CS 6400 A

Final Review

12/01/25

Final Logistics

Final will be released Thursday Dec 4 at 3PM and due Friday Dec 5 at 9PM

- Open books and notes, closed Internet
- Unlimited time during the availability window. Submit finished exam on Gradescope

Clarification questions during exam

- Via private posts on Piazza
- Also check out the clarification question thread

The exam covers the entire class but will focus primarily on lectures not covered by the midterm. Contents EXCLUDED:

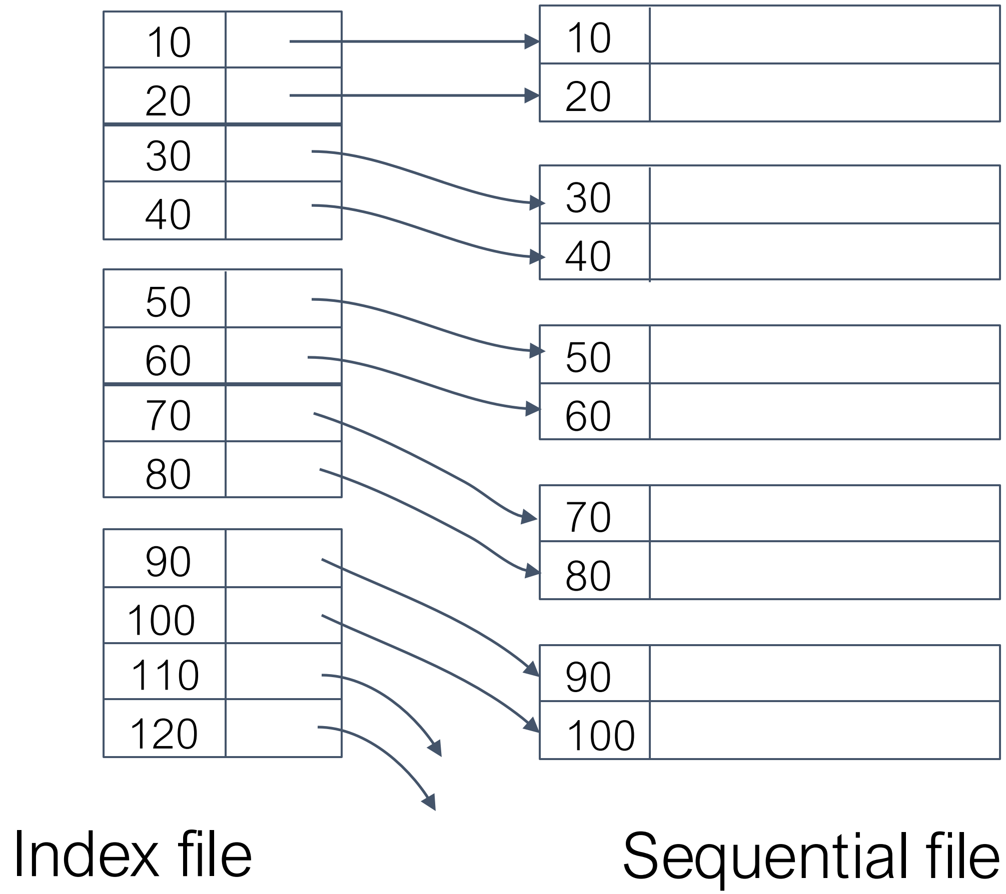
- lec 13: Multidimensional and Vector Indexes
- All lectures after lec 21 (Transaction Recovery)

Past Exam: available on canvas, under Files->Past Exams

Index Basics

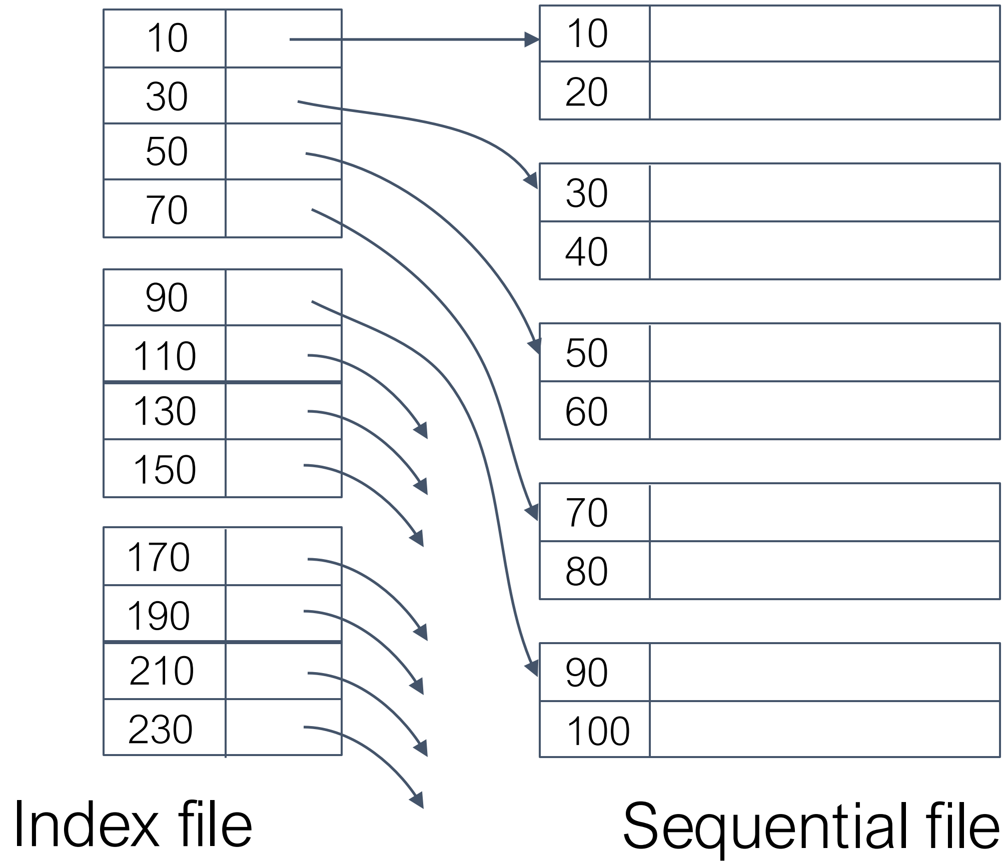
Dense index

- A sequence of blocks holding keys of records and pointers to the records



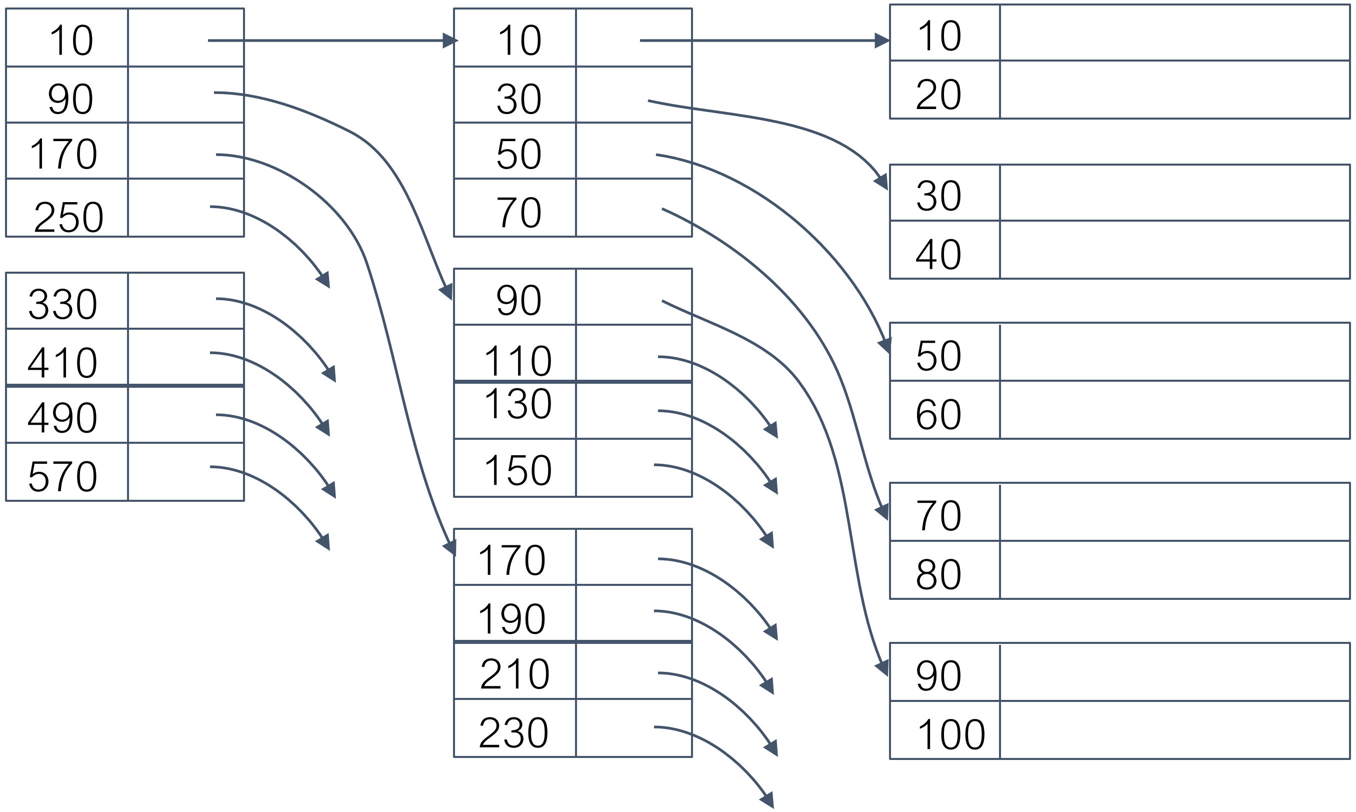
Sparse index

- Has one key-pointer pair per block of the data file
- Uses less space than dense index, but needs more time to find a record

