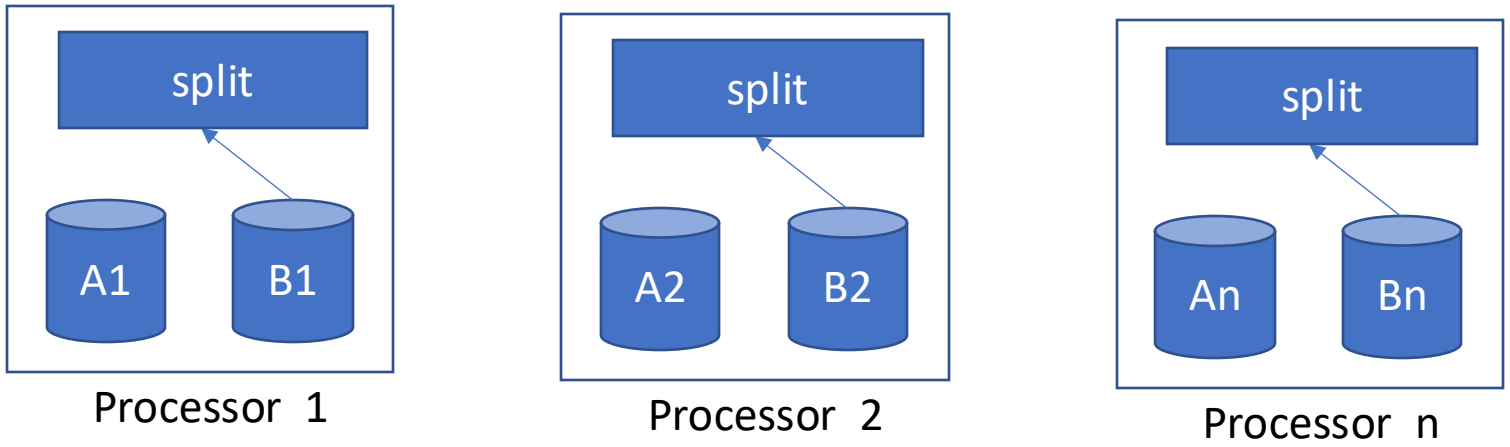
Table Pre-Partitioned on Join Attribute

- Suppose we have hashed A on a, using hash function F to get F(A.a) → 1..n (n = # machines)
- Also hash B on b using *same* F
- Query: SELECT * FROM A,B WHERE A.a = B.b