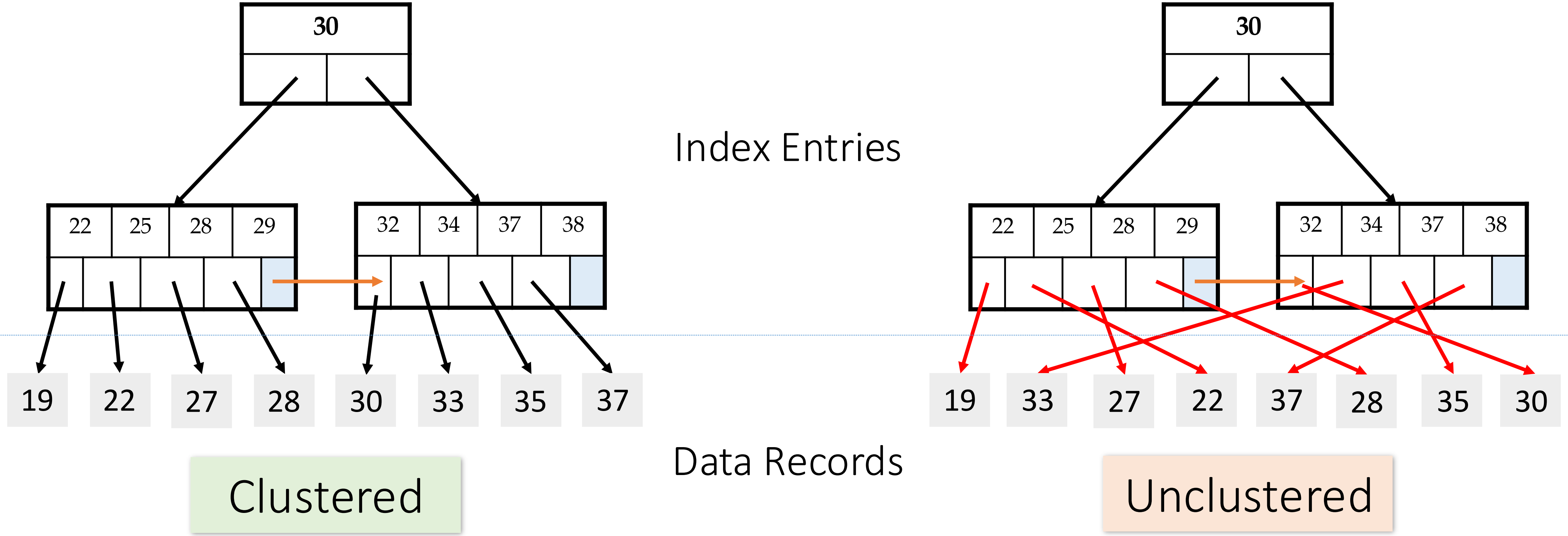


Multiple levels of index

If the index file is still large, add another level of indexing



Clustered vs. Unclustered Index



1 Random Access IO + Sequential IO
(# of pages of answers)

Random Access IO for each **value**
(i.e. # of tuples in answer)

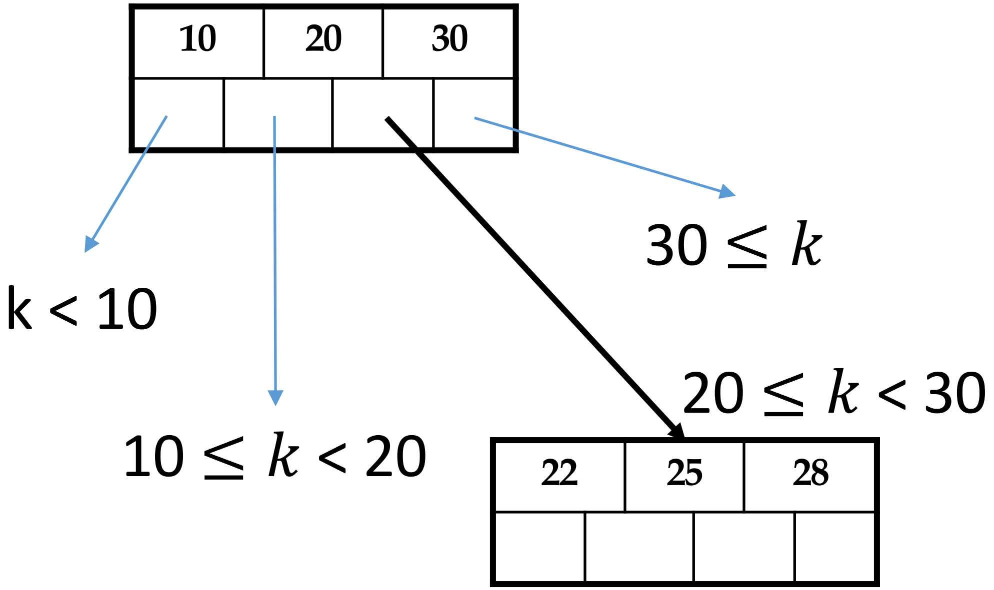
Clustered can make a ***huge*** difference for range queries!

B+ Tree

B+ Tree Basics

Parameter ***d*** = the degree

Non-leaf or *internal* node



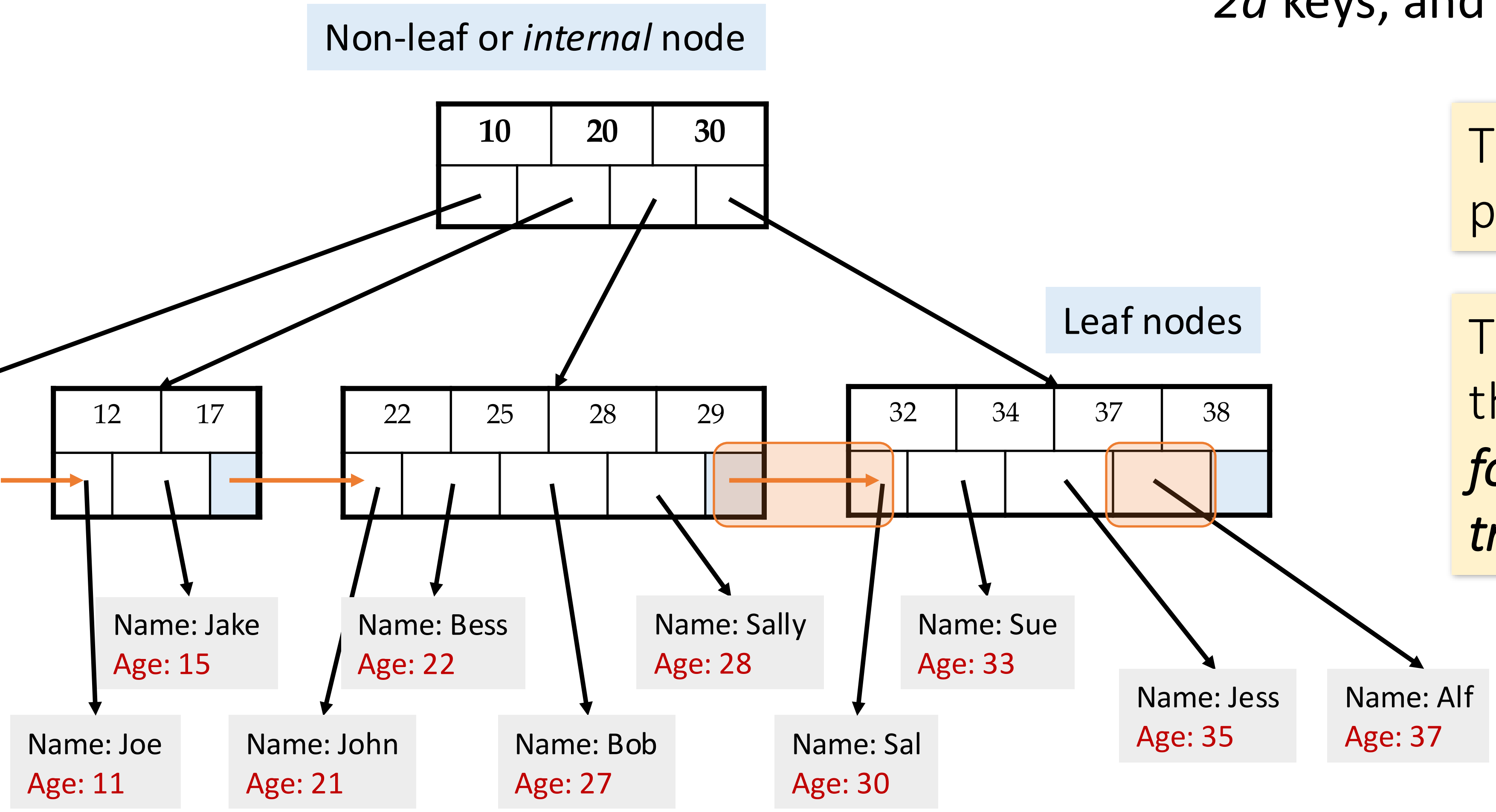
Each non-leaf ("interior") node has node has $\geq$ d and $\leq$ 2d keys*

The *k* keys in a node define *k+1* ranges

except for root node, which can have between* ***1* *and 2d keys*

For each range, in a *non-leaf* node, there is a **pointer** to another node with keys in that range