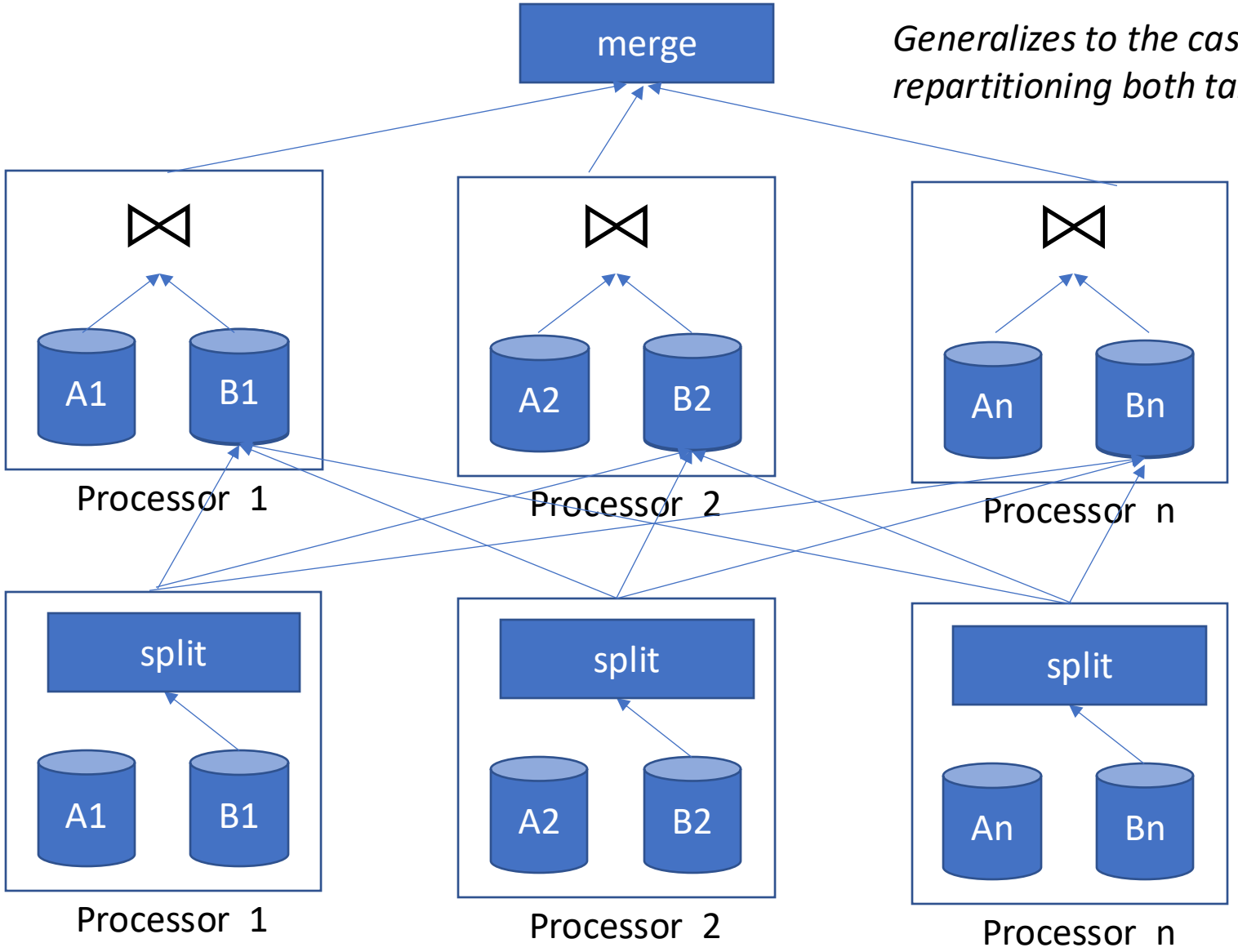
Repartitioning Example – "Shuffle Join"

- Suppose A pre-partitioned on a, but B needs to be repartitioned



Repartitioning Example

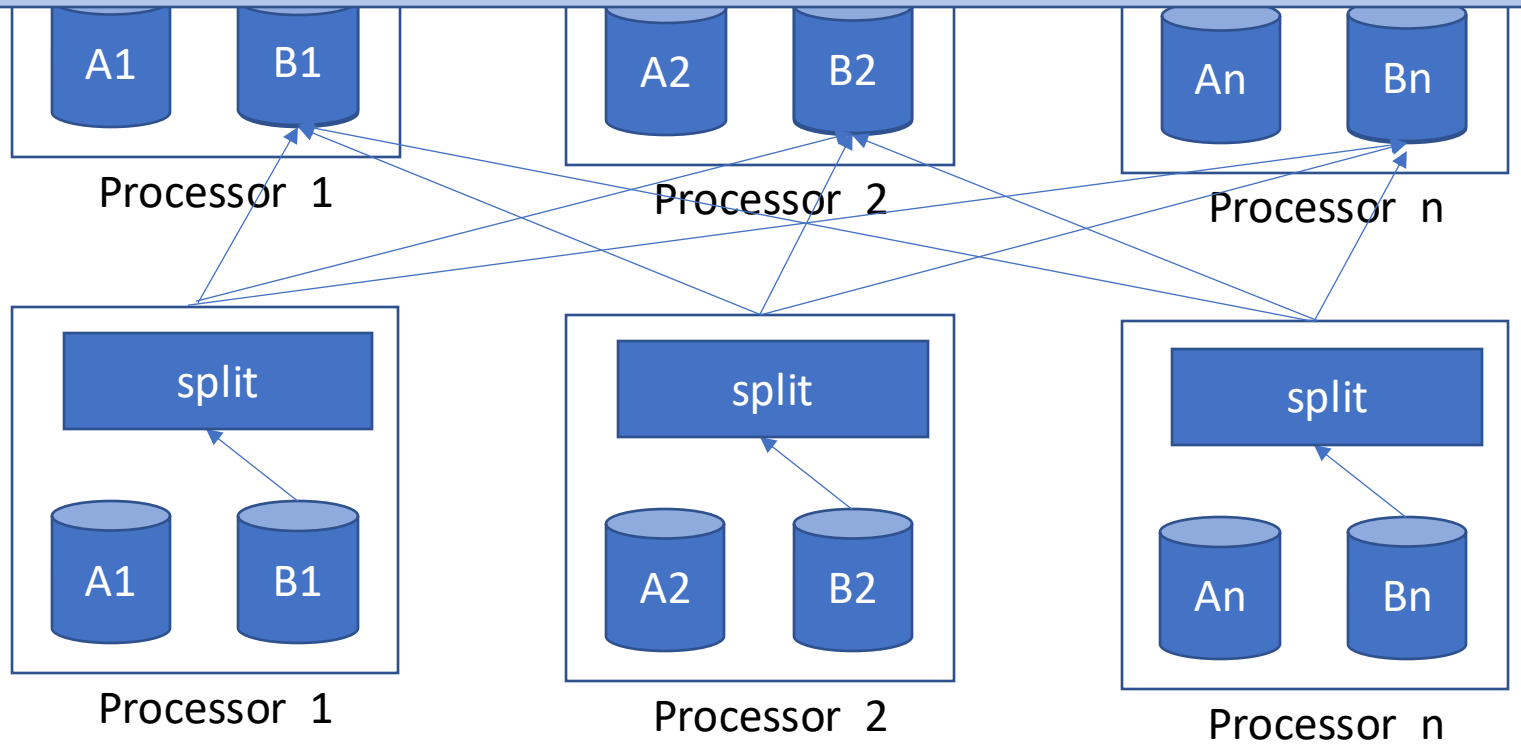
- Suppose A pre-partitioned on a, but B needs to be repartitioned