B+ Tree Basics

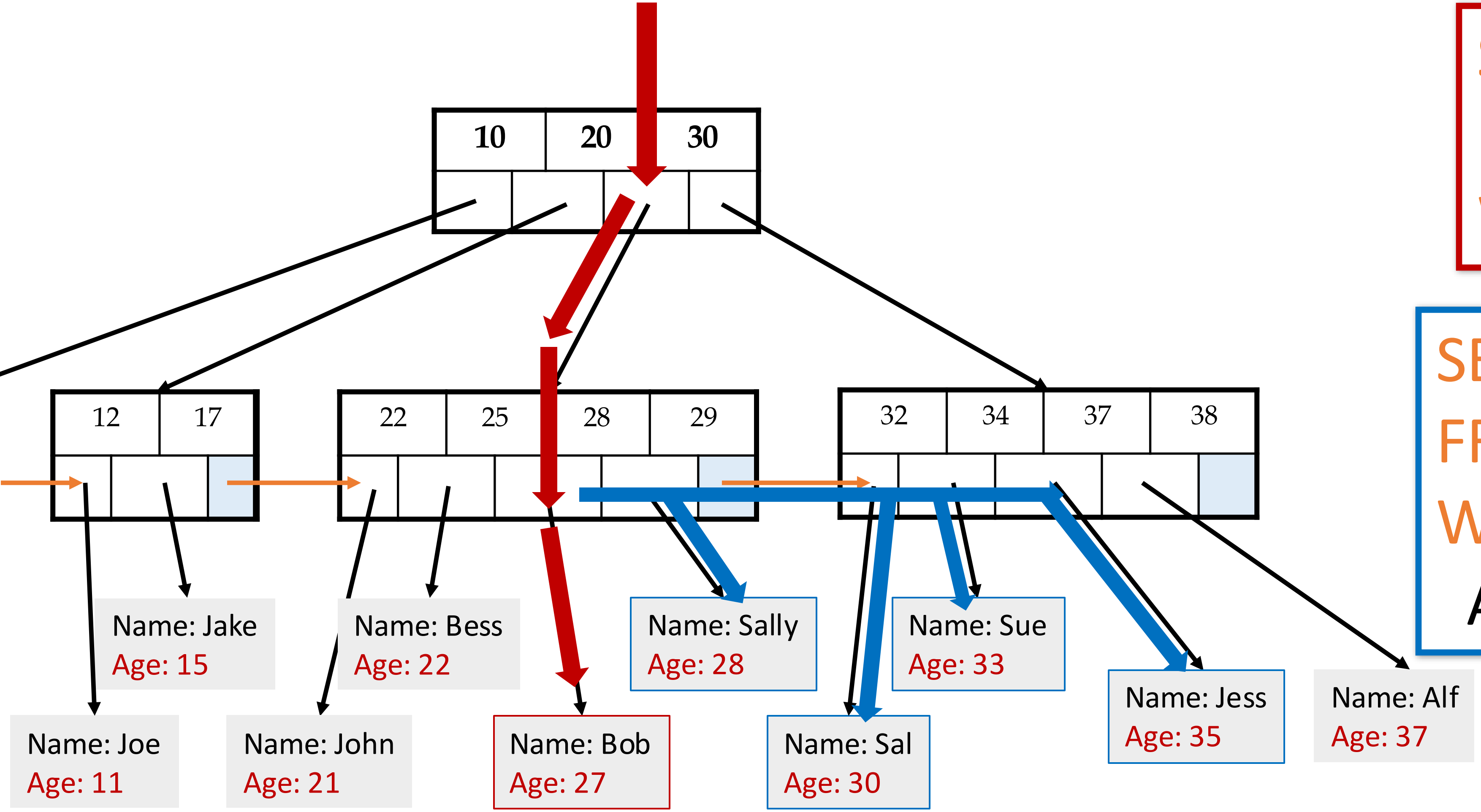


Leaf nodes also have between d and *2d* keys, and are different in that:

Their key slots contain pointers to data records

They contain a pointer to the next leaf node as well, ***for faster sequential traversal***

Searching a B+ Tree




```
SELECT name
FROM   people
WHERE  age = 27
```

```
SELECT name
FROM   people
WHERE  27 <= age
  AND  age <= 35
```

B+ Tree Cost Model

Note that exact search is just a special case of range search (R = 1)

Goal: Get the results set of a range (or exact) query with minimal IO

Key idea:

- A B+ Tree has high ***fanout ($d \sim= 10^2$-10^3)***, which means it is very shallow → we can get to the right root node within a few steps!
- Then just traverse the leaf nodes using the horizontal pointers

Details:

- One node per page (thus page size determines *d*)
- Fill only some of each node's slots (the ***fill-factor***) to leave room for insertions
- We can keep some levels of the B+ Tree in memory!

The fanout f is the number of pointers coming out of a node. Thus:

$$d + 1 \leq f \leq 2d + 1$$

Note that we will often approximate f as constant across nodes!

We define the ***height*** of the tree as counting the root node. Thus, *given constant fanout f*, a tree of height ***h*** can index f^h pages and has f^{h-1} leaf nodes

B+ Tree Cost Model

Given:	• Fill-factor ***F*** • ***B*** available pages in buffer • A B+ Tree over ***N*** pages • **f** is the average fanout	
Input:	A a range query.	
Output:	The ***R*** values that match	
IO COST:	$\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + \mathbf{Cost}(Out)$ *where* $B \geq \sum_{l=0}^{L_B-1} f^l$	***Depth of the B+ Tree:*** For each level of the B+ Tree we read in one node = one page ***# of levels we can fit in memory:*** These don't cost any IO! ***This equation*** is just saying that the sum of all the nodes for L_B levels must fit in buffer

Practice Question

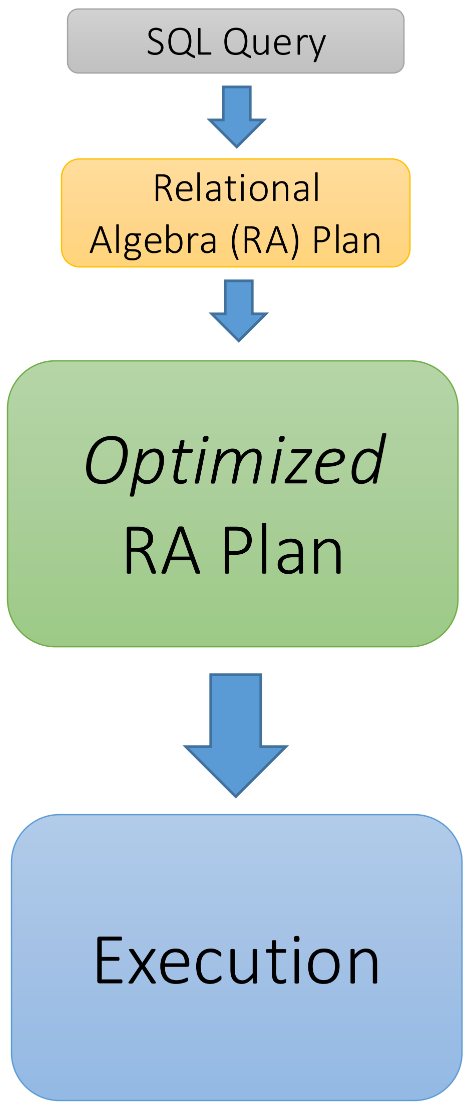
- 4. Indexing Problem
    - 4.2
    - 4.3

Query Optimization

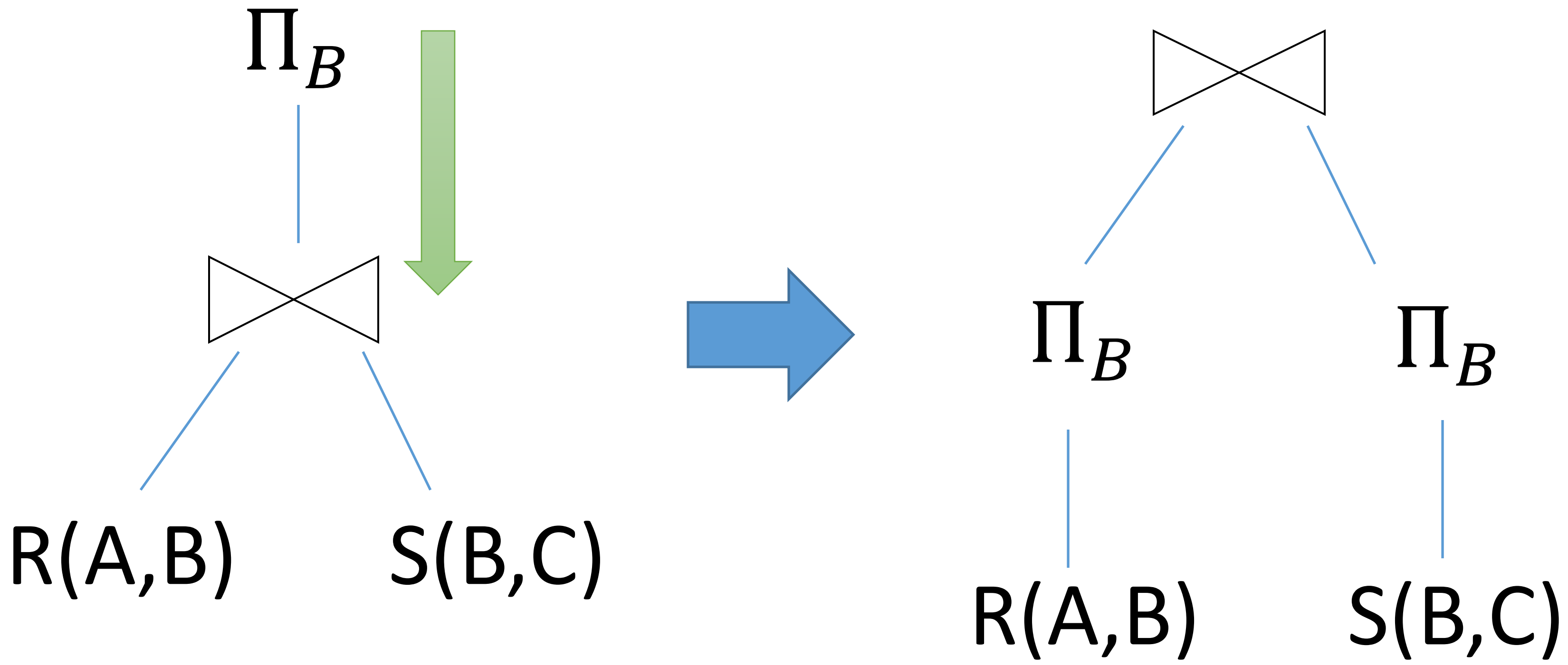
Logical vs. Physical Optimization

- **Logical optimization:**
  - Find equivalent plans that are more efficient
  - *Intuition: Minimize # of tuples at each step by changing the order of RA operators*