https://clicker.mit.edu/6.8530/

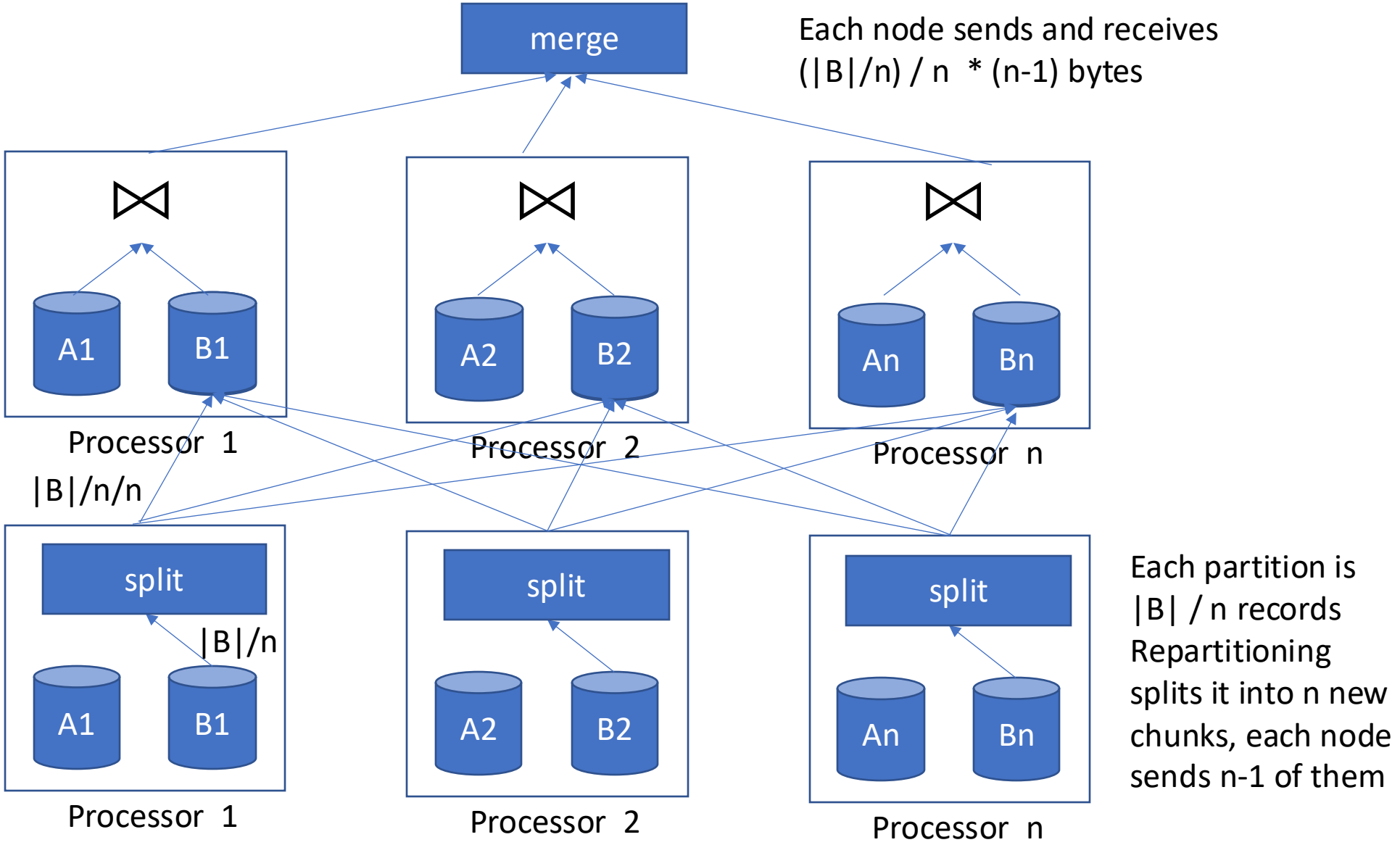
How many bytes are sent from each machine?

A: (|B|/n) bytes
B: (|B|/n) / n * (n-1) bytes
C: 2* (A) (|B|/n) / n * (n-1) bytes



Repartitioning Example

- Suppose A pre-partitioned on a, but B needs to be repartitioned



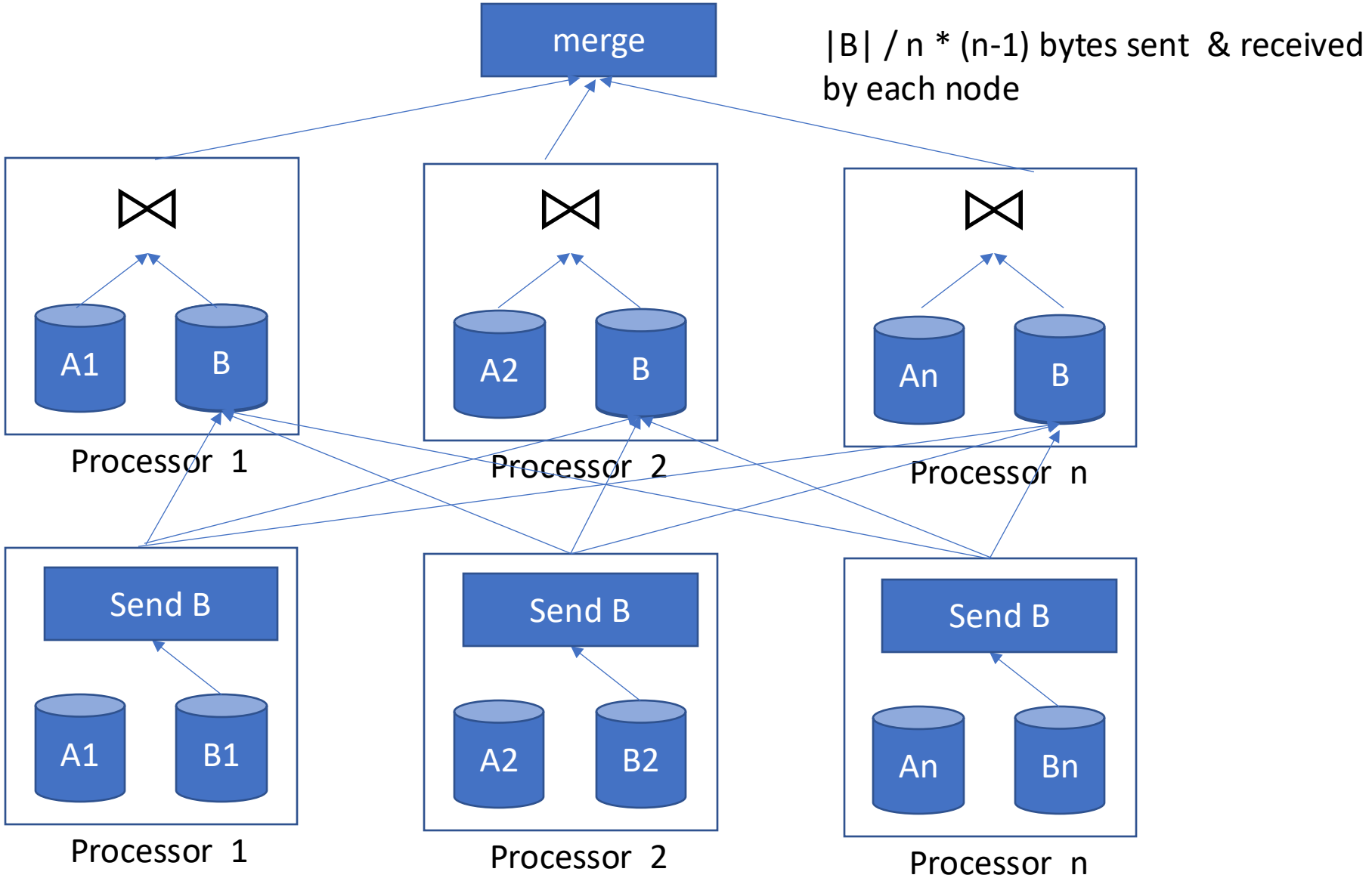
Repartitioning Both Tables

- Suppose both tables, A and B, need to be repartitioned
- Each node sends and receives

$$(|A|/n)/n * (n-1) + (|B|/n)/n * (n-1) \text{ bytes}$$

Replication Example

- Suppose we replicate B to all nodes



Replication vs Repartioning

- Replication requires each node to send smaller table to all other nodes
  - (|T| / n) * (n-1) bytes sent by each node
  - vs ((|T| / n) / n) * (n-1) to repartition one table
- When would replication be preferred over repartitioning for joins?
  - If size of smaller table < data sent to repartition one or both tables
  - Should also account for cost of join: will be higher with replicated table
- Example: |B| = 1 MB, |A|=100 MB, n=3
- Need to repartition A (B distributed on join attr)
  - Data to repartition A is |A|/3 / 3 * 2 = 22.2 MB per node
    - Join .33 MB to 33 MB
  - Data to broadcast B is |B| = 1/3 * 2 = .66 MB
    - Join 1 MB to 33 MB

https://clicker.mit.edu/6.8530/

Replication: (|T| / n) * (n-1) bytes sent by each node
Partitioning: ((|T| / n) / n) * (n-1) to repartition one table

- Suppose we have two tables R and K, partitioned across 3 nodes
- |R| = 9 MB
- |K| = 90 MB
- Join is R.b = K.b
  - K is hash partitioned on b, R is **not** partitioned on b
- How much data does each node send if repartition R vs replicate R:

A) 2 MB vs 6 MB

B) 6 MB vs 2 MB

C) 0.9 MB vs 9 MB

https://clicker.mit.edu/6.8530/

Replication: (|T| / n) * (n-1) bytes sent by each node
Partitioning: ((|T| / n) / n) * (n-1) to repartition one table

- Suppose we have two tables R and K, partitioned across 3 nodes
- |R| = 9 MB
- |K| = 90 MB
- Join is R.b = K.b
  - K is hash partitioned on b, R is **not** partitioned on b
- How much data does each node send if we:
  1. Repartition R — (9 / 3) / 3 * 2 = 2 MB
  2. Replicate R — 9 / 3 * 2 = 6 MB

Additional Options for Joins

- Pre-replicated small tables
  - If space permits, can be a good option

- ”Semi-join”
  - send list of join attribute values in each partition of B to A,
  - then send list of matching tuples from A to B,
  - then compute join at B
- Good for selective joins of wide tables
  - Pre-filters A with join values that actually occur in B, rather than sending all of B

ble

A	B	C	D	E
4	d	e	h	i
3	z	a	f	g

A	B	C	D	E
1	x	y	J	k

T1 P1

1	**x**	**y**
3	z	a

T1 P2

A	B	C
2	b	c
4	**d**	**e**

Node 1

Node 2

T2 P1

A	D	E
3	f	g
4	h	i

T2 P2

A	D	E
1	j	k
5	l	m

<u>Total cost:</u>

Each nodes sends & receives

(|join col| / n) * (n-1)