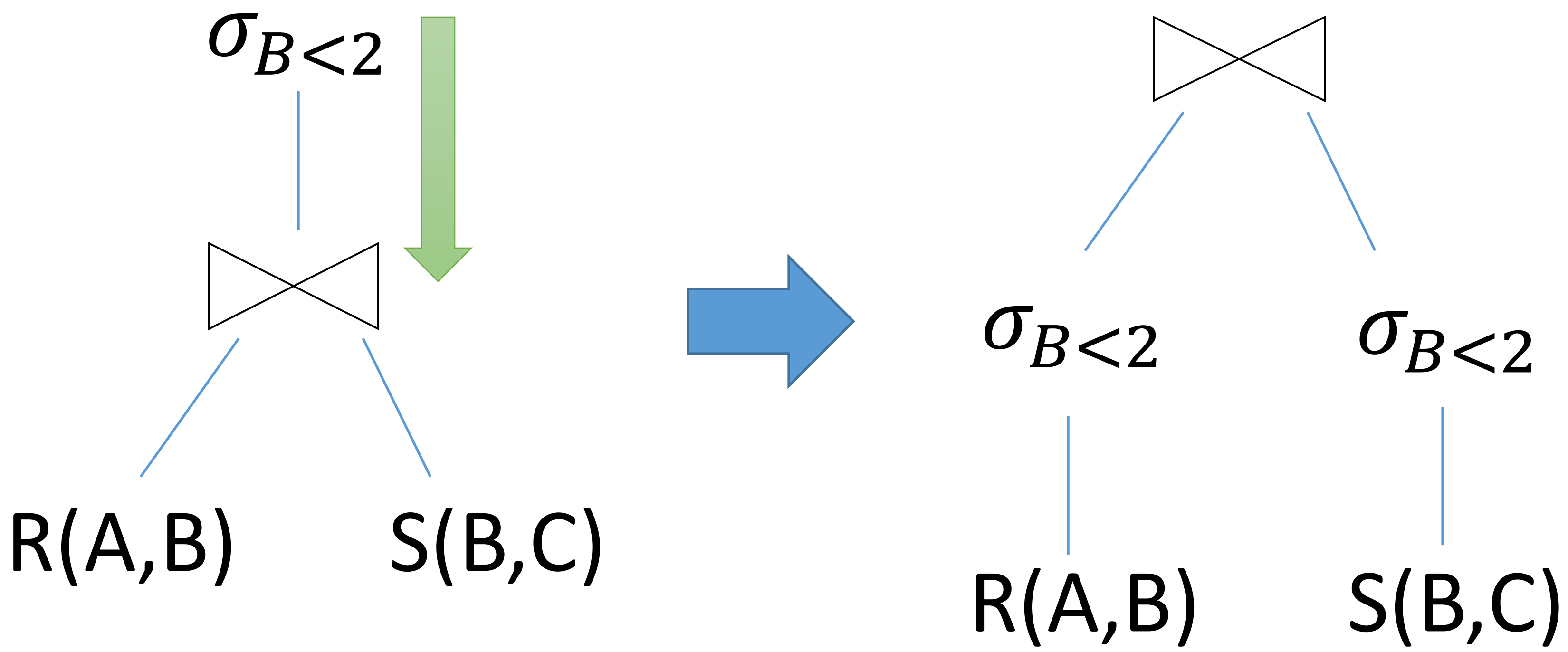
- **Physical optimization:**
  - Find algorithm with lowest IO cost to execute our plan
  - *Intuition: Calculate based on physical parameters (buffer size, etc.) and estimates of data size (histograms)*



Logical Optimization: "Pushing down" projection



Logical Optimization: "Pushing down" selection



RA commutators

The basic commutators:

- Push **projection** through **(1) selection**, **(2) join**
- Push **selection** through **(3) selection, (4) projection, (5) join**
- *Also:* Joins can be re-ordered!

Note that this is not an exhaustive set of operations

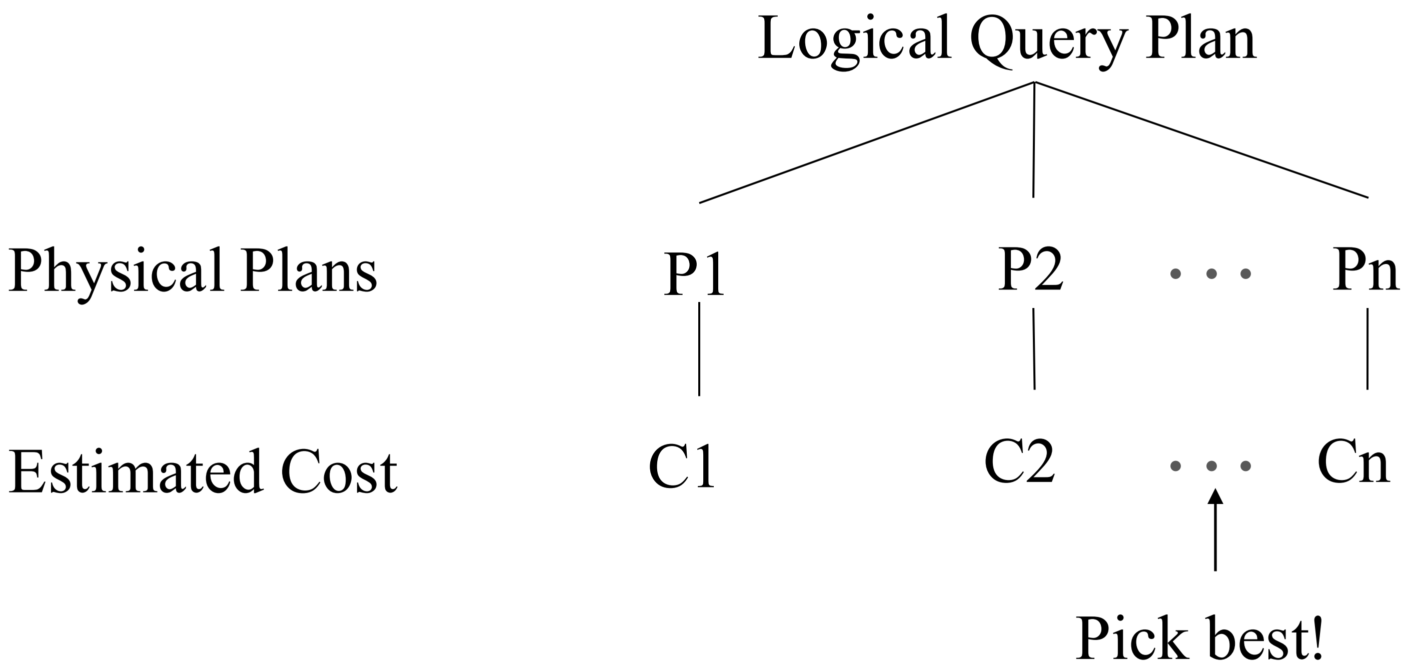
- This covers *local re-writes; global re-writes possible but much harder*

This simple set of tools allows us to greatly improve the execution time of queries by optimizing RA plans!

Practice Question

- 5.1 Logical Optimization

Select physical query plan



In general, there can be many possible physical plans

Query execution

$\pi_{starName}$ (On the fly)

$\sigma_{year\ =\ 2008\ AND\ studioName\ =\ \text{'Ghibli'}}$ (On the fly)

$\bowtie$ (Nested loop join)

StarsIn (File scan) Movies (File scan)

Machine Code (e.g., C)

The best physical plan is translated to actual machine code

Estimating the cost of a physical query plan

Step 1: Estimate the size of results

- Projection
- Selection
- Joins

Step 2: Estimate the # of disk I/O’s

Estimating size of join

$R(X,Y) \bowtie S(Y,Z)$

Two simplifying assumptions

- Containment of value sets: if $V(R,Y) \leq V(S,Y)$, then every Y-value of R is a Y-value of S
- Preservation of value sets: $V(R \bowtie S, X) = V(R, X)$

$\boldsymbol{Case\ 1}: V(R,Y) \geq V(S,Y)$
$\Rightarrow T(R \bowtie S) = T(R)T(S)/V(R,Y)$

$\boldsymbol{Case\ 2}: V(R,Y) < V(S,Y)$
$\Rightarrow T(R \bowtie S) = T(R)T(S)/V(S,Y)$

For each pair (r, s), we know that the Y-value of S is one of the Y-values of R by containment of value sets, so the probability of r having the same Y-value is 1/V(R,Y)

$$T(R \bowtie S) = T(R)T(S)/\max(V(R,Y),V(S,Y))$$

IO Cost Estimation via Histograms

- For **index selection**:
  - What is the cost of an index lookup?

- Also for **deciding which algorithm to use**:
  - Ex: To execute $\mathrm{R} \bowtie S$, which join algorithm should DBMS use?

  - **What if we want to compute $\boldsymbol{\sigma_{A>10}(\mathrm{R}) \bowtie \sigma_{B=1}(S)}$?**

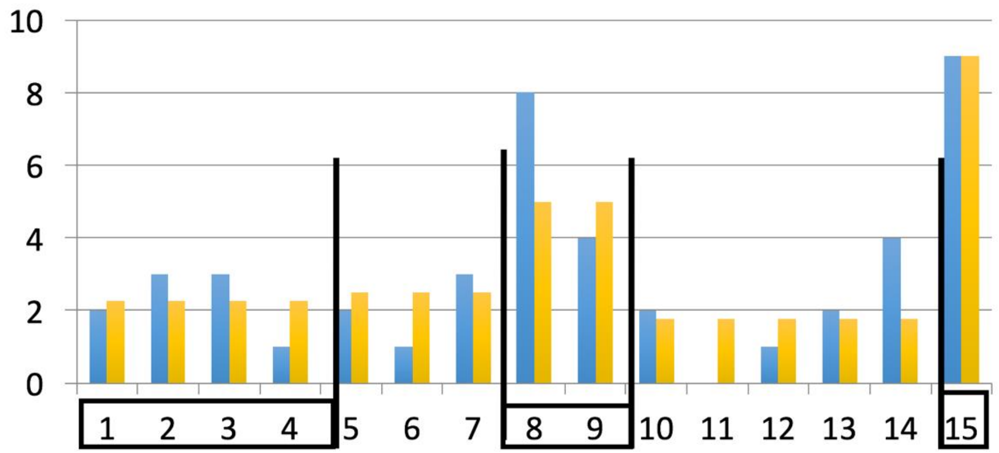
- In general, we will need some way to ***estimate intermediate result set sizes***

Histograms provide a way to efficiently store estimates of these quantities

Histogram types

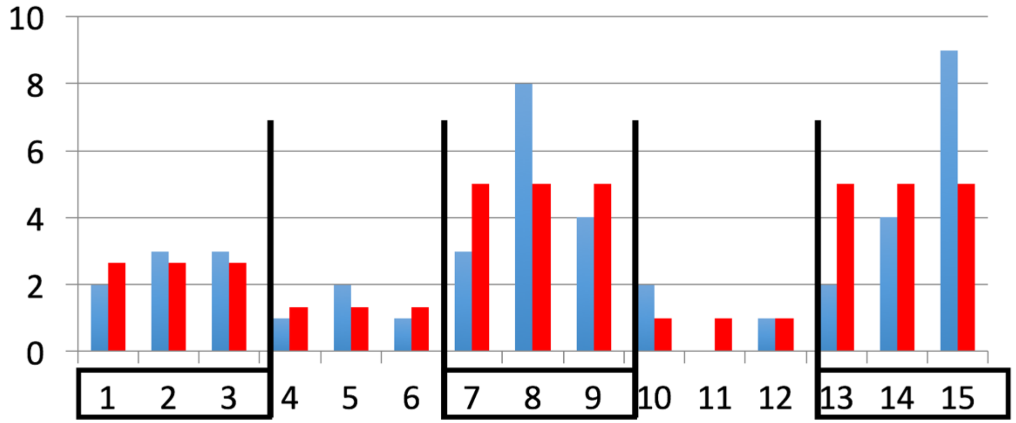
Equi-depth

All buckets contain roughly the same number of items (total frequency)



Equi-width

All buckets roughly the same width

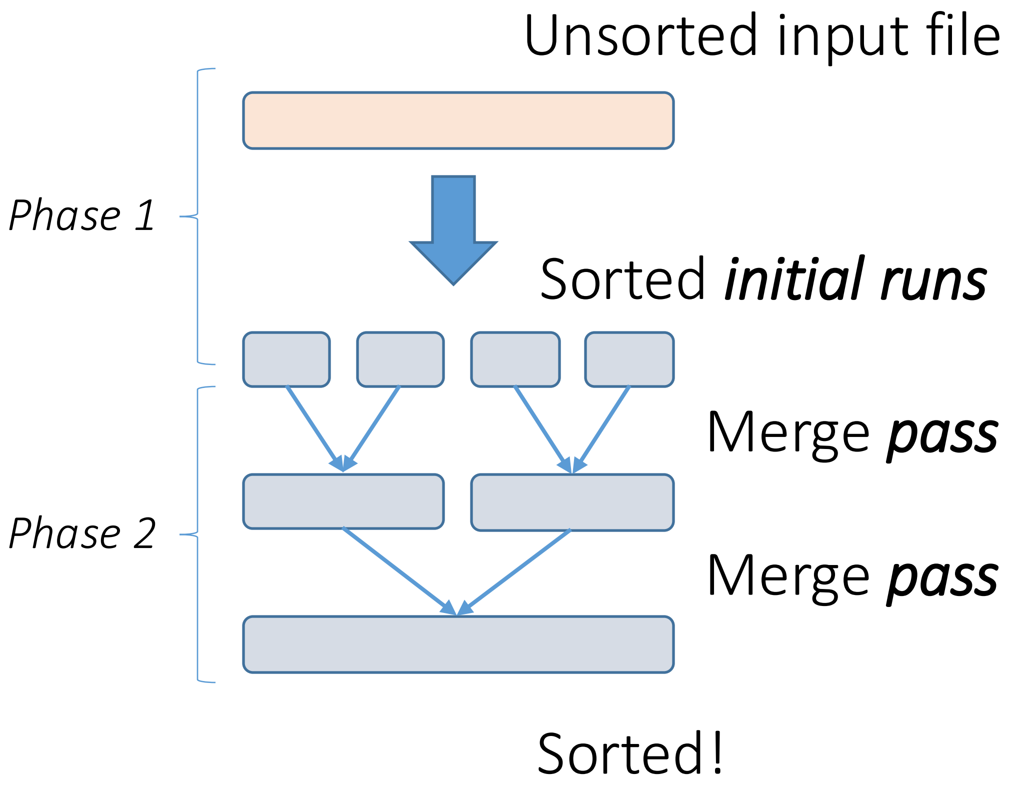


EMS and Join Algorithms

External Merge Sort Algorithm

- ***Goal:*** Sort a file that is much bigger than the buffer

- ***Key idea:***
  - *Phase 1:* Split file into smaller chunks ("initial runs") which can be sorted in memory
  - *Phase 2:* Keep merging (do "passes") using external merge algorithm until one sorted file!



External Merge Algorithm

Goal: Merge sorted files that are much bigger than buffer

- Basic algorithm
  - (B+1)-length initial runs
  - B-way merging
- Repacking optimization for longer initial runs
  - Effect: runs will have **~2(B+1)** length

$$2N(\lceil \log_2 N \rceil + 1)$$

Starting with runs of length 1

$$2N\left(\left\lceil \log_2 \frac{\boldsymbol{N}}{\boldsymbol{B+1}} \right\rceil + 1\right)$$

Starting with runs of length ***B+1***

$$2N\left(\left\lceil \log_B \frac{\boldsymbol{N}}{\boldsymbol{B+1}} \right\rceil + 1\right)$$

Performing ***B***-way merges

External Merge Sort Algorithm

Given:	***B+1*** buffer pages	
Input:	Unsorted file of length ***N*** pages	
Output:	The sorted file	
IO COST:	$2N\left(\left\lceil \log_B \left\lceil \frac{N}{B+1} \right\rceil \right\rceil + 1\right)$	***Phase 1:*** Initial runs of length B+1 are created • There are $\left\lceil \frac{N}{B+1} \right\rceil$ of these • The IO cost is 2N ***Phase 2:*** We do passes of B-way merge until fully merged • Need $\left\lceil \log_B \left\lceil \frac{N}{B+1} \right\rceil \right\rceil$ passes • The IO cost is 2N per pass

Join Algorithms: Overview

For $\mathrm{R} \bowtie S \text{ on } A$

- NLJ: An example of a *non*-IO aware join algorithm
- BNLJ: Big gains just by being IO aware & reading in chunks of pages!

Quadratic in P(R), P(S)
*I.e. O(P(R)*P(S))*

- SMJ: Sort R and S, then scan over to join!
- HJ: Partition R and S into buckets using a hash function, then join the (much smaller) matching buckets

Given sufficient buffer space, ***linear*** in P(R), P(S)
I.e. ~O(P(R)+P(S))

By only supporting equijoins & taking advantage of this structure!

Nested Loop Join (NLJ)

Compute $\mathrm{R} \bowtie S \ on \ A$:

```
for r in R:
  for s in S:
    if r[A] == s[A]:
      yield (r,s)
```

Note that IO cost based on number of *pages* loaded, not number of tuples!

<u>Cost:</u>

P(R) + T(R)*P(S) + OUT

1. Loop over the tuples in R
2. For every tuple in R, loop over all the tuples in S
3. Check against join conditions
4. **Write out (to page, then when page full, to disk)**

Have to read *all of S* from disk for *every tuple in R!*

Block Nested Loop Join (BNLJ)

Given ***B+1*** pages of memory

Compute R $\bowtie S \ on \ A$:

```
for each B-1 pages pr of R:
  for page ps of S:
    for each tuple r in pr:
      for each tuple s in ps:
        if r[A] == s[A]:
          yield (r,s)
```

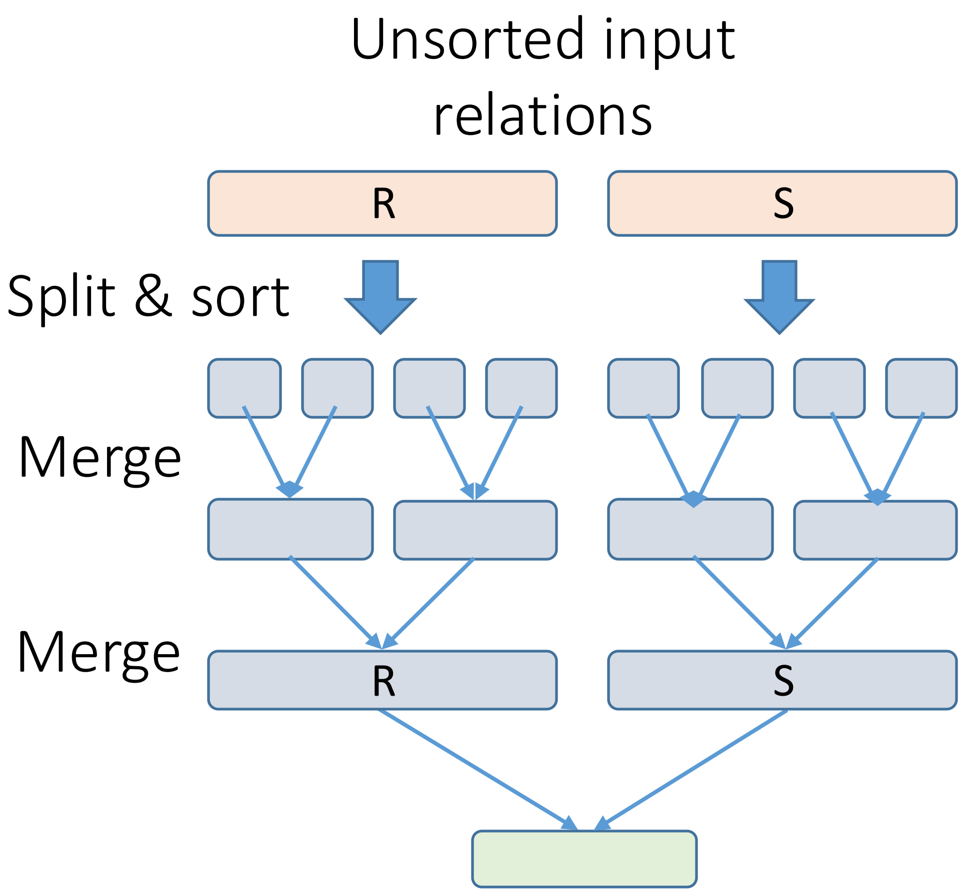
<u>Cost:</u>

$$P(R) + \frac{P(R)}{B-1} P(S) + \text{OUT}$$

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)
2. For each (B-1)-page segment of R, load each page of S
3. Check against the join conditions
4. **Write out**

Again, ***OUT*** could be bigger than P(R)*P(S)… but usually not that bad

Basic SMJ: Total cost



Cost of SMJ is **cost of sorting** R and S…

Plus the **cost of scanning**: ~P(R)+P(S)