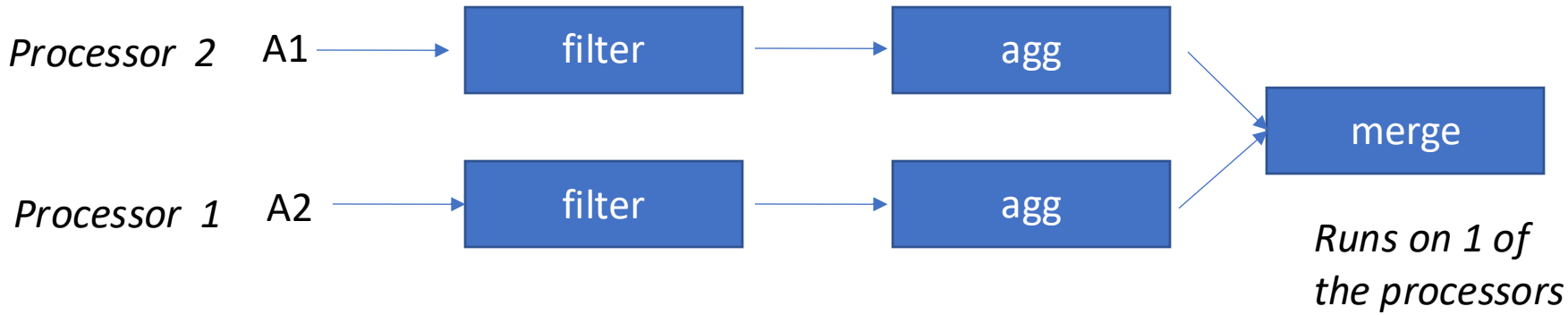
+

(f * |A| / n) * (n-1)

Where f is join selectivity

(Like cost of replication, but only for 1 column +filtered |A|)

Aggregation



In general, each node will have data for the same groups

So merge will need to combine groups, e.g.:

MAX (MAX1, MAX2)
SUM (SUM1, SUM2)

What about average?

Maintain SUMs and COUNTs, combine in merge step

Generalized Parallel Aggregates

- Express aggregates as 3 functions:
  - INIT – create partial aggregate value
  - MERGE – combine 2 partial aggregates
  - FINAL – compute final aggregate

  - E.g., AVG:
    - INIT(tuple) → (SUM=tuple.value, COUNT=1)
    - MERGE (a1, a2) → (SUM=a1.SUM + a2.SUM, COUNT=a1.count+a2.count)
    - FINAL(a) → a.SUM/a.COUNT

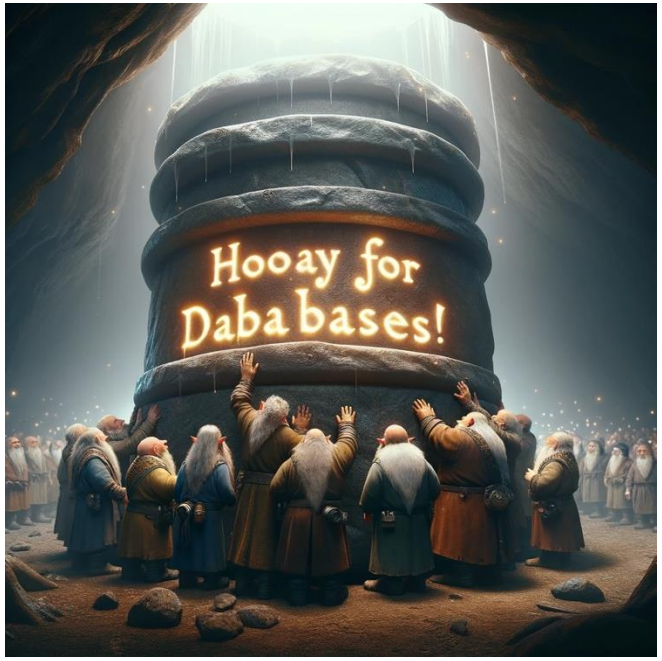
What does MERGE do?

- For aggregate queries, receives partial aggregates from each processor, MERGEs and FINALizes them

- For non-aggregates, just UNIONs results

DB Parallel Processing vs General Parallelism

- Shared nothing partitioned parallelism is the dominant approach
- Hooray for the relational model!
  - Apps don't change when you parallelize system (physical data independence!).
  - Can tune, scale system without changing applications!
  - Can partition records arbitrarily, w/o synchronization
- Essentially no synchronization except setup & teardown
  - No barriers, cache coherence, etc.
  - DB transactions work fine in parallel



Next time: Distributed Transactions!