- Because of *backup*: in worst case P(R)*P(S); but this would be very unlikely

Plus the **cost of writing out**

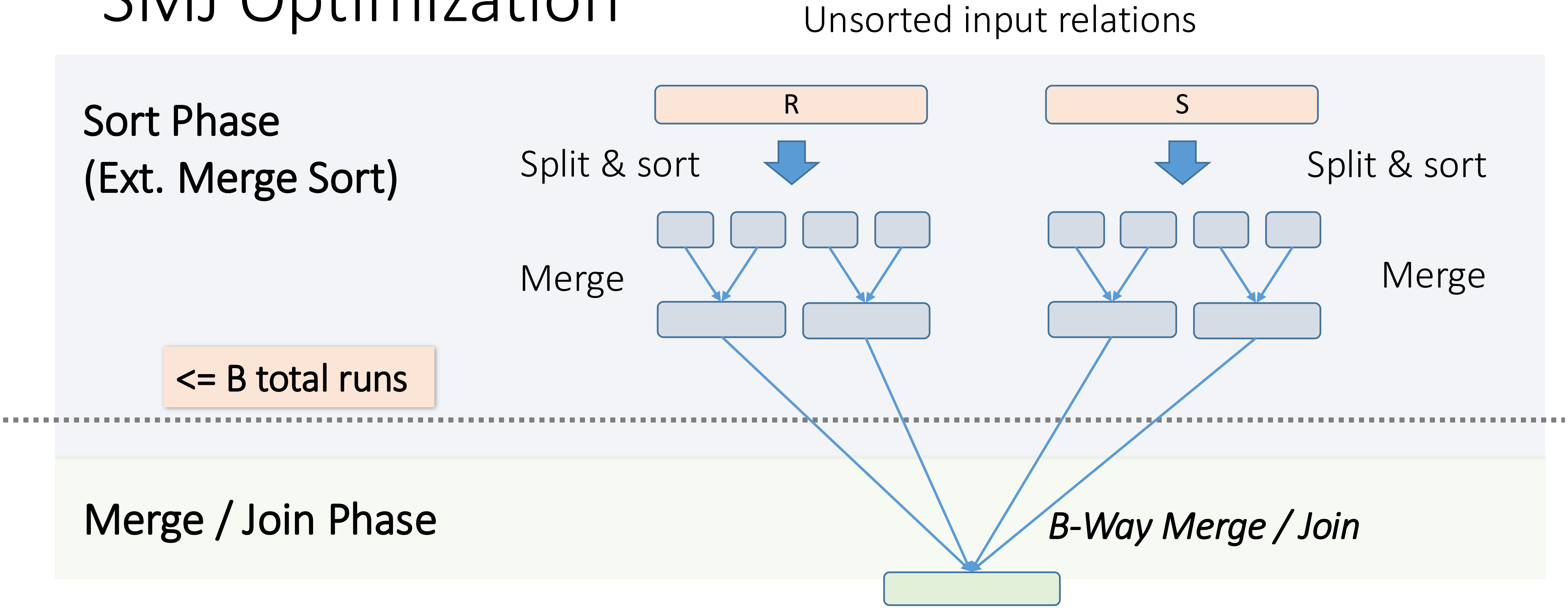
$$\sim \mathrm{Sort}(P(R)) + \mathrm{Sort}(P(S)) + P(R) + P(S) + \mathrm{OUT}$$

Recall: $\mathrm{Sort}(N) \approx 2N\left(\left\lceil \log_B \frac{\boldsymbol{N}}{\boldsymbol{B+1}} \right\rceil + 1\right)$

Note: this is WITHOUT the repacking optimization

SMJ Optimization

Given ***B+1*** buffer pages



Optimization: Keep merging until (# of runs of R) + (# of runs of S) $\leq B$, then we are ready to complete the join in one pass

Hash Join: High-level procedure

Given ***B+1*** buffer pages

To compute $\mathrm{R} \bowtie S \ on \ A$:

1. **Partition Phase:** Using one (shared) hash function $\boldsymbol{h_B}$ per pass partition R *and* S into ***B*** buckets.
   - Each phase creates B more buckets that are a factor of B smaller.
   - Repeatedly partition with a new hash function
   - Stop when *all buckets for one relation are smaller than B-1 pages*

Each pass takes $2(\mathrm{P}(\mathrm{R}) + \mathrm{P}(\mathrm{S}))$

2. **Join Phase:** Take pairs of buckets whose tuples have the same values for ***h***, and join these
   - Use BNLJ here for each matching pair.

$\mathrm{P}(\mathrm{R}) + \mathrm{P}(\mathrm{S}) + \mathrm{OUT}$

Join phase cost can be worse due to hash collision and skew

SMJ vs. HJ

Given enough memory, both SMJ and HJ have performance:

$$\sim 3(P(R)+P(S)) + OUT$$

"Enough" memory =

- SMJ: $B^2 > \max\{P(R), P(S)\}$
- HJ: $B^2 > \min\{P(R), P(S)\}$

Hash Join superior if relation sizes differ greatly.

Practice Question

- 6. Query Optimization

Transaction

Transactions: Basic Definition

A **transaction ("TXN")** is a sequence of one or more ***operations*** (reads or writes) which reflects ***a single real-world transition***.

In the real world, a TXN either happened completely or not at all

```
START TRANSACTION
        UPDATE Product
        SET Price = Price – 1.99
        WHERE pname = 'Gizmo'
COMMIT
```

Motivation for Transactions

Grouping user actions (reads & writes) into *transactions* helps with two goals:

1. **Recovery & Durability**: Keeping the DBMS data consistent and durable in the face of crashes, aborts, system shutdowns, etc.

2. **Concurrency:** Achieving better performance by parallelizing TXNs *without* creating anomalies

Transaction Properties: ACID

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- Isolated
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- Durable
  - Once a txn has committed, its effects remain in the database

ACID continues to be a source of great debate!

Comparison of SQL isolation levels

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
READ UNCOMMITTED	✓	✓	✓
READ COMMITTED	🚫	✓	✓
REPEATABLE READ	🚫	🚫	✓
SERIALIZABLE	🚫	🚫	🚫

Why Interleave TXNs?

- Interleaving TXNs might lead to anomalous outcomes… why do it?

- Several important reasons:
  - Individual TXNs might be *slow*- don't want to block other users during!

  - Disk access may be *slow*- let some TXNs use CPUs while others accessing disk!

All concern large differences in ***performance***

Scheduling Definitions

- A **<u>serial schedule</u>** is one that does not interleave the actions of different transactions

- A and B are **<u>equivalent schedules</u>** if, ***for any database state***, the effect on DB of executing A **is identical to** the effect of executing B

- *A* **<u>serializable schedule</u>** is a schedule that is equivalent to ***some*** serial execution of the transactions.

The word "**some**" makes this definition powerful & tricky!

Conflicts: Anomalies with Interleaved Execution

Conditions for conflicts:

- The operations must belong to **different transactions** (no conflict within the same transaction).
- The operations must access the **same database object**
- At least one of the operations must be a **write** operation.

Types of conflicts:

- Write-Read (WR)
- Read-Write (RW)
- Write-Write (WW)

<u>Implication for schedules:</u>
A pair of consecutive actions that cannot be interchanged without changing behavior

DB isolation levels define which types of conflicts a database will prevent or allow.

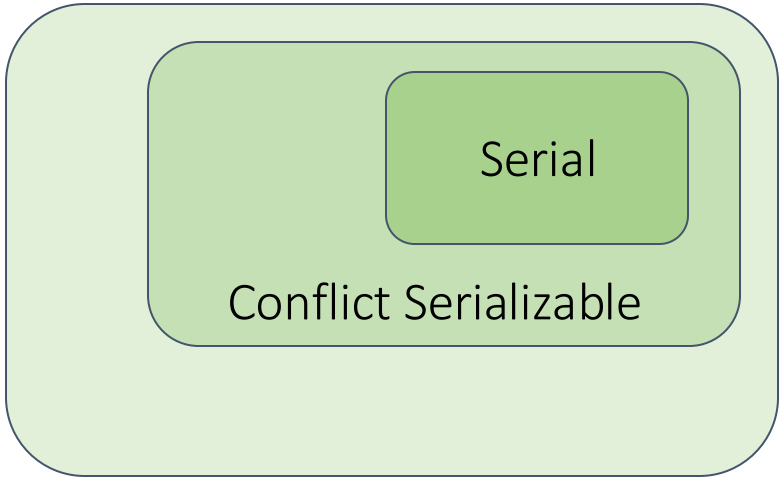
Characterizing Schedules based on Serializability (2)

Conflict equivalent

- Two conflict equivalent schedules have the same effect on a database
- All pairs of conflicting actions are in same order
- one schedule can be obtained from the other by swapping “non-conflicting” actions
  - either on two different objects
  - or both are read on the same object

Conflict serializable

- A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S’.



Testing for conflict serializability

Through a precedence graph:

- Looks at only read_Item (X) and write_Item (X) operations
- Constructs a precedence graph (serialization graph) - a graph with directed edges
- An edge is created from Ti to Tj if one of the operations in Ti appears before a conflicting operation in Tj
- The schedule is serializable if and only if the precedence graph <u>has no cycles</u>.

Practice Question

- 1. Serializability
  - 1.1 Conflicts and Serializability
  - 1.2 Conflict Serializable

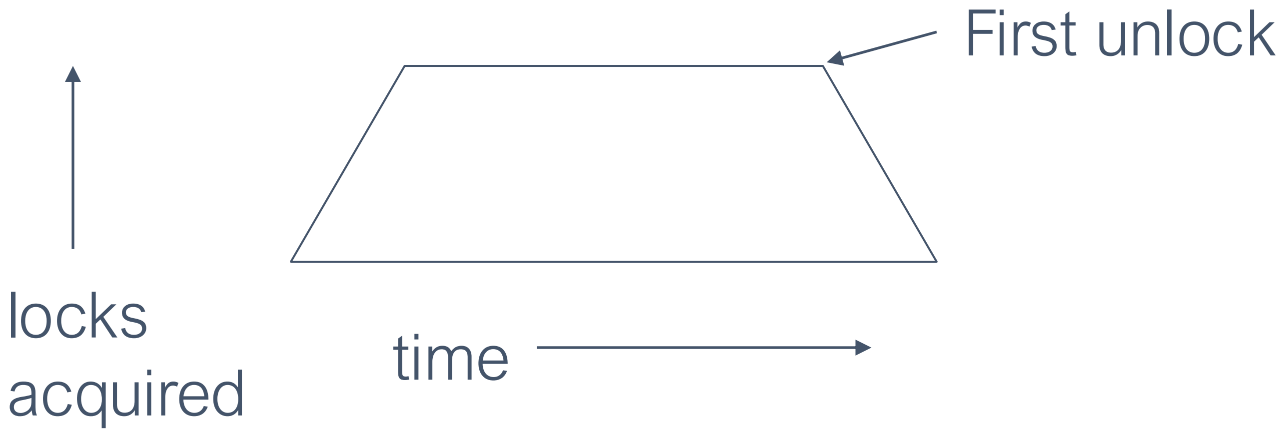
Two-phase locking (2PL)

TXNs obtain:

- An **X (*exclusive*) lock** on object before **writing**.
  - If a TXN holds, no other TXN can get a lock (S or X) on that object.
- An **S (*shared*) lock** on object before **reading**
  - If a TXN holds, no other TXN can get *an X lock* on that object

Two-phase locking (2PL)

- In every transaction, all lock actions precede all unlock actions
- Guarantees a legal schedule of consistent transactions is conflict serializable



Practice Question

- 2.1. 2PL Feasibility

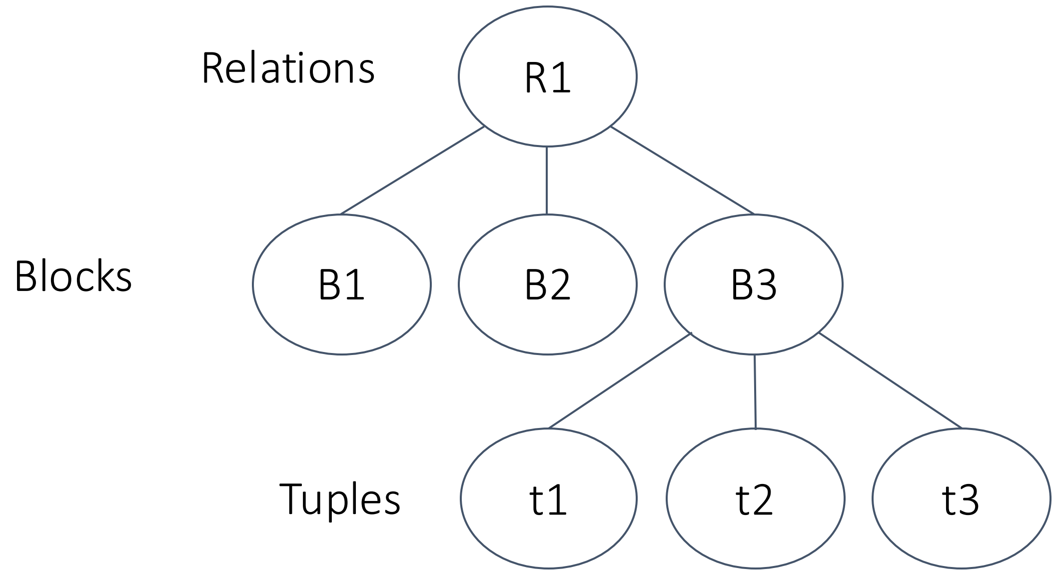
Locking with several modes

- Compatibility matrix

		Lock requested	
		S	X
Lock held in mode	S	Yes	No
	X	No	No

Locking with Multiple Granularities

- Ordinary locks: S and X
- Warning locks: I (shows intention to lock)



Compatibility matrix

- For shared, exclusive, and intention locks

Requestor

Holder		IS	IX	S	X
	IS	Yes	Yes	Yes	No
	IX	Yes	Yes	No	No
	S	Yes	No	Yes	No
	X	No	No	No	No

Practice Question

- 2.3 Locking with multiple granularities

Optimistic Concurrency Control

Optimistic methods

- Two methods: validation (covered next), and timestamping
- Assume no unserializable behavior
- Abort transactions when violation is apparent
- may cause transactions to rollback

In comparison, locking methods are pessimistic

- Assume things will go wrong
- Prevent nonserializable behavior
- Delays transactions but avoids rollbacks

> Optimistic approaches are often better than lock when transactions have low interference (e.g., read-only)

To validate, scheduler maintains three sets

START: set of transactions that started, but have not validated

- START(T), the time at which T started

VAL: set of transactions that validated, but not yet finished write phase

- VAL(T), time at which T is imagined to execute in the hypothetical serial order of execution

FIN: set of transactions that have completed write phase

- FIN(T), the time at which T finished.

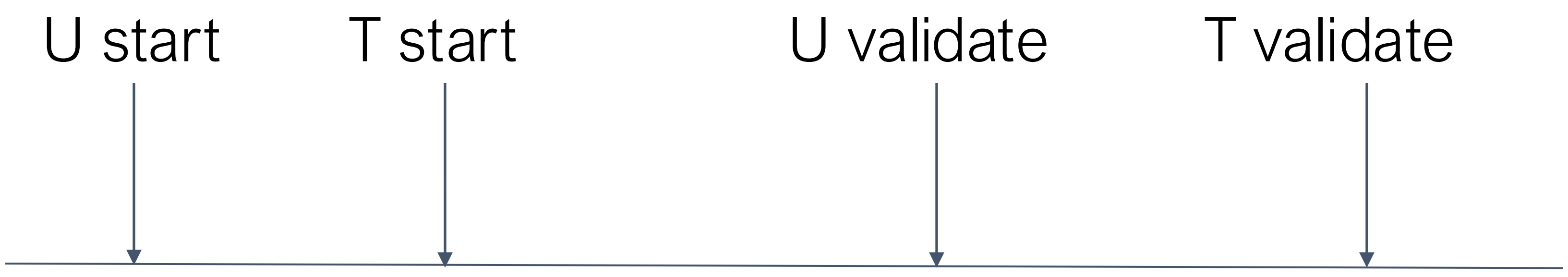
Validation rules (assume U validated)

Rule 1: if FIN(U) > START(T), RS(T) ∩ WS(U) = ∅

WS(U) = {A, B}

RS(T) = {B, C}

This violates rule 1 because T may be reading B before U writes B

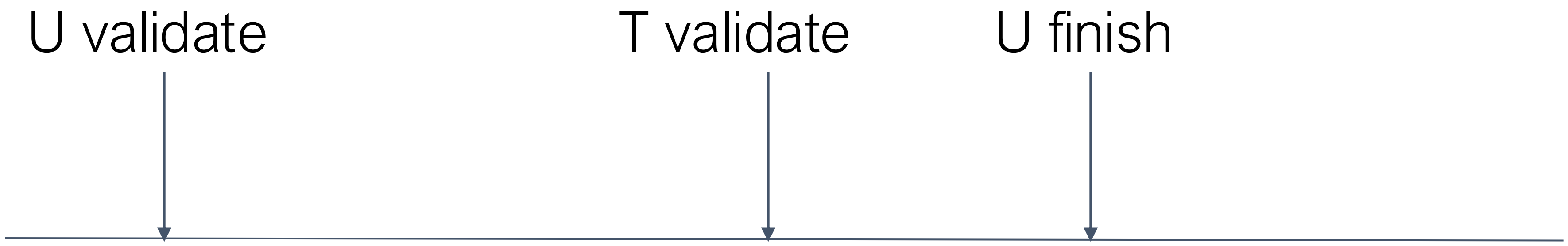


Validation rules (assume U validated)

Rule 2: if FIN(U) > VAL(T), WS(T) ∩ WS(U) = ∅

WS(U) = {A, B}   WS(T) = {B, C}

This violates rule 2 because T may write B before U writes B



Practice Question

- 2.2 Optimistic Concurrency Control

Basic Idea: (Physical) Logging

- Record UNDO information for every update!
  - Sequential writes to log
  - Minimal info (diff) written to log

- The **log** consists of **an ordered list of actions**
  - Log record contains:

    <T, X, v, w> : T changed value of X from v to w

Why do we need logging for atomicity?

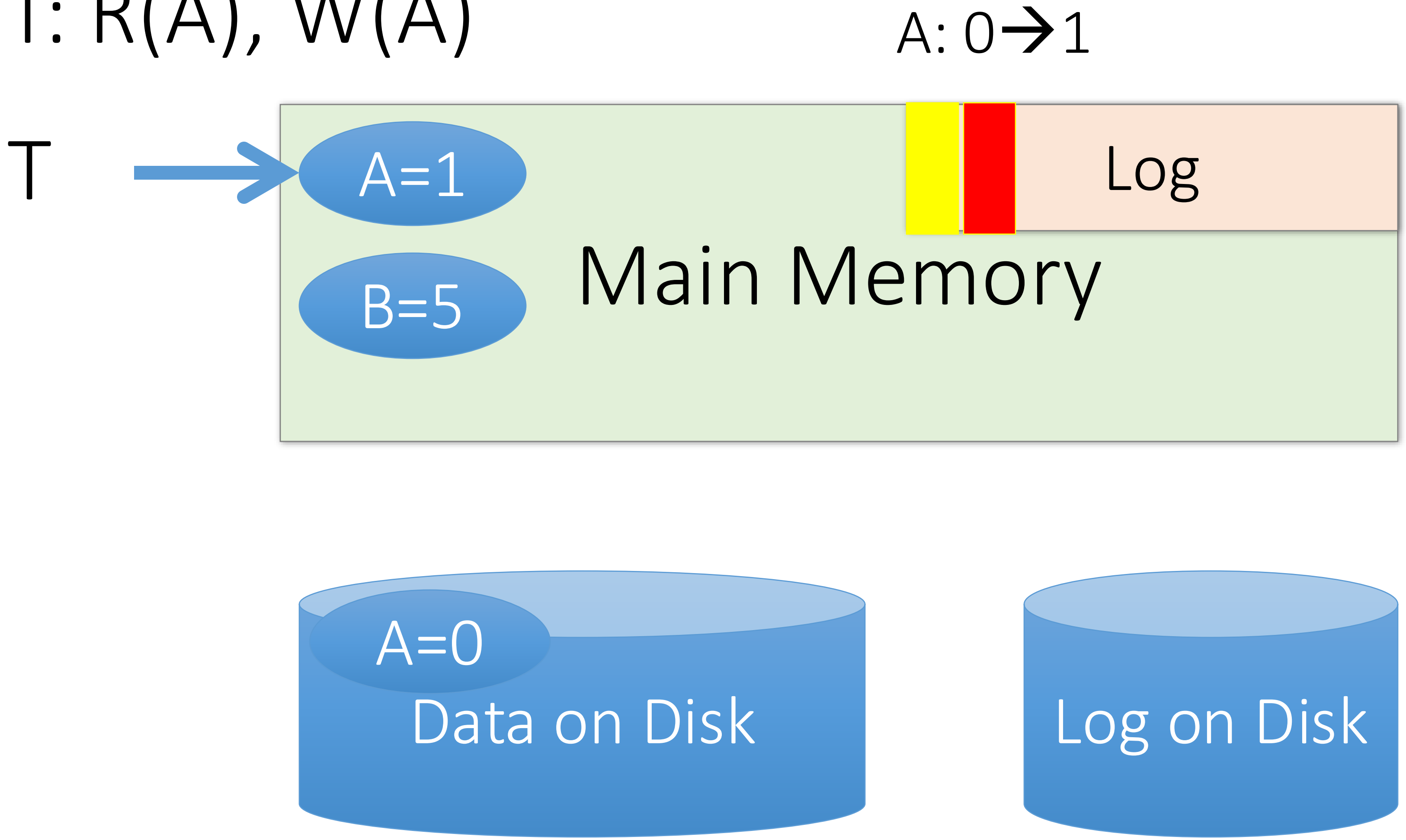
- Couldn’t we just write TXN to disk **only** once whole TXN complete?
  - Then, if abort / crash and TXN not complete, it has no effect- atomicity!
  - *With unlimited memory and time, this could work…*

- However, we **need to log partial results of TXNs** because of:
  - Memory constraints (enough space for full TXN??)
  - Time constraints (what if one TXN takes very long?)

We need to write partial results to disk!
…And so we need a **log** to be able to ***undo*** these partial results!

Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)



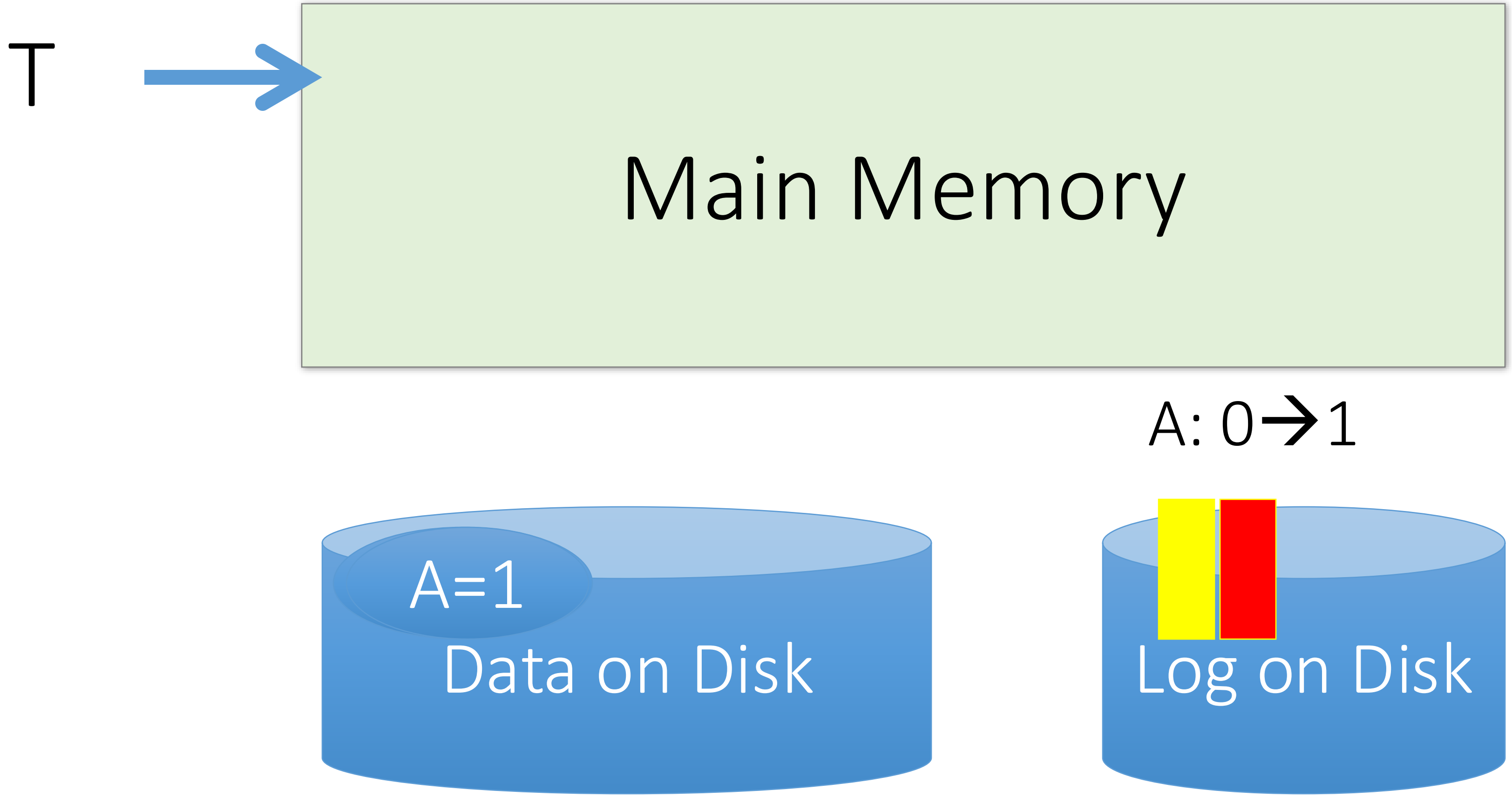
This time, let's try committing ***after* we've written log to disk but *before* we've written data to disk**... this is WAL!

OK, Commit!

If we crash now, is T durable?

Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)



This time, let's try committing ***after* we've written log to disk but *before* we've written data to disk**... this is WAL!

OK, Commit!

If we crash now, is T durable?

USE THE LOG!

Write-Ahead Logging (WAL)

- DB uses **Write-Ahead Logging (WAL)** Protocol:

Each **update** is logged! Why not reads?

1. Must *force log record* for an update *before* the corresponding data page goes to storage

→ Atomicity

2. Must *write all log records* for a TX *before commit*

→ Durability

Undo logging

- Idea: Undo incomplete transactions, and ignore committed ones

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Undo log format:

<*T*, *X*, *v*>: T updated database element X whose old value is v

Recovery using undo logging

- Simplifying assumption: use entire log, no matter how long

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Crash

Recovery

A = 16
B = 16

Ignore (*T* was committed)

Ignore (*T* was committed)

Observe <COMMIT *T*> record

Nonquiescent checkpointing

- Motivation: avoid shutting down system while checkpointing
- Checkpoint all active transactions, but allow new transactions to enter system

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>
<START CKPT (T1, T2)>
<T2, *C*, 15>
<START T3>
<T1, *D*, 20>
<COMMIT T1>
<T3, E, 25>
<COMMIT T2>
<END CKPT>
<T3, F, 30>

Nonquiescent checkpointing

- Motivation: avoid shutting down system while checkpointing
- Checkpoint all active transactions, but allow new transactions to enter system

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>
<START CKPT (T1, T2)>
<T2, *C*, 15>
<START T3>
<T1, *D*, 20>
<COMMIT T1>
<T3, E, 25>
<COMMIT T2>
<END CKPT>
<T3, F, 30>

Crash

If we first meet <END CKPT>, only need to recover until <START CKPT (T1, T2)>

Nonquiescent checkpointing

- Motivation: avoid shutting down system while checkpointing
- Checkpoint all active transactions, but allow new transactions to enter system

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>
<START CKPT (T1, T2)>
<T2, *C*, 15>
<START T3>
<T1, *D*, 20>
Crash
<COMMIT T1>
<T3, E, 25>
<COMMIT T2>
<END CKPT>
<T3, F, 30>

If we first meet <START CKPT (T1, T2)>, only need to recover until <START T1>

Redo logging

Redo logging ignores incomplete transactions and repeats committed ones

- Undo logging cancels incomplete transactions and ignores committed ones

<*T*, *X*, *v*> now means *T* wrote new value *v* for database element *X*

One rule: all log records (e.g., <*T*, *X*, *v*> and <COMMIT *T*>) must appear on disk before modifying any database element *X* on disk

Recovery with redo logging

- Scan log forward and redo committed transactions

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
$t := t * 2$	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 16>
READ(*B*, *t*)	8	16	8	8	8	
$t := t * 2$	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 16>
						<COMMIT *T*>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	

Crash

Recovery

A = 16
B = 16

Nonquiescent checkpointing for redo log

- Write to disk all DB elements modified by committed transactions

<START T1>
<T1, *A*, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 10>
<START CKPT (T2)>
<T2, *C*, 15>
<START T3>
<T3, *D*, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

Write to disk all DB elements by transactions that already committed when START CKPT was written to log (i.e., T1)

Nonquiescent checkpointing for redo log

- After crash, redo committed transactions that either started after START CKPT or were active during START CKPT

<START T1>
<T1, *A*, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 10>
<START CKPT (T2)>
<T2, *C*, 15>
<START T3>
<T3, *D*, 20>
<END CKPT>
<COMMIT T2>
Crash
<COMMIT T3>

Undo/redo logging

More flexible than undo or redo logging in ordering actions

$<T, X, v, w>$: T changed value of X from v to w

One rule: $<T, X, v, w>$ must appear on disk before modifying X on disk

Nonquiescent checkpointing for undo/redo logging

- Simpler than other logging methods

<START T1>
<T1, *A*, 4, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 9, 10>
<START CKPT (T2)>
<T2, *C*, 14, 15>
<START T3>
<T3, *D*, 19, 20>
<END CKPT>

Write to disk all the buffers that are dirty

Nonquiescent checkpointing for undo/redo logging

- After a crash, redo committed transactions, and undo uncommitted ones

<START T1>
<T1, *A*, 4, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 9, 10>
<START CKPT (T2)>
<T2, *C*, 14, 15>
<START T3>
<T3, *D*, 19, 20>
<END CKPT>
<COMMIT T2>
Crash
<COMMIT T3>

Redo T2 by setting C to 15 on disk
(No need to set B to 10 thanks to CKPT)
Undo T3 by setting D to 19 on disk

Practice Question

- 3. Recovery